

Blockpraktikum Zeitreihenanalyse mit R

20. Mai 2010

Sebastian Mentemeier und Matti Schneider

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart und R Umgebung
- 3 Datenstrukturen
 - Vektoren
 - Datentabellen
 - Matrizen
 - Arrays, Listen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen
- 5 Plotten in R
- 6 Wahrscheinlichkeitsverteilungen

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart und R Umgebung
- 3 Datenstrukturen
 - Vektoren
 - Datentabellen
 - Matrizen
 - Arrays, Listen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen
- 5 Plotten in R
- 6 Wahrscheinlichkeitsverteilungen

Ablauf

Nächste Woche Dienstag, 25.05 bis Freitag, 28.05

- 10h - 12h: Theorie im **SR2**
- 13h30 - 15h: Besprechung der Übungsaufgaben vom Vortag im **SRA**
- 15h - 18h: betreute Bearbeitung der Übungsaufgaben im **SRA**
- Es darf/soll in 2er Gruppen gearbeitet werden, um die Aufgaben zu lösen
- Materialien (Übungszettel, Folien, ...) befinden sich auf der Praktikums-Homepage: <http://wwwmath.uni-muenster.de/statistik/lehre/SS10/BlockprakZeit/>

Literatur



Skript zum Blockpraktikum

<http://wwwmath.uni-muenster.de/statistik/lehre/SS10/BlockprakZeit/Zeitreihenanalyse.pdf>



Silke Ahlers

Einführung in die Statistik mit R

<http://wwwmath.uni-muenster.de/statistik/lehre/SS10/BlockprakZeit/Skript.pdf>



Paul Cowpertwait und Andrew Metcalfe

Introductory Time Series with R

Springer



Peter Brockwell und Richard Davis

Time Series: Theory and Methods

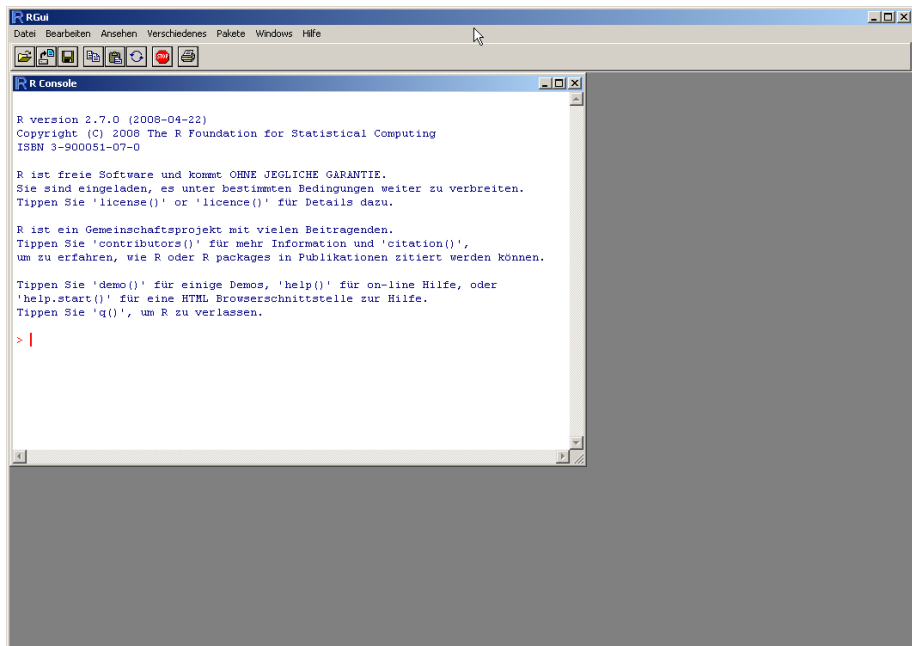
Springer

Gliederung

- ① Hinweise zum Praktikum
- ② Programmstart und R Umgebung
- ③ Datenstrukturen
 - Vektoren
 - Datentabellen
 - Matrizen
 - Arrays, Listen, ...
- ④ Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen
- ⑤ Plotten in R
- ⑥ Wahrscheinlichkeitsverteilungen

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.



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/SS10/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 angegeben, 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

- 1 Hinweise zum Praktikum
- 2 Programmstart und R Umgebung
- 3 **Datenstrukturen**
 - Vektoren
 - Datentabellen
 - Matrizen
 - Arrays, Listen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen
- 5 Plotten in R
- 6 Wahrscheinlichkeitsverteilungen

Vektorenerstellung

- Vektoren erzeugt man in R mit der Funktion `c`
- Die Erzeugung eines Vektors mit den Daten x_1, \dots, x_n erhält man durch Eingabe von `c(x_1, \dots, 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 zugeordnet 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 geschieht 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 geschieht 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

```
> Messung[Messung>6]
> 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 $v = (v_1, \dots, v_n)$ und $w = (w_1, \dots, w_m)$
- Für $m = n$ geschieht die Auswertung komponentenweise
- Ist n ein Vielfaches von m , so geschieht die Auswertung zyklisch, d. h.
 $v + w = (v[1] + w[1], \dots, v[m] + w[m], v[m+1] + w[1], \dots, v[n] + w[m])$

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 eingeschlossen 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 3. Zeileneintrag der 2. 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:

<code>read.table("Pfad")</code>	liest externen Datensatz ein
<code>read.csv("Pfad")</code>	liest durch Kommata getrennte Spalten
<code>read.csv2("Pfad")</code>	liest durch Semikola getrennte Spalten, Komma als Dezimalkomma
<code>read.delim("Pfad")</code>	liest Tab-getrennte Spalten
<code>read.delim2("Pfad")</code>	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 Matrizenmultiplikation $A \%*\% B$, die Determinantenberechnung $\det(A)$ oder das Lösen linearer Gleichungssysteme $Ax = b$ mit $\text{solve}(A,b)$
- 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))
```

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`)
- *Zeitreihe* = Vektor mit Zeitangabe; dazu am Dienstag mehr (R Befehl: `ts`)
- ...
- Welche Struktur eine Variable hat, lässt sich mit dem Befehl `str` herausfinden

Beispiel

```
> str(Tabelle)  
> str(A)
```

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart und R Umgebung
- 3 Datenstrukturen
 - Vektoren
 - Datentabellen
 - Matrizen
 - Arrays, Listen, ...
- 4 Programmierung**
 - Funktionen in R schreiben
 - Schleifen und Abfragen
- 5 Plotten in R
- 6 Wahrscheinlichkeitsverteilungen

Funktionen

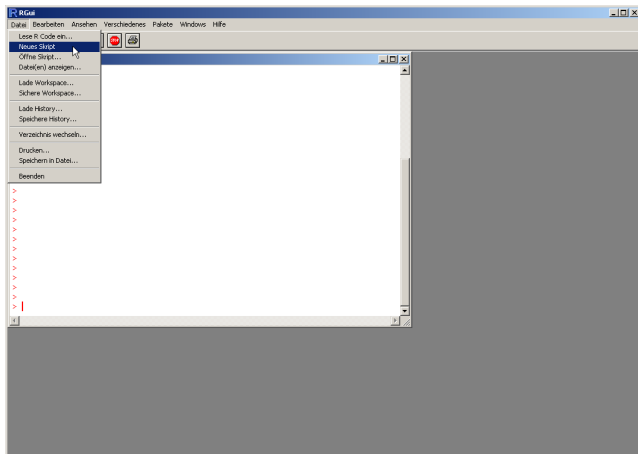
- Eigene Funktionen erstellt man mit dem Befehl
`function(Argumente) {`
Körper der Funktion
`return(Ergebnis)}`
- Optionale Argumente gibt man durch `arg=Wert` an
- Alle im Körper definiertern Variablen sind lokal, d. h. man kann sie außerhalb der Funktion nicht nutzen

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 I

- 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)

allgemeine Programmierhinweise II

- 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
- Rücken Sie den Code so ein, dass man auf einen Blick sieht, wo sich zusammengehörige `{ }` Klammern befinden

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 „Vektor“ 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 Vorfeld 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)  
}
```

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart und R Umgebung
- 3 Datenstrukturen
 - Vektoren
 - Datentabellen
 - Matrizen
 - Arrays, Listen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen
- 5 Plotten in R**
- 6 Wahrscheinlichkeitsverteilungen

Der plot Befehl

- Die Punkte $(x_1, y_1), \dots, (x_n, y_n)$ kann man mittels `plot(x,y)` in ein Koordinatensystem eintragen
- Gibt man nur `plot(y)` ein, so wird $x_i = i$ gesetzt
- Will man die Punkte mit Linien verbinden, so muss man `type = "l"` setzen

Beispiel

```
> plot(1:10,20:11)
> plot(1:10,rep(11:15,2))
> plot(1:10,rep(11:15,2),type="l")
```

Zusätzliche Plot Elemente

- zusätzliche Linien erhält man mit `lines(x,y)`
- zusätzliche Punkte mit `points(x,y)`
- Die Gerade $x \mapsto b \cdot x + a$ kann man mit `abline(a,b)` einfügen
- Eine horizontale Linie erhält man mit `abline(h)`
- Eine vertikale Linie mit `abline(v)`

Beispiel

```
> points(4:7,14:11,col="red")  
> lines(c(4,2,1),c(13,14,12),col="blue")  
> abline(a=11,b=0.5,col="grey")  
> abline(h=12,col="yellow")  
> abline(v=8,col="purple")
```

Funktionsgraphen

- Mit `curve(f,from,to)` wird die Funktion `f` im Intervall `[from,to]` geplottet
- Durch das Argument `add = TRUE` geschieht dies im aktuell bestehenden Plotfenster

Beispiel

Erinnerung: `f<-function(x,y=2) x^y-x`

```
> curve(f,add=TRUE,col="green")  
> curve(f,from=-10,to=10)  
> curve(sin(x))  
> curve(dnorm(x),add=TRUE,col="blue")
```

Legende

- Eine Legende erstellt man mit `legend(x,y=NULL,legend)`
- `(x,y)` gibt die Koordinaten an, wo die Legende erscheinen soll (lässt man `y` als Argument weg, so kann man für `x` relative Positionen wie `"top"` oder `"bottomleft"` nutzen)
- Mit `lty` und `col` lassen sich Linientyp- und Farbe der Legende angeben
- Will man keinen Linientyp sondern ein Punktsymbol angeben, so muss man das Argument `pch` entsprechend setzen

Beispiel

```
> legend("topleft",c("Sinus","N(0,1)
  Dichte"),lty=1,col=c("black","blue"))
> points(-1,0,col=2,pch=0)
> legend("bottomright",c("Der Punkt (-1,0)"),pch=0,col=2)
```

Einige weitere plot-Argumente

- Eine Überschrift erhält man mit `main="text"` (im `plot` Befehl)
- x- und y-Achse lassen sich mit `xlab="text"` bzw. `ylab="text"` beschriften
- Es gibt noch eine Fülle weitere Argumente, etwa `col` (Farbe der Linie oder des Punktes), `lty` (Linientyp), `lwd` (Liniendicke), `pch` (Symbol der Punkte), `bg` (Farbe der Punkte), ...

Beispiel

```
> plot(c(1:10,9:1),type="b",main="Text",  
      xlab="x",ylab="y",pch=22,bg=2,lty=3)
```

Gliederung

- ① Hinweise zum Praktikum
- ② Programmstart und R Umgebung
- ③ Datenstrukturen
 - Vektoren
 - Datentabellen
 - Matrizen
 - Arrays, Listen, ...
- ④ Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen
- ⑤ Plotten in R
- ⑥ Wahrscheinlichkeitsverteilungen

Dichte, Verteilungsfunktion und Quantile

- Mit `dnorm(x,mean=0,sd=1)` erhält man die Dichte der Normalverteilung mit Mittelwert `mean` und Standardabweichung `sd` an der Stelle `x`
- Mit `pnorm(x,mean,sd)` die entsprechende Verteilungsfunktion an der Stelle `x`
- Das α -Quantil dieser Verteilung erhält man durch `qnorm(a,mean,sd)`

Beispiel

```
> dnorm(0,0,sqrt(0.2))  
> pnorm(0,0,sqrt(0.2))  
> qnorm(0.05,-5,sqrt(2))
```

weitere Verteilungen

Viele bekannte Verteilungen sind in R schon implementiert, z. B.

<code>binom(x,n,p)</code>	Binomialverteilung
<code>geom(x,p)</code>	geometrische Verteilung
<code>hyper(x,m,n,k)</code>	Hypergeometrische-Verteilung
<code>nbinom(x,size,prob)</code>	negative Binomialverteilung
<code>pois(x,lambda)</code>	Poisson-Verteilung
<code>unif(x,a,b)</code>	Gleichverteilung auf $[a,b]$
<code>exp(x,rate)</code>	Exponentialverteilung
<code>chisq(x,df)</code>	χ^2 -Verteilung

Beispiel

```
> dexp(10,rate=1/3)
> pbinom(5,size=10,prob=0.5)
> qchisq(0.02,df=10)
```


(Pseudo-)Zufallszahlen und Samples erzeugen

- Setzt man „r“ vor eine Verteilung, so werden n Zufallszahlen generiert
- Bsp: 100 Zufallszahlen der Exponentialverteilung mit Parameter $1/5$ generiert man mit `rexp(n=100,rate=1/5)`
- Will man aus einem Vektor (Matrix, ...) x ein Sample der Größe `size` ziehen, so geschieht das mit `sample(x,size)`
Optionale Argumente: `replace(=FALSE)` gibt an, ob mit zurücklegen gezogen werden soll; mit `prob` kann man die Wahrscheinlichkeiten setzen (Standard = Gleichverteilung auf allen Elementen von x)

Beispiel

```
> rnorm(100)
> sample(1:10,10)
> sample(1:10,10,TRUE)
```