

Funktionsdefinitionen in \mathbb{R} — diskutiert am Beispiel: Funktionsschnittpunkte

Hans Peter Wolf

3.–17. Juni 2004, gedruckt: July 12, 2004

Inhaltsverzeichnis

1 Funktionsaufrufe – wie werden Funktionen eingesetzt?	3
1.1 Funktionskopf – welche Argumente kennt eine Funktion?	3
1.2 Aufrufe – welche Arten von Funktionsaufrufen gibt es?	3
1.2.1 Verkürzte Argumentnamen – wie kann Schreibfaule Zeit gewinnen?	4
1.2.2 Drei Punkte – was gibt’s noch?	4
2 Funktionen – wie werden Funktionen per Hand definiert?	5
2.1 Kommandozeile – wie schreiben wir unsere ersten Funktionen?	5
2.2 Stil – welche fundamentalen Vorgehensregeln gibt es?	6
2.3 Kontrollen – wie können wir Zwischenergebnisse ausgeben?	6
2.4 Kontrollstrukturen – welche Arten der Flusssteuerung gibt es?	7
2.5 Funktionsschnittpunkte – wie könnte ein geeigneter Funktionskopf aussehen?	9
2.6 Funktionseditor – wie aktivieren wir den Funktionseditor?	9
2.6.1 – wie beheben wir Syntaxfehler in der Funktionsdefinition?	9
2.6.2 – wie studieren wir semantische Fehler in der Funktionsdefinition?	10
2.7 Funktionsschnittpunkte – Lösung via Adlerauge?	11
2.8 Input-Check – welche Arten der Überprüfung gibt es?	13
2.9 Funktionsschnittpunkte – wie zoomen wir einen Punkt heran?	15
2.10 Dialoge – wie kann eine Funktion kommunizieren?	16
2.11 Fehlerauffangen – wie gehen wir mit riskanten Anweisungen um?	17
2.12 Funktionsschnittpunkte – wie finden wir mehrere Punkte?	18
2.13 Funktionen – wie schreiben wir Funktionen außerhalb von R?	19
2.13.1 Objekte – wie speichern wir Objekte in einer Datei?	19
2.13.2 Objekte – wie bearbeiten wir Objekte außerhalb von R?	20
2.13.3 Objekte – wie lade wir eine Datei mit Objektdefinitionen?	20
3 WEB – wie programmieren wir im <i>literate programming style</i>?	20
3.1 Beispiel – wie sieht eine literate Lösung für das Schnittproblem aus?	21
3.2 Reflexion – wie erhält man literate Dokumente?	23
4 Funktionen – wie vergrößern wir den Einsatzbereich?	24
4.1 Scoping – wie finden Funktionen ihre Objekte?	24
4.2 Funktionsschnittpunkte – wie verarbeiten wir R-Funktionen?	26
4.3 Funktionsschnittpunkte – wie verarbeiten wir R-Ausdrücke?	28
4.4 Funktionen – wie definieren wir Funktionen per Programm?	30
4.5 Aufrufe – wie werden Funktionsaufrufe bebaut und aktiviert?	30
4.6 Ausdrücke – wie werden Zeichenketten aktiviert?	30
4.7 Funktionen – wie verändern wir Funktionen per Programm?	30
5 OO – wie entwickeln wir objektorientiert?	31
5.1 <i>S version 3</i> – wie entwickeln wir nach <i>S3</i> Lösungen?	31
5.2 Input-Check – wir überprüfen wir Argumente nach <i>S3</i> ?	31
5.3 <i>S version 4</i> – wie programmieren wir auf Basis von <i>S4</i> ?	33
6 Library – wie erstellen wir ein einfaches R-package?	37
6.1 Bibliotheken und Pakete – wie gehen wir damit um?	37
6.2 Objekte – wie erstellen wir Objekte für ein Paket?	38
6.3 Help-Page – wie erstellen wir zu einer Funktion eine Hilfeseite?	38
6.4 Paketverzeichnis – wie richten wir es ein und wie checken wir es?	40
6.5 R-Code – wie bringen wir unsere Objekte ein?	41
6.6 Help – wie verankern wir unsere Hilfeseite?	43
6.7 Paketerstellung – wie erstellen, verschüren und installieren wir unser Paket?	44
6.8 Test – wie testen wir unser Paket?	45
7 Literatur	46

Der Neuling wird in diesem Papier in kleinen Schritten an die Erstellung eigener Funktionen herangeführt und auf einige Besonderheiten am Wegesrand hingewiesen. Um nicht im luftleeren Raum zu verharren, greifen wir zum Studium von Aufrufmöglichkeiten auf die Funktion `sample()` zurück und demonstrieren die Entwicklung von Funktionen am Beispiel des Problems *Schnittpunktbestimmung von zwei Funktionen auf graphischem Weg*. Dabei geht es nicht darum, schnell die eleganteste Lösung zu finden, sondern anhand dieses Beispiels Werkzeuge und Vorgehensweisen kennenzulernen. Wer nur an den Schnittpunktstellen interessiert ist, wird diese mit der R-Funktion `uniroot()` schnellstens herausfinden.

In einer Funktion werden wiederholt benötigte Anweisungsfolgen zusammengefasst, so dass diese nicht immer neu eingetippt werden müssen. Für den Gebrauch ergeben sich zwei Fragen: Was leistet eine Funktion und wie werden mit ihr Daten ausgetauscht? Die Anweisungsfolgen und damit die Wirkungen sind im Körper der Funktion festgelegt. Dort wird auch festgelegt, welches Funktionsergebnis nach Beendigung zurückgeliefert wird. Zur Anpassung an verschiedene Datensituationen können Argumente an Funktionen übergeben werden. Daneben gibt es weitere Möglichkeiten des Informationsaustausches mit der Arbeitsumgebung, jedoch sind solche nur mit Bedacht zu nutzen. Untersuchen wir zunächst mit welcher Vielfalt Funktionen aufgerufen werden können.

1 Funktionsaufrufe – wie werden Funktionen eingesetzt?

Bevor wir uns an die Definition neuer Funktionen wagen, wollen wir ein paar Fakten zum Aufruf von Funktionen sammeln. Denn an dieser Stelle zeigt sich, ob der Designer einer Funktion für den Anwender eine geeignete Spracherweiterung, bestehend aus Funktionsnamen und Argumenten, vorgenommen hat. Welche Argumente eine Funktion besitzt, ist dem Funktionskopf zu entnehmen. Hierzu soll uns die Funktion `sample` Modell stehen, mit der Stichproben aus einer Datenmenge gezogen werden können.

1.1 Funktionskopf – welche Argumente kennt eine Funktion?

Viele Funktionen verwenden wir in R so, wie wir es von den Funktionen der Mathematik her kennen. So berechnet `cos(x)` den Wert der Kosinusfunktion an der Stelle (an den Stellen) x . Viele Funktionen besitzen mehrere Argumente, deren Bedeutung der Hilfe entnommen werden kann. Oft reicht eine Übersicht der Argumente einer Funktion völlig aus, so dass nicht die Dokumentation durchstöbert werden muss. Die Argumente einer Funktion (mit Ausnahme der *primitiven Funktionen*) erhalten wir mit `args()`:

```
in 1: | args(sample)
```

Diese Anweisung zeigt uns die Namen und die Voreinstellung der Argumente von `sample()`:

```
out 1: | function(x, size, replace = FALSE, prob = NULL)
```

Über das Argument x wird die Menge der Elemente festgelegt, aus der eine Stichprobe vom Umfang `size` nach dem Prinzip mit (`replace==TRUE`) oder ohne (`replace==FALSE`) Zurücklegen gezogen werden soll. Über `prob` können die Elemente mit Ziehungswahrscheinlichkeiten gewichtet werden.

1.2 Aufrufe – welche Arten von Funktionsaufrufen gibt es?

Manche Funktionen, wie die Funktion `date()` zur Datumsausgabe, können ganz ohne Parameter aufgerufen werden.

```
in 2: | date()
```

```
out 2: | [1] "Mon Jun 7 10:21:23 2004"
```

`sample()` führt dagegen ohne Parameter nur zu einer Fehlermeldung:

```
in 3: | sample()
```

```
out 3: | Error in sample() : Argument "x" is missing, with no default
```

Es ist also ein `x` anzugeben. `x` ist ein Platzhalter bzw. der interne Name für das erste Argument. Dieser *formale Parameter* erhält erst beim Aufruf einen Inhalt zugeordnet und wird dadurch zum *aktualen Parameter*. Wollen wir aus den ersten 10 Zahlen eine Stichprobe vom Umfang 5 ziehen, so schreiben wir:

```
in 4: | sample(1:10, 5)
```

Die im Aufruf mitgegebenen Elemente werden in ihrer Reihenfolge an die Funktion übergeben. `x` erhält so die Werte `1:10` und `size` den Wert 5. Da die restlichen Argumente nicht angegeben wurden, wird auf die Voreinstellung zurückgegriffen. `replace` hat dann den Wert `FALSE` und `prob` wird `NULL` zugewiesen. Wie man sieht, werden Argumente durch Kommata abgetrennt. Befindet sich beim Funktionsaufruf zum Beispiel zwischen zwei Kommata kein Eintrag oder nur ein Leerzeichen, so wird für das entsprechende Argument der Defaultwert eingesetzt. Eleganter und unabhängiger gegenüber versionsbedingter Veränderungen ist es jedoch, die gewünschten Argumente zu benennen. Bei der Verwendung von Bezeichnungen ist die Reihenfolge nicht relevant. So sind folgende Aufrufe identisch. Dabei ist zu beachten, dass intern `size` – falls nicht angegeben – auf die Anzahl der Elemente von `x` gesetzt wird.

```
in 5: | sample(1:10, , TRUE)
      | sample(1:10, replace=TRUE)
      | sample(replace=TRUE, x=1:10)
      | sample(replace=TRUE, 1:10, size=10)
```

1.2.1 Verkürzte Argumentnamen – wie kann Schreibfaule Zeit gewinnen?

☞ Manchmal ist es lästig, lange Argumentnamen auszuschreiben. Glück gehabt! Denn die Namen der Argumente müssen nur so weit aufgeschrieben werden, bis sie intern eindeutig identifiziert werden können. Vergleiche hierzu auch die Funktion `match.args()`. Demnach kann die Liste wirkungsgleicher `sample`-Beispielaufufe weiter verlängert werden:

```
in 6: | sample(rep=TRUE, x=1:10, si=10, p=NULL)
```

1.2.2 Drei Punkte – was gibt's noch?

☞ Wesentliche Zusammenhänge zum Funktionsaufruf sind damit verkündigt. Unter der Überschrift *objektorientiertes Programmieren* gibt es noch verschiedenes zu sagen, führt jedoch hier zu weit. Dort aber auch anderswo begegnet uns das Argument `"..."`. Dieses spezielle Argument erlaubt uns, eine (fast) beliebige Menge von Datenobjekten zu übergeben. Betrachten wir als unmittelbar einsichtiges Beispiel die Funktion `max()` zur Bestimmung des Maximums einer Menge von Objekten.

```
in 7: | args(max)
```

```
out 7: | function(..., na.rm = FALSE)
```

☞ ☞ Übrigens gibt es neben der Funktion `args()` noch die Funktions `formals()`, mit der sich die Argumente einer Funktion umdefinieren lassen. So könnte man sich schnell eine Funktion zum Ziehen von Stichproben mit Voreinstellung `replace=TRUE` basteln:

```
in 8: | sample.repl<-sample
      | formals(sample.repl)<-alist(x=, size=, replace=TRUE, prob=NULL)
      | sample.repl(1:10, 19)
```

Wir erhalten:

```
out 8: | [1] 8 4 7 6 3 2 8 7 9 2 8 4 2 3 5 8 6 6 10
```

Die Anwendung von `formals()` dürfte Spezialisten vorbehalten bleiben.

2 Funktionen – wie werden Funktionen per Hand definiert?

R unterstützt mehrere Wege zur Entwicklung von Funktionen. Hauptsächlich stehen drei Möglichkeiten zur Funktionsdefinition ins Auge:

1. per Kommandozeile
2. per Editor via `fix()`
3. extern per Editor mit anschließendem Einlesen mittels `source()`

Der Anfänger wird mit der ersten Möglichkeit beginnen, schnell zur zweiten überwechseln und für größere Dinge die dritte zu schätzen lernen. Für ganz besondere Anlässe kann ein Entwickler Funktionen funktionsunterstützt aus Bausteinen zusammensetzen. Hierzu werden wir später Beispiele anfügen.

2.1 Kommandozeile – wie schreiben wir unsere ersten Funktionen?

In der Kommandozeile lässt sich mit wenigen Handgriffen eine einfache Funktion definieren. Als einfaches Beispiel definieren wir die Funktion `root.cubic()`, die die dritte Wurzel aus den Werten des Input-Argumentes `x` zieht:

```
in 9: | root.cubic<-function(x) x^(1/3)
```

Die Anweisung zur Funktionsdefinition besteht also von links beginnend aus einer Zuweisung zur Kreation des Objektes `root.cubic`, dem Keyword `function` gefolgt vom Funktionskopf und dem Funktionskörper mit den gewünschten Anweisungen. Mit der Funktion `args()` können wir sofort überprüfen, ob der Funktionskopf passend eingerichtet worden ist.

```
in 10: | args(root.cubic)
```

```
out 10: | function (x)
```

Der Körper von `root.cubic()` besteht nur aus einer einzigen Anweisung. Mehrere Anweisungen müssen mit geschweiften Klammern zusammengefasst werden. Bei der Funktionsauswertung werden diese wie Kommandozeilen-Eingaben nacheinander ausgeführt. Das zuletzt in einer Funktion erarbeitete Ergebnis wird ausgegeben. Zur Verdeutlichung empfiehlt sich, das Ergebnis mittels `return()` zurückzuliefern. Die Funktion `root.p.1` ("root.p" ergänzt um eine Versionsnummer) zur Ziehung der p -ten Wurzel demonstriert diese Hinweise:

```
in 11: | root.p.1<-function(x,p=1) {  
      |   # Funktion zur Berechnung der p-ten Wurzel  
      |   result <- x^(1/p)  
      |   return(result)  
      | }
```

2.2 Stil – welche fundamentalen Vorgehensregeln gibt es?

Schon bei dieser kleinen Funktion lässt sich erahnen, dass der Weg zu einer korrekten und geeigneten Funktion Disziplin bzw. einen fehlervermeidenden Stil erfordert. Fassen wir wichtige Empfehlungen zusammen:

1. kläre den Zweck der zu entwerfenden Funktion
2. wähle einen problemgerechten Funktionsnamen
3. wähle für die zu übergebenden Daten geeignete Datenstrukturen und verständliche Namen
4. wähle eine zweckmäßige Reihenfolge für die Argumente
5. wähle geeignete Defaultsetzungen
6. gebe das Ergebnis über `return()` zurück

Auf dem Weg zum Ziel sind drei Hürden zu nehmen: syntaktische Korrektheit, semantische Richtigkeit und Validität. Syntaktische Fehler werden vom Interpreter gemeldet. Ob mit der neuen Funktion überhaupt ein relevantes Problem gelöst wird, muss der Anwender entscheiden. Für die semantische Korrektheit können extreme und typische Inputsituationen probiert und die Ergebnisse auf Korrektheit überprüft werden.

2.3 Kontrollen – wie können wir Zwischenergebnisse ausgeben?


Für die Überprüfung der Richtigkeit kann es hilfreich sein, während der Programmentwicklung Kontrollausgaben vorzusehen:

- `print`-Anweisungen zur Ausgabe von Zwischenergebnissen einzugeben, ist der einfachste Weg.

```
in 12: | root.p.2<-function(x,p=1) {  
      | # Funktion zur Berechnung der p-ten Wurzel  
      | exponent<-1/p  
      | ↪ print(exponent)  
      | result <- x^exponent  
      | return(result)  
      | }
```

- Bedingte `print`-Anweisungen zur Ausgabe erlauben unter bestimmten Bedingungen (`DEBUG==TRUE`) Ausgaben hervorzurufen.

```
in 13: | root.p.3<-function(x,p=1,DEBUG=F) {  
      | # Funktion zur Berechnung der p-ten Wurzel  
      | exponent<-1/p  
      | ↪ if(DEBUG) print(exponent)  
      | result <- x^exponent  
      | return(result)  
      | }
```

-  Der Einsatz spezieller Debugging-Routinen und logischer Anschalter (`DEBUG`) stellt eine folgerichtige Weiterentwicklung dar. Je nach Problemsituation und Umfang kann es sehr hilfreich sein, sich vorher geeignete Instrumente zur Ausgabe wichtiger Zwischenergebnisse zu überlegen. Diese können dann situationsabhängig Auskünfte über den Abarbeitungsprozess ausgeben.

```

in 14: | DEBUG<-TRUE
        | show.info<-function(message,variable){
        |   if(DEBUG){
        |     if(!missing(message )) cat("\n",message)
        |     if(!missing(variable)){
        |       cat(" Wert von ",substitute(variable)," : ",sep="")
        |       print(variable)
        |     }
        |   }
        | }

```

Eine Anwendung ist in der Funktion `root.p.4()` zu finden.

```

in 15: | root.p.4<-function(x,p=1,DEBUG=F) {
        |   show.info("== root.p.4 startet ==")
        |   # Funktion zur Berechnung der p-ten Wurzel
        |   exponent<-1/p
        |   ~~~ show.info("---- vor Wurzelziehung",exponent)
        |   result <- x^exponent
        |   show.info("== root.p.4: Berechnungen beendet ==")
        |   return(result)
        | }
        | root.p.4(2.3,5)

```

```

out 15: | == root.p.4 startet ==
        | ---- vor Wurzelziehung
        | Wert von exponent:
        | [1] 0.2
        | == root.p.4: Berechnungen beendet ==
        | [1] 1.181260

```

2.4 Kontrollstrukturen – welche Arten der Flusssteuerung gibt es?

In R finden wir die typischen Kontrollstrukturen vor, wie sie uns aus anderen Programmiersprachen bekannt sind. In den bisherigen Beispielen sind uns bereits Sequenzen von Anweisungen, die durch `{` und `}` zu einer Anweisungsliste zusammengefasst werden können, und `if`-Anweisungen begegnet.

`if`.

Die `if`-Kontrollstruktur begegnet uns in zwei Formen: mit und ohne `else`-Teil.

```
if ( Bedingung ) { Anweisungssequenz }
```

oder

```
if ( Bedingung ) {
  Anweisungssequenz
}else{
  Anweisungssequenz
}
```

Für die Übersicht ist es vorteilhaft, solche Anweisungen geordnet über mehrere Zeilen zu verteilen. Dabei ist bei der Eingabe einer `if-then-else`-Konstruktion zu berücksichtigen, dass bei einem Zeilenwechsel nach dem `then`-Teil die Anweisung als komplett angesehen werden kann. Dieses führt zum Beispiel bei der Verwendung außerhalb von Funktionsdefinitionen dazu, dass das folgende `else` nicht verarbeitet werden kann und eine Fehlermeldung ausgegeben wird.

```
> if(T) 1
[1] 1
> else 2
Error: syntax error
```

Als zweiter Hinweis sei bemerkt, dass bei in sich verschachtelten `if`-Anweisungen ein `else` zu dem unmittelbar vorherigen `if` gehört, sofern nicht Klammern eine andere Zuordnung erzwingen. Neu dürfte für viele die Verwendung innerhalb eines Ausdruck, zum Beispiel in einer Zuweisung sein:

```
in 16: | x.0.5 <- (if(x<0) -x else x)^0.5
```

`ifelse`. Die gerade angegebene Lösung funktioniert leider nur für einelementige Vektoren. Oft benötigen wir die Behandlung von mehreren Vektorelementen in Abhängigkeit einer Bedingung. Hier leistet `ifelse()` gute Dienste. Hiermit können wir – auch in dem Wissen, dass die Funktion `abs()` existiert – mit `ifelse()` eine verbesserte Lösung angeben. Zusätzlich werden `NA`-Einträge auf 0 gesetzt:

```
in 17: | x <-ifelse(is.na(x), 0, x)
      | x.0.5<-ifelse(x < 0, (-x)^0.5, x^0.5)
```

Mit `ifelse()` werden die Werte des ersten Arguments als Bedingungen interpretiert, und es wird jeweils in Abhängigkeit der Wahrheitswerte entweder auf einen Wert des zweiten oder auf einen des dritten zurückgegriffen.

`for`. Für eine feste Anzahl von Wiederholungen einer Anweisungssequenz bietet sich eine `for`-Schleife an. Sie hat die allgemeine Gestalt

```
for ( Element in Vektor ) { Anweisungssequenz }
```

Oft ist der Vektor ein Indexvektor. So druckt ...

```
in 18: | for(i in seq(x)){
      |   cat("Element",i,"von x hat Wurzel",sqrt(x[i]),"\n")
      | }
```

... die Wurzeln der Elemente von `x` aus. Einfacher wäre in diesem Fall natürlich:

```
in 19: | cat("Element",seq(x),"von x hat die Wurzel",sqrt(x),"\n")
```

`while`. Hängt die Wiederholungsanzahl von einer Bedingung ab und ist sie vorher nicht klar, kann auf eine `while`-Schleife zurückgegriffen werden. Die allgemeine Syntax lautet:

```
while ( Bedingung ) { Anweisungssequenz }
```

```
in 20: | i<-1
      | while(i <= length(x)){
      |   cat("ungerades Element",i,"von x hat Wurzel",sqrt(x[i]),"\n")
      |   i<-i+2
      | }
```

Wir sehen, dass wir uns ggf. um das Inkrementieren der Laufvariable selbst kümmern müssen, können mit dieser jedoch frei hantieren. Die Überprüfung der Bedingung finden immer vor einem Durchlauf statt.

repeat. Mit der repeat-Schleife wird im Prinzip eine Schleife gestartet, die solange weiterläuft, bis eine vorsätzliche Unterbrechung durch die Anweisung break wirksam wird.

```
while ( Bedingung ) { Anweisungen ; if ( Bedingung ) break ; Anweisungen }
```

Hier ein kleines Beispiel:

```
in 21: | i <- 0
      | repeat{
      |   i <- i+2
      |   if(i > length(x)) break
      |   cat("gerades Element",i,"von x hat Wurzel",sqrt(x[i]),"\n")
      | }
```

break, next. break kann nicht nur in einer repeat, sondern auch in einer for- oder while-Schleife eingesetzt werden. Die Anweisung next bricht den aktuellen Schleifendurchlauf ab und startet den nächsten, sofern noch einer zu erledigen ist.

2.5 Funktionsschnittpunkte – wie könnte ein geeigneter Funktionskopf aussehen?

Nach diesen Bemerkungen wollen wir uns der Frage der Schnittpunktberechnung zuwenden. Der Abschnitt über Funktionsaufrufe hat uns nicht nur gezeigt, wie vielfältig Funktionen aufgerufen werden können, sondern auch die große Bedeutung der Argumente einer Funktion. Vor der Codierung der Wirkungen muss sich der Entwickler ausreichend Zeit für die Auswahl eines passenden Namens und für die Argumente nehmen. Was ist ein passender Name für eine Funktion, die die Schnittpunkte zweier mathematischer Funktionen berechnet? Hierfür eignet sich der Name cut.function. Beginnen wollen wir jedoch zunächst mit einer Vorstufe, einer Funktion, die die Schnittpunkte zweier Normalverteilungsdichten berechnet. Diese soll den Namen cut.dnorm() bekommen. Verschiedene Versionen werden wir durch Anhängen einer Nummer kennzeichnen. Beide Dichten lassen sich über ihrer Parameterpaare (μ, σ) beschreiben; diese Paare sollen mittels der Argumente f1.par und f2.par an cut.dnorm() übermittelt werden. Falls nur die erste Dichte genau spezifiziert wird, soll die Standardnormalverteilung als zweites Modell dienen. Damit können wir eine fast körperlose Funktion entwerfen:

```
in 22: | cut.dnorm.1<-function(f1.par, f2.par=c(0,1),DEBUG=T){
      |   show.info("== cut.dnorm.1 startet ==")
      |   return("under construction")
      | }
```

Jetzt müssen wir nur noch den Funktionskörper definieren. Da wir hierfür eine ganze Reihe Anweisungen benötigen, ist die Eingabe mit Hilfe eines Editors vorzuziehen und so kommen wir zur zweiten Art, Funktionen zu definieren bzw. weiterzuentwickeln.

2.6 Funktionseditor – wie aktivieren wir den Funktionseditor?

fix(). R erleichtert die Bearbeitung von Funktionen mit der Funktion fix(), der als Argument der Name der zu bearbeitenden Funktion zu übergeben ist. fix() startet einen Editors, mit dem die aktuelle Funktionsdefinition angezeigt wird und verändert werden kann. Zur Übung kann der Leser mit der Funktion root.p() eigene Experimente unternehmen:

```
in 23: | fix(root.p)
```

2.6.1 – wie beheben wir Syntaxfehler in der Funktionsdefinition?

Schnell merkt der experimentierfreudige Leser, dass sich syntaktische Fehler einschleichen. Dann scheitert nach dem Verlassen des Editors die Einrichtung der neuen Funktionsversion und es wird eine Fehlermeldung ausgegeben.

```
in 24: | fix(root.p)
```

`edit()`. Im Falle eines Syntaxfehlers erscheint zum Beispiel folgende Fehlermeldung:

```
out 24: | Error in edit(name, file, editor) : An error occurred on line 8
        | use a command like
        | x <- edit()
        | to recover
```

Dieser Meldung ist zu entnehmen, dass mit `edit()` an den syntaktisch fehlerhaften Anweisungen Korrekturen vorgenommen werden können. Damit das Ergebnis von `edit()` nicht verloren geht, muss es zugewiesen werden. Traut man seinen eigenen Verbesserungen nicht, sollte man eine unschädliche Zuweisung wählen wie zum Beispiel:

```
in 25: | root.p.neuer.versuch<-edit()
```

Bei Erfolg kann man ja die alte Funktion durch die verbesserte überschreiben:

```
in 26: | root.p<-root.p.neuer.versuch
```

2.6.2 – wie studieren wir semantische Fehler in der Funktionsdefinition?

`browser()`. Zur Überprüfung semantischer Fragen bietet R mit der Funktion `browser()` dem Anwender ein interaktives Werkzeug zur Inspektion an. An der Stelle, an der `browser()` aufgerufen wird, wird der Bearbeitungsfluss der Funktion unterbrochen und dem Anwender ein Eingabeaufforderungszeichen gezeigt. Hinter diesen können nun alle möglichen Anweisungen eingegeben werden. Besonders sei darauf hingewiesen, dass `ls()` nun die lokalen Objekte der laufenden Funktion zeigt. Eine Eingabe von `c` beendet die Inspektion. Wird ein `n` eingegeben, blendet `browser()` das nächste Kommando ein bzw. führt es durch. Auf diese Weise können wir schrittweise eine Funktion testen. Werden während des `browser`-Dialogs Objekte verändert, wird bei der Fortführung der Funktionsauswertung mit den veränderten Inhalten weitergearbeitet.

```
in 27: | root.p.5<-function(x,p=1) {
        |   # Funktion zur Berechnung der p-ten Wurzel
        |   exponent<-1/p
        |   ~~~
        |   browser()
        |   result <- x^exponent
        |   return(result)
        | }
```

Hier ein kleiner Mitschnitt:

```
out 27: | > root.p.5(2.3,5)
        | Called from: root.p.5(2.3, 5)
        | Browse[1]> ls()
        | [1] "exponent" "p"          "x"
        | Browse[1]> exponent
        | [1] 0.2
        | Browse[1]> exponent<-1
        | Browse[1]> exponent
        | [1] 1
        | Browse[1]>
        | [1] 2.3
        | >
```

`debug()` **und** `undebug()`. Eine schrittweise Funktionsausführung kann durch Setzen eines *debugging flag* initiiert werden. Dieses Flag wird für die Funktion `root.p.6()` durch die Anweisung `debug(root.p.6)` gesetzt mit der Wirkung, dass die Funktion bei einem Aufruf Schritt für Schritt ausgeführt wird. Zwischen den einzelnen Schritten erscheint ein Prompt-Zeichen zur Anweisungseingabe. `n` führt einen Schritt voran, `c` beendet die Inspektion und andere R-Anweisungen werden wie immer ausgewertet. Das `debuggin` flag wird mittels `undebug(root.p.6)` wieder zurückgesetzt.

```
in 28: | root.p.6<-function(x,p=1,DEBUG=F) {
      |   # Funktion zur Berechnung der p-ten Wurzel
      |   exponent<-1/p
      |   result <- x^exponent
      |   return(result)
      | }
      | debug(root.p.6)
      | root.p.6(2.3,5)
```

Hier ein Beispiel-Protokoll:

```
out 28: | debugging in: root.p.6(2.3, 5)
      | debug: {
      |   exponent <- 1/p
      |   result <- x^exponent
      |   return(result)
      | }
      | Browse[1]> n
      | debug: exponent <- 1/p
      | Browse[1]> exponent
      | Error: Object "exponent" not found
      | Browse[1]> n
      | debug: result <- x^exponent
      | Browse[1]> exponent
      | [1] 0.2
      | Browse[1]>
      | debug: return(result)
      | Browse[1]> ls()
      | [1] "DEBUG"      "exponent" "p"          "result"    "x"
      | Browse[1]>
      | exiting from: root.p.6(2.3, 5)
      | [1] 1.181260
```

Fassen wir zusammen:

- `fix()` ruft Editor zur Funktionsdefinition auf
- `xyz<-edit()` erlaubt Syntaxfehler zu beheben und die Funktion unter `xyz` abzulegen
- `browser()` gestattet die Funktionsunterbrechung zur Objektinspektion
- `debug()` / `undebug` ermöglicht die schrittweise Funktionsabarbeitung

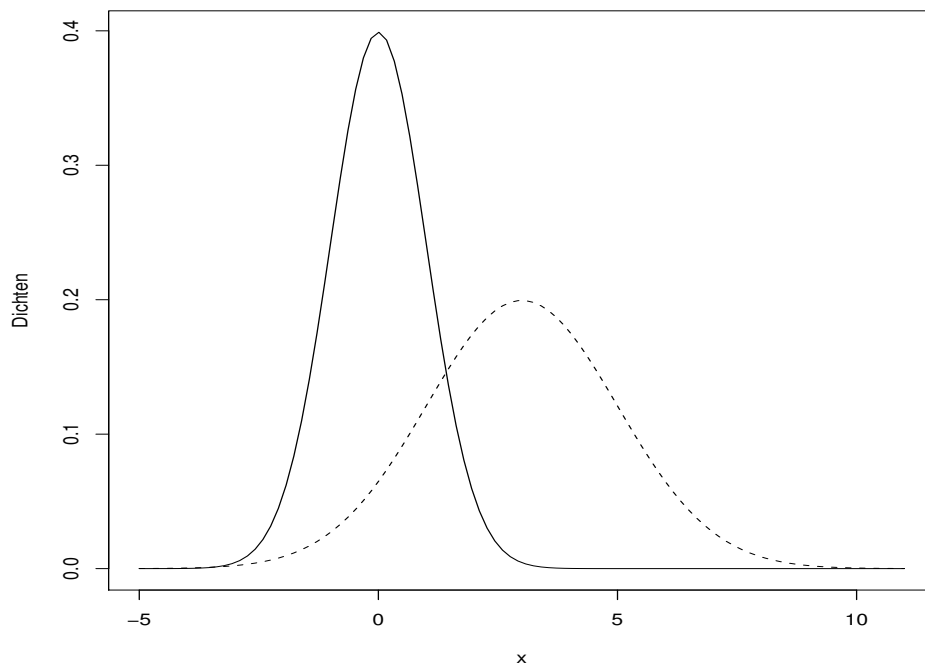
2.7 Funktionsschnittpunkte – Lösung via Adlauge?

Wir wollen mit diesem Wissen die Entwicklung von `cut.dnorm()` vorantreiben. Wir streben in diesem Papier eine graphische Lösung an. Die Anweisungen der Funktion gliedern wir in einen Teil zur Input-Überprüfung, einen Berechnungs- und einen Darstellungsteil. Im zentralen Handlungsteil stellen wir die Dichten graphisch dar und lösen damit das Schnittpunktproblem nach der Methode *Adlauge*.

Funktionswerteberechnungen und Plot. Aufgrund der eingegebenen Parameter legen wir den relevanten Bereich nach dem Motto $E(X) \pm 4 \cdot \sqrt{\text{Var}(X)}$ fest. Wir generieren in diesem Bereich 100 x -Werte und berechnen für diese die Dichtewerte. Zum Schluss stellen wir die Punkte durch Polygonzüge dar.

```
in 29: cut.dnorm.2<-function(f1.par,f2.par=c(0,1)){
  show.info("== cut.dnorm.2 startet ==")
  # Inputcheck
  # ... noch zu erledigen
  # Berechnungen
  x1.min<-f1.par[1]-4*f1.par[2];   x1.max<-f1.par[1]+4*f1.par[2]
  x2.min<-f2.par[1]-4*f2.par[2];   x2.max<-f2.par[1]+4*f2.par[2]
  x<-seq(min(x1.min,x2.min), max(x1.max,x2.max),length=100)
  y1<-dnorm(x,f1.par[1],f1.par[2])
  y2<-dnorm(x,f2.par[1],f2.par[2])
  # Plot
  plot(x,y1,type="l",ylim=c(0,max(y1,y2)), ylab="Dichten")
  lines(x,y2,lty=2)
  return("under construction")
}
cut.dnorm.2(0:1,3:2)
```

Auf diese Weise erhalten wir folgenden schönen Plot.



2.8 Input-Check – welche Arten der Überprüfung gibt es?

Bedingte Fehlermeldungen. Damit wir bei der Abarbeitung keinen Schiffsbruch erleiden, überprüfen wir den Input. Die beiden Argumente müssen hinreichend die Parameter der Verteilungen spezifizieren. Was alles schief laufen kann, ist im Einzelfall schwer zu beantworten und oft nur sehr mühsam zu implementieren. Hier ein erster Vorschlag:

```
in 30: cut.dnorm.3<-function(f1.par,f2.par=c(0,1)){
  show.info("== cut.dnorm.3 startet ==")
  # Inputcheck
  if(missing(f1.par)) return("ERROR: Parameter Funktion 1 fehlen")
  if(length(f1.par)<1) return("ERROR: Parameter Funktion 1 fehlen")
  if(any(is.na(f1.par))) return("ERROR: NA in Parameter Funktion 1")
  if(!is.numeric(f1.par))return("ERROR: Parameter F 1 nicht numerisch ")
  if(length(f1.par)<2) f1.par<-c(f1.par,1)
  if(f1.par[2]<=0) return("ERROR: SD Funktion 1 nicht positiv")
  if(length(f2.par)<1) return("ERROR: Parameter Funktion 2 fehlen")
  if(any(is.na(f2.par))) return("ERROR: NA in Parameter Funktion 2")
  if(!is.numeric(f2.par))return("ERROR: Parameter F 2 nicht numerisch ")
  if(length(f2.par)<2) f2.par<-c(f2.par,1)
  if(f2.par[2]<=0) return("ERROR: SD Funktion 2 nicht positiv")
  # Berechnungen
  x1.min<-f1.par[1]-4*f1.par[2]; x1.max<-f1.par[1]+4*f1.par[2]
  x2.min<-f2.par[1]-4*f2.par[2]; x2.max<-f2.par[1]+4*f2.par[2]
  x<-seq(min(x1.min,x2.min), max(x1.max,x2.max),length=100)
  y1<-dnorm(x,f1.par[1],f1.par[2]); y2<-dnorm(x,f2.par[1],f2.par[2])
  # Plot
  plot(x,y1,type="l",ylim=c(0,max(y1,y2)),ylab="Dichten")
  lines(x,y2,lty=2)
  return("under construction")
}
cut.dnorm.3(0:1,3:2)
```

`missing()`. Mit Hilfe der Funktion `missing()` lässt sich ermitteln, ob ein Funktionsargument mit einem Wert versehen ist. Ist ein Defaultwert gesetzt worden, liefert `missing()` natürlich `FALSE` zurück.

Lokale Funktionen zum Input-Check. Akzeptabel ist die grobe Gliederung der Funktion in einen Überprüfungs-, einen Berechnungs- und einen Ausgabeteil. Unschön ist, dass obwohl für die beiden Parameter fast identische Überprüfungen ablaufen und der Code damit doppelt vorhanden ist. Eine Verbesserung besteht darin, eine lokale Funktion zum Inputcheck zu schreiben.

```

in 31: cut.dnorm.4<-function(f1.par, f2.par=c(0,1)) {
  show.info("== cut.dnorm.4 startet ==")
  # Inputcheck
  ~~~ check.par<-function(f.par, msg="") {
    er<-"ERROR:"
    if(length(f.par)<2) return(paste(er, "Parameter fehlen", msg))
    if(any(is.na(f.par))) return(paste(er, "NA in Parametervektor", msg))
    if(!is.numeric(f.par)) return(paste(er, "Parameter nicht numerisch", msg))
    if(f.par[2]<=0) return(paste(er, "sigma nicht positiv", msg))
    return("ok")
  }
  ~~~ if("ok"!=(msg<-check.par(f1.par, "\n-> Dichte 1"))) return(msg)
  ~~~ if("ok"!=(msg<-check.par(f2.par, "\n-> Dichte 2"))) return(msg)
  # Berechnungen
  x1.min<-f1.par[1]-4*f1.par[2]; x1.max<-f1.par[1]+4*f1.par[2]
  x2.min<-f2.par[1]-4*f2.par[2]; x2.max<-f2.par[1]+4*f2.par[2]
  x<-seq(min(x1.min, x2.min), max(x1.max, x2.max), length=100)
  y1<-dnorm(x, f1.par[1], f1.par[2]); y2<-dnorm(x, f2.par[1], f2.par[2])
  # Plot
  plot(x, y1, type="l", ylim=c(0, max(y1, y2)), ylab="Dichten")
  lines(x, y2, lty=2)
  return("under construction")
}
cut.dnorm.4(0:1, 3:2)

```

Wir sehen, dass auf diese Weise Anweisungen gespart werden. Jedoch werden nicht mehr die Unterschiede bei der Defaultsetzungen berücksichtigt, die Prüfung mit "missing" ist auf der Strecke geblieben.

Globale Checkfunktionen. Da die Checkfunktion auch an anderen Stellen gute Dienste leisten könnte, wollen wir sie als globale Funktion definieren. Für jede Verteilung können wir eine entsprechende Parameter-Checkfunktion entwerfen und erhalten so ein ganzes Bündel von Routinen.

```

in 32: check.par.dnorm<-function(f.par, msg="") {
  er<-"ERROR:"
  if(length(f.par)<1) return(paste(er, "Parameter fehlen", msg))
  if(any(is.na(f.par))) return(paste(er, "NA in Parametervektor", msg))
  if(!is.numeric(f.par)) return(paste(er, "Parameter nicht numerisch", msg))
  if(f.par[2]<=0) return(paste(er, "sigma nicht positiv", msg))
  return("ok")
}

```

Mit dieser Checkfunktion vereinfacht sich unsere Schnittpunktfunktion zu:

```

in 33: cut.dnorm.5<-function(f1.par, f2.par=c(0,1)) {
  show.info("== cut.dnorm.5 startet ==")
  # Inputcheck
  if("ok"!=(msg<-check.par.dnorm(f1.par, "\n-> Dichte 1"))) return(msg)
  if("ok"!=(msg<-check.par.dnorm(f2.par, "\n-> Dichte 2"))) return(msg)
  # Berechnungen
  x1.min<-f1.par[1]-4*f1.par[2]; x1.max<-f1.par[1]+4*f1.par[2]
  x2.min<-f2.par[1]-4*f2.par[2]; x2.max<-f2.par[1]+4*f2.par[2]
  x<-seq(min(x1.min, x2.min), max(x1.max, x2.max), length=100)
  y1<-dnorm(x, f1.par[1], f1.par[2]); y2<-dnorm(x, f2.par[1], f2.par[2])
  # Plot
  plot(x, y1, type="l", ylim=c(0, max(y1, y2)), ylab="Dichten")
  lines(x, y2, lty=2)
  return("under construction")
}
cut.dnorm.5(0:1, 3:2)

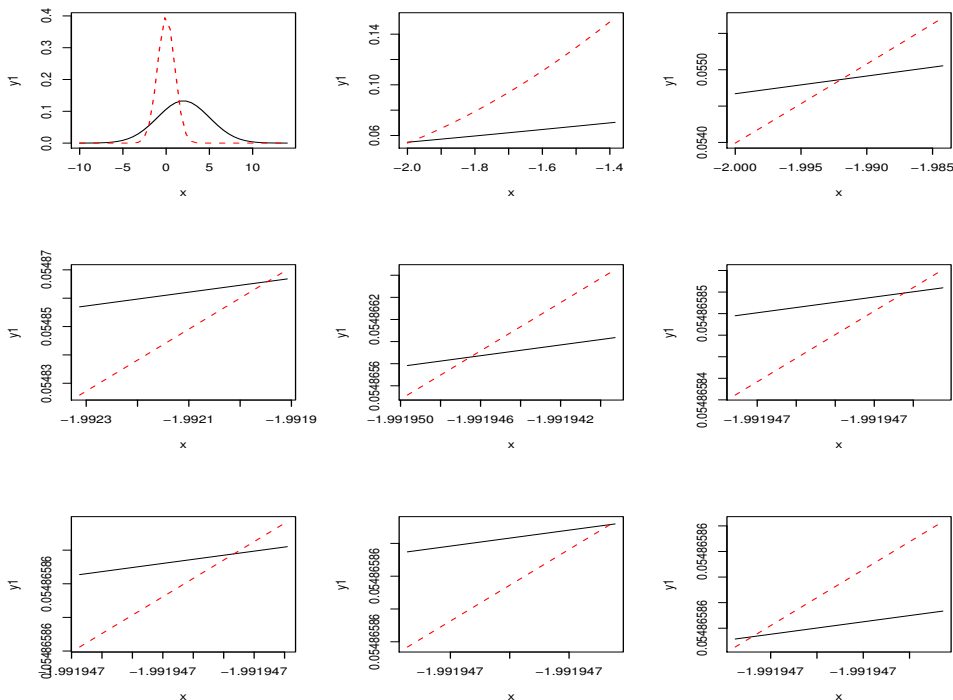
```

2.9 Funktionsschnittpunkte – wie zoomen wir einen Punkt heran?

Wir wollen unseren Vorschlag weiter verbessern, indem wir nach dem Prinzip der Intervallschachtelung eine Schnittpunktstelle der Dichten schrittweise eingrenzen. Dazu suchen wir die beiden x -Werte, zwischen denen sich ein Schnittpunkt befinden muss, und wenden das oben entwickelte Vorgehen erneut an. Weiter werden die ersten Intervallgrenzen etwas eleganter berechnet und auf `x.extr` abgelegt.

```
in 34: cut.dnorm.6<-function(f1.par,f2.par=c(0,1),i.max=3,x.n=40){
  show.info("== cut.dnorm.6 startet ==")
  # Inputcheck
  if("ok"!=(msg<-check.par.dnorm(f1.par,"\n-> Dichte 1")) return(msg)
  if("ok"!=(msg<-check.par.dnorm(f2.par,"\n-> Dichte 2")) return(msg)
  # Berechnungen initialer Extremwerte
  x.extr<-range( cbind(1,c(4,-4))%% cbind(f1.par,f2.par) )
  # Aufteilung Graphics Device
  par(mfrow=rep(ceiling(i.max^0.5),2))
  # Zooming-Schleife: Funktionswerte, Plot, neue Intervallgrenzen
  for(i in 1:i.max){
    x<-seq(x.extr[1], x.extr[2], length=x.n)
    y1<-dnorm(x,f1.par[1],f1.par[2]); y2<-dnorm(x,f2.par[1],f2.par[2])
    plot(x,y1,type="l",ylim=c(min(y1,y2),max(y1,y2)))
    lines(x,y2,lty=2,col="red")
    index<-which(diff(sign(y1-y2))!=0)[1]; x.extr<-x[index+0:1]
  }
  par(mfrow=c(1,1))
  return("under construction")
}
cut.dnorm.6(2:3,i.max=9)
```

Für den Beispielaufruf erhalten wir folgenden Plot:



Man beachte, dass der Schnittpunkt auf 7 Stellen genau bestimmt ist, also genauer als die Stellen der x -Achse bezeichnet sind.

2.10 Dialoge – wie kann eine Funktion kommunizieren?

Da die Genauigkeit vor schlecht festgesetzt werden kann, bietet es sich an, den Anwender zu fragen, ob er weiter zoomen möchte. Wir zeigen an dieser Stelle nur eine ganz spartanische Möglichkeit, mit dem Anwender aus der Funktion heraus zu kommunizieren. Dazu verwenden wir die Funktion `readline()`. Daneben hat R noch Funktionen wie `scan()` und `menu()` im Vorrat. Der Leser möge sich zu deren Einsatz die Hilfeseiten durchlesen. In unserem Beispiel wird durch die Frage, ob weiter gezoomt werden soll, das Argument `i.max` überflüssig. Die `for`-Schleife wandeln wir in eine `while`-Schleife um. Zur Absicherung verpacken wir das Abfragen in eine kleine Funktion ...

```
in 35: ja.nein.frage<-function(text){
      repeat{
        cat("\n",text,"\ngeben Sie j fuer ja oder n fuer nein ein:")
        antwort<-readline()
        antwort<-gsub(" ", "", antwort)
        if(nchar(antwort)==0) next
        antwort<-substring(antwort,1,1)
        if(antwort!="j" && antwort!="n") next
        break
      }
      return(antwort)
    }
```

... und setzen sie in `cut.dnorm.7()` ein:

```
in 36: cut.dnorm.7<-function(f1.par, f2.par=c(0,1), x.n=40){
      show.info("== cut.dnorm.7 startet ==")
      # Inputcheck
      if("ok"!=(msg<-check.par.dnorm(f1.par, "\n-> Dichte 1")) return(msg)
      if("ok"!=(msg<-check.par.dnorm(f2.par, "\n-> Dichte 2")) return(msg)
      # Berechnungen initialer Extremwerte
      x.extr<-range( cbind(1, c(4, -4))%% cbind(f1.par, f2.par) )
      # Zooming-Schleife: Funktionswerte, Plot, neue Intervallgrenzen
      fertig<-FALSE
      while(!fertig){
        x<-seq(x.extr[1], x.extr[2], length=x.n)
        y1<-dnorm(x, f1.par[1], f1.par[2]); y2<-dnorm(x, f2.par[1], f2.par[2])
        plot(x, y1, type="l", ylim=c(min(y1, y2), max(y1, y2)))
        lines(x, y2, lty=2, col="red")
        fertig<-"n"==ja.nein.frage("Weiterzoomen?")
        index<-which(diff(sign(y1-y2))!=0)[1]; x.extr<-x[index+0:1]
      }
      par(mfrow=c(1,1))
      return("under construction")
    }
cut.dnorm.7(2:3)
```


2.11 Fehlerauffangen – wie gehen wir mit riskanten Anweisungen um?

Auch wenn alle Syntaxfehler behoben sind, kann es zu Laufzeitfehlern kommen. Zum Beispiel kann in der letzten Funktion die Suche nach dem Intervall mit dem Schnittpunkt ins Leere gehen, wenn in dem Bereich kein Schnittpunkt auffindbar ist (probiere `cut.dnorm.6(0:1)`). Dann wird dem Objekt `x.extr` ein leerer Vektor zugewiesen und im nächsten Schleifendurchlauf erzeugt `seq()` eine Fehlermeldung. Zur Verhinderung können wir an der Quelle ansetzen und bei `index`-Defekten korrigierend eingreifen.

`try()`. Wir können aber auch auf die Funktion `try()` zurückgreifen. Wird dieser Funktion eine Anweisungsliste ohne Fehler übergeben, dann wird diese – wie gewohnt – abgearbeitet. Tritt jedoch ein Fehler auf, dann kommt es nicht zu einem Absturz, sondern `try()` gibt ein Objekt der Klasse `try-error` mit der Fehlermeldung als Inhalt aus. So können wir mit `try()` gefährliche Stellen absichern:

```
in 37: | cut.dnorm.8<-function(f1.par,f2.par=c(0,1),i.max=3,x.n=40){
      |   show.info("== cut.dnorm.8 startet ==")
      |   # Inputcheck
      |   if("ok"!=(msg<-check.par.dnorm(f1.par,"\n-> Dichte 1")) return(msg)
      |   if("ok"!=(msg<-check.par.dnorm(f2.par,"\n-> Dichte 2")) return(msg)
      |   # Berechnungen initialer Extremwerte
      |   x.extr<-range( cbind(1,c(4,-4))%% cbind(f1.par,f2.par) )
      |   # Aufteilung Graphics Device
      |   par(mfrow=rep(ceiling(i.max^0.5),2))
      |   # Schleife ueber Intervalle: Funktionswerte, Plot, neue Intervallgrenzen
      |   ~~~ res<-try(for(i in 1:i.max){
      |       |     x<-seq(x.extr[1], x.extr[2], length=x.n)
      |       |     y1<-dnorm(x,f1.par[1],f1.par[2]); y2<-dnorm(x,f2.par[1],f2.par[2])
      |       |     plot(x,y1,type="l",ylim=c(min(y1,y2),max(y1,y2)))
      |       |     lines(x,y2,lty=2,col="red")
      |       |     index<-which(diff(sign(y1-y2))!=0)[1]; x.extr<-x[index+0:1]
      |       |   })
      |   ~~~ par(mfrow=c(1,1))
      |   ~~~ if(class(res)=="try-error") cat("Fehler: kein Schnittpunkt gefunden!")
      |   ~~~ return("under construction")
      | }
      | }
```

```
38 <error 38>≡
    cut.dnorm.8(0:1)
```

Für Anfänger sei die bedingte Anweisung zur Ausgabe der Fehlermeldung noch einmal betrachtet. Denn oft ist man versucht, die Bedingung in Form von

```
if(res==NULL) ...
```

auszudrücken. Da `NULL` ein Vektor der Länge 0 ist, ist `res==NULL` auch von der Länge 0 und das `if` reagiert mit einer Fehlermeldung. Deshalb lautet die richtige Abfrage:

```
if(is.null(res) ...
```

Wahrscheinlich muss auch die Anweisung `which(diff(sign(y1-y2))!=0)[1]` kommentiert werden. In dieser wird die erste Position der x -Werte bestimmt, nach der sich die beiden Dichten schneiden. Dazu ermitteln wir zuerst die Vorzeichen der Differenzen an den Stellen x durch `vorz<-sign(y1-y2)`. Die interessante Stelle erkennen wir von links kommend daran, dass der Vorzeichenvektor sich ändert, dass also dessen Differenzvektor von 0 verschieden ist: `diff(vorz)!=0`. Die Funktion `which()` liefert uns die Positionen, an denen ein logischer Vektor den Eintrag `TRUE` besitzt, hier: die Indizes für die linken Intervallgrenzen. Die Analyse der Berechnung von `x.extr` wird dem Leser als Übung überlassen.

2.12 Funktionsschnittpunkte – wie finden wir mehrere Punkte?

Wir sehen, wie schön sich die erste gefundene Stelle immer mehr eingrenzen lässt. Sicher wird der Anwender eher einen Lösungsvorschlag vorziehen, bei dem alle Schnittpunkte ins Visier genommen werden. Dieser folgt mit der nächsten Implementierung.

Die Anzahl der in jeder Iteration zu untersuchenden x -Werte wird über das neue Argument $x.n$ und die Anzahl der Iterationen über $i.max$ festgelegt. Erforderlich wird eine innere Schleife, in der für jeden Schnittpunkt die Schachtelung umgesetzt wird. Hierzu halten wir die Intervallgrenzen auf der zweizeiligen Matrix $x.extr$. In deren ersten Zeile stehen die linken und in der zweiten die rechten Intervallgrenzen.

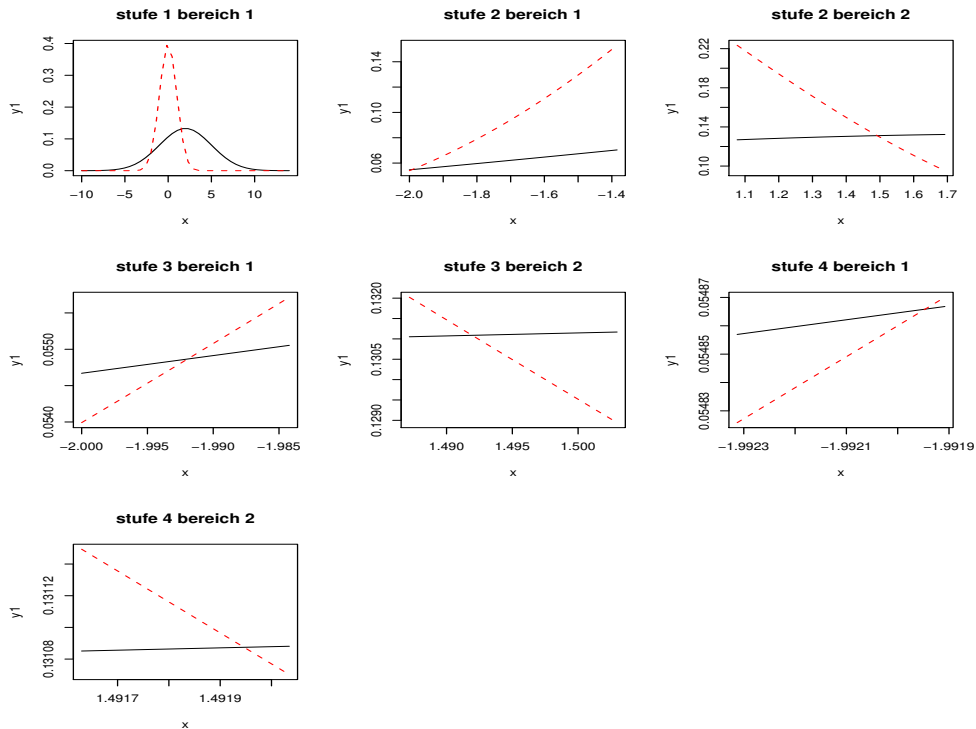
Damit in der inneren Schleife der Aufruf von `dnorm` einfacher wird, schreiben wir zwei lokale Funktionen `f1` und `f2`. Positiv ist hierzu herauszuheben, dass auf diese Weise nur der erste Teil der Funktion von dem Spezialfall *Normalverteilungsdichten* abhängt. Damit ist auch der Grundstein zur Verallgemeinerung gelegt.

```
in 39: cut.dnorm.9<-function(f1.par, f2.par=c(0,1), i.max=4, x.n=40) {
  show.info("== cut.dnorm.9 startet ==")
  # Inputcheck
  if(length(f1.par)<2) f1.par<-c(f1.par,1)
  if(length(f2.par)<2) f2.par<-c(f2.par,1)
  ~~~ if("ok"!=(msg<-check.par.dnorm(f1.par, "\n-> Dichte 1")) return(msg)
  ~~~ if("ok"!=(msg<-check.par.dnorm(f2.par, "\n-> Dichte 2")) return(msg)
  # Funktionen fixieren
  f1<-function(x) dnorm(x, f1.par[1], f1.par[2])
  f2<-function(x) dnorm(x, f2.par[1], f2.par[2])
  # Anfangsbereich festlegen
  x.extr.new<-cbind(range( cbind(1, c(4, -4))%*% cbind(f1.par, f2.par) ))
  # Iterationen durchfuehren
  par(mfrow=c(3,3))
  res<-try(for(i in 1:i.max){
    x.extr<-x.extr.new; x.extr.new<-NULL
    ~~~ for(j in 1:ncol(x.extr)){
      x<-seq(x.extr[1, j], x.extr[2, j], length=x.n); y1<-f1(x); y2<-f2(x)
      plot(x, y1, type="l", ylim=c(min(y1, y2), max(y1, y2)),
           main=paste("stufe", i, "bereich", j))
      lines(x, y2, lty=2, col="red")
      index<-which(diff(sign(y1-y2)) !=0)
      if(0<length(index))
        ~~~ x.extr.new<-cbind(x.extr.new, rbind(x[index], x[index+1]))
    }
  })
  if(class(res)=="try-error") cat("Fehler: kein Schnittpunkt gefunden!")
  par(mfrow=c(1,1))
  ("under construction")
}
```

```
40 <* 40>≡
  cut.dnorm.9(2:3)
```

Zum Einsatz der Funktion `cut.dnorm.9()` ist kritisch zu bemerken, dass sie nur die Schnittpunkte findet, die durch die Rasterung erkennbar sind. Deshalb ist die Zahl der Iterationen und die Anzahl der x -Werte mit Überlegung zu wählen.

Hier sind die zu dem Beispielaufruf erzeugten Graphiken:



2.13 Funktionen – wie schreiben wir Funktionen außerhalb von R?

Wenn die Programmieraufgaben größer werden, kann sich die Entwicklungsarbeit mit `fix()` als zu beengend erweisen. Zum Beispiel können wir mit `fix()` immer nur eine einzige Funktion gleichzeitig bearbeiten. Deshalb ist der Übergang zu folgendem Dreisprung naheliegend:

- speichere (ggf. einmalig) den Code zu einer oder mehrerer Funktionen in lesbarer Form in einer Datei
- bearbeite die Datei mit dem Lieblings-Editor
- lade die verbesserte Version in die R-Umgebung zurück

Da Speicherungs- und Ladeprozesse immer auch mit einem Ort verbunden sind, muss der Anwender Dateinamen ggf. mit einem Pfad ausstatten. Er kann aber auch mit `setwd()` den Arbeitsort wechseln. Der aktuelle Ort lässt sich mit `getwd()` anzeigen. Übrigens müssen unter WINDOWS Rückstriche im Pfad verdoppelt werden. Praktischer ist es jedoch einfache Schrägstriche zu verwenden:

```
in 41: | setwd("C:/R/work")
```

2.13.1 Objekte – wie speichern wir Objekte in einer Datei?

Ob Daten, ob Funktionen – immer wieder ergibt sich der Wunsch, dass man Objekte zwecks Verarbeitung in einer Datei speichern möchte. Eine Möglichkeit besteht darin, per *cut-and-paste*-Technik Objekte zu transferieren. Tastaturorientiert erledigt diese Aufgabe die Funktion `dump()`, der wir eine Liste von Objekten und einen Dateinamen mitgeben können. Über das `append`-Argument können wir bestehende Dateien verlängern. `envir` erlaubt die Umgebung mit den Objekten zu spezifizieren, was an dieser Stelle nicht weiter erklärt wird.

```
in 42: | args(dump)
```

```
out 42: | function (list, file = "dumpdata.R", append = FALSE,  
           |         envir = parent.frame())
```

Die entstehende Datei (voreingestellt ist `dumpdata.R`) enthält nur ASCII-Zeichen und kann nach der Speicherung editiert werden.

2.13.2 Objekte – wie bearbeiten wir Objekte außerhalb von R?

Hier lautet die Aufforderung: öffne die Datei mit der Funktion bzw. den Objekten mit einem Editor und speichere die Veränderungen.

2.13.3 Objekte – wie lade wir eine Datei mit Objektdefinitionen?

Die Funktion `source()` öffnet eine Datei, überprüft die abgelegten Anweisungen auf Syntaxfehler und versucht die Ausdrücke auszuführen. Auf diesem Weg können Datenobjekte definiert und Funktionen in die R-Umgebungen eingebracht werden. Einer Funktionsdefinition können natürlich Anweisungen zum Test der Funktion folgen. Für weitere Feinheiten sei auf die Hilfe verwiesen und nur als Beispiel ein einfacher Aufruf gegeben.

```
in 43: | source("dumpdata.R")
```

Damit bietet sich für die Bearbeitung kleiner Programmieraufgaben an: Starte Editor, bearbeite dort die Funktionen und lade die Zwischenversionen zum Test mit `source()` in die Umgebung.

3 WEB – wie programmieren wir im *literate programming style*?

Mittlerweile dürfte es dem Leser störend aufgestoßen sein, dass sich die Lösungen zu unserem Problem nur in wenigen Punkten unterscheiden. Trotzdem wird immer der gesamte Code angeboten. Der Leser wird besser nicht durch schon bekannte, aber für den R-Interpreter notwendige Verzierungen abgelenkt, sondern sollte sich auf die lokalen Diskussionspunkte konzentrieren können, die ihm schrittweise vorgeführt werden. Es ist also eine geeignete Form von Modularisierung gesucht, die zudem Papier einzusparen vermag.

Neben didaktischen Überlegungen ist ein durchschlagendes Argument für eine sequenzielle Fokussierung auf unterschiedliche Teilprobleme die große Komplexität vieler Probleme bzw. Lösungsansätze. Es wäre doch viel besser, wenn wir eine Lösung Schritt für Schritt durchdenken und entwickeln könnten und am Schluss wie von selbst ein fertiges Programm in den Händen halten. Haben Sie sich schon einmal gefragt, wie Experten vorgehen? Nach meiner Erfahrung wird jeder (Experte) eine Art *Teile-und-herrsche-Stil* praktizieren, der sich vornehmlich an seinen Gedanken ausrichtet.

Für uns ist also ein Stil ratsam, mit dem wir *gedankenorientiert* und *häppchenweise* unsere Funktionen entwickeln können. Der *literate programming style* bietet uns für dieses Ansinnen einen einfachen und erfolgversprechenden Vorschlag an.

Was kennzeichnet den literaten Lösungs-Stil? Beginnen wir mit einigen Stichpunkten zur Charakterisierung des literaten Stils.

- Expliziere alle Deine Gedanken auf dem Weg zur Lösung für *Menschen* verständlich!
- Überlege zuerst → schreibe dann auf → setze zum Schluss um!
- Bilde überschaubare Einheiten aus erklärenden, natürlich sprachlichen Bemerkungen (*text chunks*) und den zugehörigen Programmschritten (*code chunks*)!
- Zerlege Probleme in kleinere!

- Verwende zur Behandlung großer Probleme Module, auch wenn sie bislang noch nicht definiert sind (top-down-Entwurf)!
 - Definiere elementare Module, die für das große Problem hilfreich sein werden (bottom-up-Entwurf)!
- Verarbeite den Quelltext (z.B.: `beispiel.rev`) auf zwei Wegen. Der TANGLE-Prozess setzt das Programm zusammen (z.B.: `beispiel.sch`). Der WEAVE-Prozess generiert eine Output-Datei (z.B.: `beispiel.tex`), die ein Textformatierer (z.B.: L^AT_EX) zu einer schönen Dokumentation verarbeitet.

3.1 Beispiel – wie sieht eine literate Lösung für das Schnittproblem aus?

Zur Demonstration schreiben wir die Funktion `cut.dnorm.9` noch einmal und erhalten so die 10. Version.

Problemstellung. Es soll eine Funktion entworfen werden, die die Schnittpunkte von zwei Normalverteilungsdichten auf graphischem Weg findet. Dazu sollen an äquidistanten Stellen der x -Achse die Funktionen ausgewertet und gezeichnet werden. Die hierdurch erkennbaren Schnittpunkte sollen durch Zooming genauer unter die Lupe genommen werden. Die Dichten sollen über ihre Parameter spezifiziert werden, die als Parameter der Funktion zu übergeben sind. Wird die zweite Dichte nicht über Parameter bestimmt, wird die Normalverteilungsdichte als Default verwendet. Weiterhin sind für die Berechnung die Anzahl der Stützstellen und die Anzahl der Zooming-Schritte anzugeben. Der relevante Bereich, in dem Schnittpunkte gesucht werden, wird automatisch bestimmt.

Syntax und Kopf. Aus den Anforderungen ergeben sich folgende Argumente

Argument	Default	Beschreibung
<code>f1.par</code>		Parameter zur ersten Dichte-Funktion
<code>f2.par</code>	0:1	ggf. Parameter zur zweiten R-Funktion
<code>i.max</code>	4	Zoomingstufen
<code>x.n</code>	40	Anzahl der Stützstellen

Damit ergibt sich der Kopf:

```
44 <start 44>≡
    cut.dnorm.10<-function(f1.par, f2.par=0:1, i.max=4, x.n=40) {
      show.info("== cut.dnorm.10 startet ==")
      <Rumpf von cut.dnorm.10 45>
    }
```

Rumpf. Der Rumpf ist klar gegliedert, wie folgender Grobstruktur entnommen werden kann.

```
45 <Rumpf von cut.dnorm.10 45>≡
    <checke die Inputargumente von cut.dnorm.10 46>
    <definiere die Funktionen f1 und f2 für cut.dnorm.10 47>
    <initialisiere Intervallgrenzen für cut.dnorm.10 48>
    <iteriere über Zoomingstufen, cut.dnorm.10 49>
    <setze Graphik-Parameter zurück, melde Fehler, beende die Funktion cut.dnorm.10 54>
```

Die grobe Struktur besteht aus mehreren Handlungsaufträgen, die später definiert werden. Die Aufträge werden über ihre Namen identifiziert. Man beachte, dass Aufträge, die in anderen Versionen modifiziert werden, sich in ihren Namen unterscheiden müssen. Über die automatisch generierten Referenznummern (hinter den Namen) lassen sich die Definitionen zu den Namen finden.

Inputcheck. Der Input-Check umfasst bislang nur die `f1.par` und `f2.par` und sollte unbedingt auf die anderen Argumente erweitert werden. Zur Anwendung kommt die auf Seite ?? definierte Funktion `check.par.dnorm()`.

```
46 <checke die Inputargumente von cut.dnorm.10 46>≡
  # Inputcheck
  if (length(f1.par)<2) f1.par<-c(f1.par,1)
  if (length(f2.par)<2) f2.par<-c(f2.par,1)
  if ("ok"!=(msg<-check.par.dnorm(f1.par,"\\n-> Dichte 1")) return(msg)
  if ("ok"!=(msg<-check.par.dnorm(f2.par,"\\n-> Dichte 2")) return(msg)
```

f1 und f2. Für das leichtere Hantieren werden die zu betrachtenden Funktionen als lokale Funktionen `f1` und `f2` eingerichtet.

```
47 <definiere die Funktionen f1 und f2 für cut.dnorm.10 47>≡
  # Funktionen fixieren
  f1<-function(x) dnorm(x,f1.par[1],f1.par[2])
  f2<-function(x) dnorm(x,f2.par[1],f2.par[2])
```

Extrema. Zur die Ermittlung der Extrema mit Hilfe eines inneren Produktes passt hier gut eine längere Abhandlung rein, diese wird dem Leser überlassen.

```
48 <initialisiere Intervallgrenzen für cut.dnorm.10 48>≡
  # Anfangsbereich festlegen
  x.extr.new<-cbind(range( cbind(1,c(4,-4))%%c cbind(f1.par,f2.par) ))
```

Zooming. Der Kern aus Berechnung und Darstellung ist zur Sicherheit in eine `try`-Konstruktion eingebettet. Für jeden Zooming-Lauf `i` werden die aktuellen Grenzen auf `extr` abgelegt. Für alle Fälle werden schon mal neun Plots einkalkuliert.

```
49 <iteriere über Zoomingstufen, cut.dnorm.10 49>≡
  # Iterationen durchfuehren
  par(mfrow=c(3,3))
  res<-try(for(i in 1:i.max){
    x.extr<-x.extr.new; x.extr.new<-NULL
    <erstelle Zeichnungen für jedes Intervall und bestimme neue Intervalle 50>
  })
```

Wir entwerfen eine Schleife über die verschiedenen Intervalle, in denen jeweils ein Schnittpunkt liegt.

```
50 <erstelle Zeichnungen für jedes Intervall und bestimme neue Intervalle 50>≡
  for(j in 1:ncol(x.extr)){
    <berechne x- und y-Werte 51>
    <erstelle Zeichnung 53>
    <ermittle neue Intervallgrenzen 52>
  }
```

Zu jedem Intervall berechnen wir einen Vektor aus äquidistanten `x`-Werten und zugehörigen Funktionswerten.

```
51 <berechne x- und y-Werte 51>≡
  x<-seq(x.extr[1,j],x.extr[2,j],length=x.n); y1<-f1(x); y2<-f2(x)
```

Zu der Verkleinerung des Intervalls wurde bereits auf Seite 17 ausführlich Stellung genommen.

```
52 <ermittle neue Intervallgrenzen 52>≡
  index<-which(diff(sign(y1-y2))!=0)
  if(0<length(index))x.extr.new<-cbind(x.extr.new,rbind(x[index],x[index+1]))
```

Mit den Funktionswerten ist die Darstellung kein besonders schweres Problem mehr.

```
53 <erstelle Zeichnung 53>≡  
    plot(x, y1, type="l", ylim=c(min(y1, y2), max(y1, y2)))  
    lines(x, y2, lty=2, col="red"); title(paste("stufe", i, "bereich", j))
```

Abschluss. Jetzt ist nur noch die Funktion geordnet zu beenden.

```
54 <setze Graphik-Parameter zurück, melde Fehler, beende die Funktion cut.dnorm.10 54>≡  
    if(class(res)=="try-error") cat("Fehler: kein Schnittpunkt gefunden!")  
    par(mfrow=c(1, 1))  
    return("under construction")
```

Test. Ein Test muss sein.

```
in 55: | cut.dnorm.10(2:3)
```

3.2 Reflexion – wie erhält man literate Dokumente?

Wie man sieht, lassen sich auf diese Weise sehr schön Details beschreiben, aber auch gut große Dinge strukturieren. Das Zerlegen selbst ist ein kreativer Akt und bleibt Aufgabe des Entwicklers. Trotzdem können einige Stilempfehlungen angegeben werden:

- Versehe die Dokumentation immer mit Datum und Namen (eventuell auch mit Versionsnummer)!
- Diskutiere zunächst immer das Problem. Es muss klar sein, was eigentlich gelöst werden soll!
- Erkläre die Syntax von zu erstellenden Funktionen!
- Beschreibe alle Seiteneffekte!
- Gebe Literatur- und andere Quellen an!
- Begründe die gewählte Struktur!
- Verwende als Modulnamen für Handlungsaufträge Imperative!
- Erkläre die Bedeutung von Variablen!

Literates Programmieren ist hauptsächlich eine Denk- und Schreibtechnik. Sie wird aber auch durch verschiedene WEB-Systeme unterstützt, so dass man wirklich auf die vorgestellte Art und Weise Lösungen entwickeln kann. Zur Technik der Umsetzung des literaten Stils wird jedoch an dieser Stelle nichts ausgeführt. Suche zum Beispiel im Internet nach `noweb`, denn in diesem Papier wurden Elemente des `noweb`-System verwendet. Eine einfache Oberfläche zur Verarbeitung finden man unter

<http://www.wiwi.uni-bielefeld.de/~wolf/lehre/ss04/liptex/tclnoweb2.zip>

4 Funktionen – wie vergrößern wir den Einsatzbereich?

4.1 Scoping – wie finden Funktionen ihre Objekte?

Betrachten wir im letzten Beispiel zur Schnittpunktberechnung die lokale Funktion `f1()`, dann stellen wir fest, dass in `f1` die Objekte `x` und `f1.par` verwendet werden, im Kopf jedoch nur `x` übergeben wird. Weiterhin wird auch die Funktion `dnorm()` im Anweisungsteil verwendet. Deshalb stellt sich (besonders bei mehrfach verwendeten Objektnamen) die Frage, wo nach Objekten gesucht wird und welches Objekt bei mehrfachen Vorkommnissen gefunden wird. R unterscheidet innerhalb von Funktionen drei Typen von Objekte:

- *Lokale Objekte* entstehen durch Zuweisung während der Funktionsabarbeitung. Sie sind nach Beendigung der Funktion nicht mehr zugreifbar. Bei Namensübereinstimmungen werden lokale Objekte zuerst gefunden.
- *Formale Parameter* erhalten beim Start einer Funktion einen Inhalt zugeordnet. Diese Aktual-Parameter werden bei der Auswertung von Ausdrücken verwendet, wenn keine lokalen Objekte gleichen Namens zu finden sind.
- *Freie Objekte* bilden die dritte Gruppe. Wird zu einem Namen weder ein lokales Objekt noch ein Aktual-Parameter gefunden, wird außerhalb der Funktion nach dem Objekt gesucht. Im Unterschied zu S-PLUS wird bei der Definition einer Funktion ihr die Umgebung zugeordnet, in der sie definiert wird. In dieser Umgebung wird dann nach dem gesuchten Objekt gefahndet. `f1` wurde innerhalb von `cut.dnorm.10` definiert, so wird beim Aufruf von `f1` in der aktuellen Umgebung von `cut.dnorm.10` nach `f1.par` gesucht. `dnorm` ist aber auch nicht in `cut.dnorm.10` zu finden. Deshalb geht die Suche weiter, nämlich in der Umgebung in der `cut.dnorm.10` definiert wurde. Wieder Fehlanzeige, so dass zum Schluss an den Orten gesucht wird, die auf dem Suchpfad vermerkt sind – (vergleiche: `search()`).

Betrachten wir zur Demonstration ein Beispiel aus *An Introduction to R*, bei dem eine Funktion `cube` mit der lokalen Funktion `sq` definiert wird. Innerhalb von `sq` wird auf das aus Sicht von `sq` freie Objekt `n` zugegriffen.

```
in 56: | cube<-function(n) {  
      |   sq<-function() n*n  
      |   n*sq()  
      | }
```

In R wird entsprechend der Regel des *lexical scoping* der Parameter `n` von `cube()` gefunden.

```
out 56: | R> cube(2)  
      | [1] 8
```

S-PLUS hat eine andere Suchvorschrift, nach der `n` – cum grano salis – in der globalen Umgebung gesucht wird.

```
out 56: | S> cube(2)  
      | Error in sq(): Object "n" not found  
      | Dumped  
      | S> n <- 3  
      | S> cube(2)  
      | [1] 18
```

Weiter wollen wir uns in einer Funktion mit lokalen Funktionen die Umgebung und die Werte von `x` in verschiedenen Situationen ausgeben lassen.

Mit dem Code ...

```
in 57: | cat ("aussen:")
      | x<-1
      | cat ("x<-1, x:", x)
      | print (environment ())
      | f<-function () {
      |   cat ("in f:")
      |   print (environment ())
      |   cat ("x", x)
      |   x<-2
      |   cat ("x", x)
      |   ff<-function () {
      |     cat ("in ff:")
      |     print (environment ())
      |     cat ("x", x)
      |     x<-3
      |     cat ("x", x)
      |   }
      |   ff ()
      |   fff<-function (x=4) {
      |     cat ("in fff:")
      |     print (environment ())
      |     cat ("x", x)
      |     x<-5
      |     cat ("x", x)
      |   }
      |   fff ()
      | }
      | f ()
```

... erhalten wir von R ausgegeben:

```
out 57: | aussen:
      | x<-1, x: 1
      | <environment: 0x8ef9570>
      | in f:
      | <environment: 0x95c5fe8>
      | x 1
      | x 2
      | in ff:
      | <environment: 0x963e330>
      | x 2
      | x 3
      | in fff:
      | <environment: 0x956fb3c>
      | x 4
      | x 5
```

Für unser Schnittproblem erweist sich der Mechanismus von R als zweckmäßiger. Jedoch sei darauf hingewiesen, dass über freie Objekte der Datenaustausch schnell außer Kontrolle geraten kann. Man sollte deshalb nur in wohlüberlegten Ausnahmefällen Informationen nicht über Funktionsargumente übermitteln. Entsprechendes gilt für Funktionsergebnisse.

4.2 Funktionsschnittpunkte – wie verarbeiten wir R-Funktionen?

Problemstellung Wie können wir unser Schnittproblem für verschiedene Typen von mathematischen Funktionen lösen? Mit der nächsten Version wollen wir Schnittpunkte von Funktionen, die als R-Funktionen vorhanden sind, finden können. Dazu müssen wir den Namen der Funktionen als Argument übermitteln. Hierdurch erhalten wir schon eine recht allgemein einsetzbare Schnittpunktsuchfunktionen. Nennen wir die erste Version `cut.function.1()`. Da für beliebige Funktionen die spannenden Bereiche nicht automatisch bestimmt werden können, muss der Anwender zusätzlich über die Argumente `xmin` und `xmax` den relevanten Bereich vorgeben. Dann können wir fast so verfahren wie bei den Funktionen `cut.dnorm.*`.

Syntax und Kopf. Aus den Anforderungen ergeben sich folgende Argumente

Argument	Default	Beschreibung
<code>f1.name</code>		erste Funktion: R-Funktionsname
<code>f2.name</code>		zweite Funktion: R-Funktionsname
<code>f1.par</code>		ggf. Parameter zur ersten R-Funktion
<code>f2.par</code>		ggf. Parameter zur zweiten R-Funktion
<code>xmin</code>		Untergrenze des Untersuchungsbereichs
<code>xmax</code>		Obergrenze des Untersuchungsbereichs
<code>i.max</code>	4	Zoomingstufen
<code>x.n</code>	40	Anzahl der Stützstellen

Damit folgt der Kopf und Rumpf:

```
58 <start 44>+≡
  cut.function.1<-
  function(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max=4, x.n=40) {
    show.info("== cut.function.1 startet ==")
    <checke die Inputargumente im Falle dnorm 59>
    <lege R-Funktionen als f1 und f2 ab 60>
    <ermittle Schnittpunkte graphisch 61>
  }
```

Input-Check. Die überlegte Checkprozedur für Parameter kann natürlich nur im Falle der Normalverteilung eingesetzt werden.

```
59 <checke die Inputargumente im Falle dnorm 59>≡
  # Inputcheck
  if (f1.name=="dnorm")
    if ("ok"!=(msg<-check.par.dnorm(f1.par, "\n-> Dichte 1")) return(msg)
  if (f2.name=="dnorm")
    if ("ok"!=(msg<-check.par.dnorm(f2.par, "\n-> Dichte 2")) return(msg)
```

`f1` und `f2`. Die hier verwendete Konstruktion wird auf Seite 30 erklärt.

```
60 <lege R-Funktionen als f1 und f2 ab 60>≡
  # Funktionen fixieren
  f1<-function(x) do.call(f1.name, c(list(x), as.list(f1.par)))
  f2<-function(x) do.call(f2.name, c(list(x), as.list(f2.par)))
```

Da im Folgenden der Kern ohne weitere Veränderungen wiederholt verwendet wird, wird hierfür ein eigenständiges Modul entworfen.

```
61 <ermittle Schnittpunkte graphisch 61>≡
  <initialisiere Intervallgrenzen 62>
  <iteriere über Zoomingstufen 63>
  <setze Graphik-Parameter zurück, berichte Fehler und beende die Funktion 65>
```

Extrema. Die angegebenen Intervallgrenzen werden auf der Variablen `x.extr.new` abgelegt.

```
62 <initialisiere Intervallgrenzen 62>≡  
    x.extr.new<-rbind(xmin, xmax)
```

Zooming. Der Kern aus Berechnung und Darstellung ist wieder in eine `try`-Konstruktion eingebettet. Für jeden Zooming-Lauf `i` werden die aktuellen Grenzen auf `extr` abgelegt.

```
63 <iteriere über Zoomingstufen 63>≡  
    res<-try(for(i in 1:i.max){  
        x.extr<-x.extr.new; x.extr.new<-NULL  
        for(j in 1:ncol(x.extr)){  
            <berechne x- und y-Werte 51>  
            <ermittle neue Intervallgrenzen 52>  
            <setze Graphik-Parameter für Bildaufteilung und erzeuge ggf. Leerbilder 64>  
            <erstelle Zeichnung 53>  
        }  
    })
```

Werden im ersten Durchlauf beispielsweise 3 Schnittpunkte entlarvt, werden zwei leere Plots erstellt, damit im Folgenden die Verfeinerungsstufen für einen Schnittpunkt wie in einer Tabelle unter einander zu finden sind.

```
64 <setze Graphik-Parameter für Bildaufteilung und erzeuge ggf. Leerbilder 64>≡  
    if(i==1){  
        par(mfrow=c(i.max, length(index)))  
        for(k in seq(index)[-1]) frame()  
    }
```

Falls ein Fehler auftritt, soll eine Meldung ausgegeben und die Graphik im ganzen noch einmal erstellt werden.

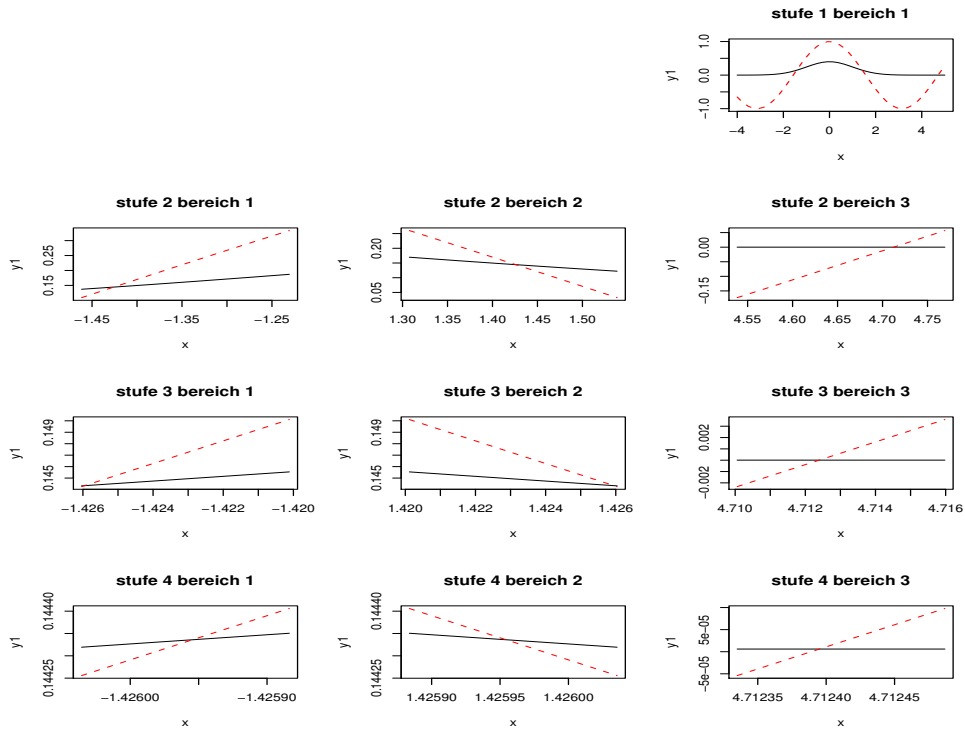
```
65 <setze Graphik-Parameter zurück, berichte Fehler und beende die Funktion 65>≡  
    par(mfrow=c(1, 1))  
    if(class(res)=="try-error"){  
        cat("Fehler: kein Schnittpunkt gefunden!")  
        <erstelle Zeichnung 53>  
    }  
    return("almost completed")
```

Zum Test folgt zwei Beispiel-Aufrufe ...

```
in 66: | cut.function.1(f1.name="pnorm", f2.name="cos", f1.par=2:3,  
                   | f2.par=NULL, xmin=-4, xmax=5, i.max=4, x.n=40)
```

```
in 67: | cut.function.1(f1.name="dnorm", f2.name="cos", f1.par=1:0,  
                   | f2.par=NULL, xmin=-4, xmax=5, i.max=4, x.n=40)
```

... der folgendes Ergebnis hervorbringt:



Dieses Beispiel erzeugt jedoch leider eine Fehlermeldung.

```
in 68: | cut.function.1(f1.name="x^2", f2.name="cos", f1.par=NULL,
                    f2.par=NULL, xmin=-4, xmax=5, i.max=4, x.n=40)
```

4.3 Funktionsschnittpunkte – wie verarbeiten wir R-Ausdrücke?

Wahrscheinlich sind oft Schnittpunkte mit Funktionen gesucht, die nicht als R-Funktionen vorliegen. repräsentiert sind. In solchen Fällen wird der Anwender statt vorher eine Funktion zu definieren lieber einen Ausdruck in x wie $2*x^2-2*x-2$ hinschreiben wollen. Werden die Ausdrücke über die Parameter `f1.name` und `f2.name` übergeben, ist nicht viel zu ändern.

```
69 <start 44>+≡
cut.function.2<-
function(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max=4, x.n=40) {
  show.info("== cut.function.2 startet ==")
  <checke die Inputargumente im Falle dnorm 59>
  <lege R-Funktionen oder Ausdrücke als f1 und f2 ab 70>
  <ermittle Schnittpunkte graphisch 61>
}
```

Auch diese Konstruktion wird erst im nächsten Abschnitt erläutert.

70

```

<lege R-Funktionen oder Ausdrücke als f1 und f2 ab 70>≡
# Funktionen fixieren
if(exists(f1.name))
  f1<-function(x) do.call(f1.name,c(list(x),as.list(f1.par)))
else {
  f1<-eval(parse(text=paste("function(x)", f1.name)))
}
if(exists(f2.name))
  f2<-function(x) do.call(f2.name,c(list(x),as.list(f2.par)))
else {
  f2<-function(x) x; body(f2)<-parse(text=f2.name)
}

```

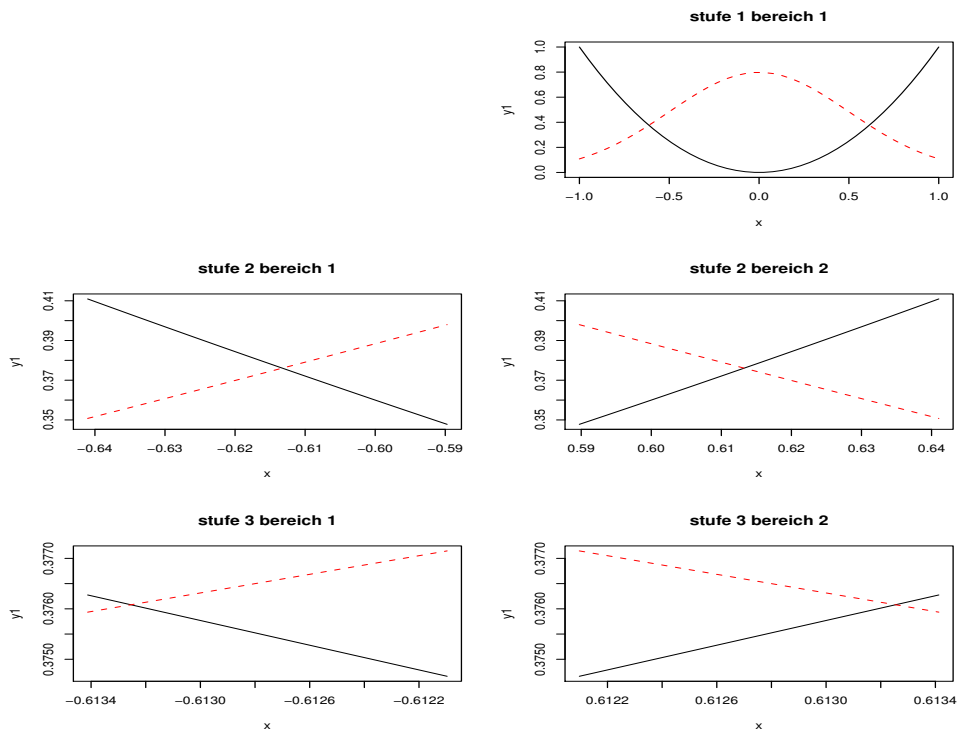
Hier wieder ein paar Aufrufe zum Test ...

```

in 71: | cut.function.2(f1.name="x^2", f2.name="dnorm", f1.par=2:3,
                      f2.par=c(0, 0.5), xmin=-1, xmax=1, i.max=3, x.n=40)

```

... und das graphische Ergebnis:



```

in 72: | cut.function.2(f1.name="sin", f2.name="cos", f1.par=NULL,
                      f2.par=NULL, xmin=-4, xmax=5, i.max=4, x.n=40)

```

Diejenigen, die einen Ausdruck ohne " übergeben wollen, müssen einige weitere Veränderungen durchführen. Diese können von der Funktion curve() abgekupfert werden.

4.4 Funktionen – wie definieren wir Funktionen per Programm?

Die entscheidenden Schritte zur Verallgemeinerung gehen auf die Konstruktionen von `f1` und `f2` zurück. Ohne weitere Bemerkungen wurde weiter oben innerhalb einer Funktion während der Abarbeitung eine neue Funktion:

```
f1<-function(x) dnorm(x, f1.par[1], f1.par[2])
```

definiert und sozusagen per Programm generiert. Dieses zeigt, dass eine Funktionsdefinition eine ganz normale Anweisung ist und wie andere Anweisungen hantiert werden kann. Nach der Definition einer neuen Funktion kann sie wie üblich verwendet werden. Dieses Vorgehen erfordert jedoch schon zur Zeit der Programmierung die exakte Kenntnis der einzelnen Schritte.

4.5 Aufrufe – wie werden Funktionsaufrufe bebaut und aktiviert?

`do.call`. In `cut.function.1()` muss der Name einer R-Funktion übergeben werden. Jedoch wird oft erst während der Funktionsabarbeitung klar, welcher Aufruf in der lokalen Funktion getätigt werden soll. Wir wissen vor der Definition der lokalen Funktion nicht, welche R-Funktion aufgerufen und mit welchen Parametern sie versorgt werden soll. Für diese Frage bietet R mit `do.call()` einen Mechanismus zum Bau und zur Ausführung eines Aufrufs an. `do.call()` wird ein Funktionsname übergeben sowie die Argumente als Liste:

```
f1<-function(x) do.call(f1.name, c(list(x), as.list(f1.par)))
```

4.6 Ausdrücke – wie werden Zeichenketten aktiviert?

Zur Interpretation einer Anweisung müssen zwei Schritte erledigt werden. Zunächst muss die Zeichenkette überprüft und durch (*Parsen*) in eine ausführbare Form übersetzt werden. Erst in einem zweiten Schritt kann eine Evaluation probiert werden. R gestattet es uns, mit der Funktion `parse()`, Zeichenketten zu parsen. Das Ergebnis ist eine sogenannte *expression*. Für die Ausführung können wir den geparseten Ausdruck der Funktion `eval()` übergeben. Auf diese Weise lassen sich beliebige Ausdrücke auf der Ebene von Zeichenketten zusammenbauen, parsen und letztlich auswerten. Im vorliegenden Beispiel wurde so die Funktion `f1()` erstellt.

```
f1<-eval(parse(text=paste("function(x)", f1.name)))
```

Kritisch bleibt anzumerken, dass der Name "`f1.name`" nicht sonderlich gut passt.

4.7 Funktionen – wie verändern wir Funktionen per Programm?

Funktionen lassen sich auch programmtechnisch verändern. Zum Beispiel wird zur Generierung von `f2()` zunächst eine Funktion mit bedeutungslosem Körper definiert und in einem zweiten Schritt der Körper mit der Funktion `body()` ausgetauscht, der eine geparsete Anweisungssequenz zu übergeben ist.

```
f2<-function(x) x; body(f2)<-parse(text=f2.name)
```

Alternativ können wir auch eine Funktion mittels `deparse(fun.name)` in einen Zeichenkettenvektor überführen, diesen durch Zeichenkettenoperationen manipulieren und anschließend wieder durch `eval(parse(. .))` in eine Funktion überführen.

5 OO – wie entwickeln wir objektorientiert?

5.1 S version 3 – wie entwickeln wir nach S3 Lösungen?

Mit dem Vorschlag S3 wird uns im bescheidenem Umfang objektorientiertes Programmieren ermöglicht. Der Grundansatz besteht darin, dass Objekte, die sich Klassen zuordnen lassen, im Mittelpunkt stehen. Dadurch verändert sich der Standpunkt: Statt *nehme Funktion plot und füttere sie mit den Daten xyz* wird formuliert: *Hier ist das Objekt xyz, ich will es geplottet haben!* Wie genau der Plot im Einzelfall zu erstellen ist, wird das Objekt schon wissen bzw. kann der zugehörigen Klasse entnommen werden; es wird die klassenspezifische plot-Methode aktiviert. Um diese Idee umzusetzen, wurden mit dem S3-Vorschlag *generische Funktionen*, wie `plot()`, `print()`, `mean()` und `summary()` eingeführt. Im einfachsten Fall bestehen sie nur aus einer Anweisung:

```
in 73: | summary
```

```
out 73: | function (object, ...)
        | UseMethod("summary")
```

In der generische Funktion wird `UseMethod()` aufgerufen, eine Funktion, die zu dem Objekt `object` die passende spezielle Methode sucht und startet. In jedem Fall muss eine Default-Methode existieren, die nach S3 am Ende `.default` und vorn wie die generische Funktion heißt. So finden wir Methoden `plot.default()` und `summary.default()`. Ist für eine bestimmte Klasse eine spezielle Behandlung – eine spezielle Methode – erforderlich, muss nur diese zusätzliche Spezialmethode entworfen werden. Alle übrigen Methoden bleiben unangetastet. Der Name einer Spezialmethode setzt sich zusammen aus dem Namen der generischen Funktion und der Klasse, für die die neue Methode entworfen werden soll.

5.2 Input-Check – wir überprüfen wir Argumente nach S3?

Zur Demonstration wollen wir die Funktionen zur Argumentüberprüfung nach dem S3-Standard entwerfen. Dieses macht Sinn, weil es viele verschiedene Dichten in R gibt und ohne Probleme neue entworfen werden können.

Zunächst schreiben wir eine generische Methode:

```
in 74: | check.par<-function(x,msg="") {
        |   UseMethod("check.par")
        | }
```

Als zweites sei eine etwas unbefriedigende Default-Methode formuliert:

```
in 75: | check.par.default<-function(x,msg) {
        |   print(paste("keine Check-Funktion zu Argument",
        |             substitute(x), "gefunden"))
        |   return("ok")
        | }
```

Eine Methode für die Klasse `dnorm` haben wir mit der Funktion `check.par.dnorm` bereits eingeführt. Wir wiederholen ihre Definition.

```
in 76: | check.par.dnorm<-function(f.par,msg="") {
        |   er<-"ERROR:"
        |   if(length(f.par)<1)   return(paste(er,"Parameter fehlen",msg))
        |   if(any(is.na(f.par))) return(paste(er,"NA in Parametervektor",msg))
        |   if(!is.numeric(f.par))return(paste(er,"Parameter nicht numerisch",msg))
        |   if(f.par[2]<=0)       return(paste(er,"sigma nicht positiv",msg))
        |   return("ok")
        | }
```

Entsprechend entwerfen wir Methoden für einige andere Dichten und Verteilungsfunktionen.

```
in 77: | check.par.pnorm<-check.par.dnorm
      | # Exponentialverteilung
      | check.par.dexp<-function(f.par,msg="",...){
      |   er<-"ERROR:"
      |   if(length(f.par)<1) return(paste(er,"Parameter fehlt",msg))
      |   if(any(is.na(f.par))) return(paste(er,"NA in Parametervektor",msg))
      |   if(!is.numeric(f.par))return(paste(er,"Parameter nicht numerisch",msg))
      |   if(f.par<=0) return(paste(er,"lambda nicht positiv",msg))
      |   return("ok")
      | }
      | check.par.pexp<-check.par.dexp
      | # t-Verteilung
      | check.par.dt<-function(f.par,msg=""){
      |   er<-"ERROR:"
      |   if(length(f.par)<1) return(paste(er,"Parameter fehlt",msg))
      |   if(any(is.na(f.par))) return(paste(er,"NA in Parametervektor",msg))
      |   if(!is.numeric(f.par))return(paste(er,"Parameter nicht numerisch",msg))
      |   if(f.par<=0) return(paste(er,"df nicht positiv",msg))
      |   return("ok")
      | }
      | check.par.pt<-check.par.dt
```

Exkurs: `substitute()`. ☞ Es sei noch eine Bemerkung zu der Funktion `substitute()` ergänzt. Diese dient – ohne näher auf Details einzugehen – hauptsächlich dem Zweck die Historie eines Arguments zu ergründen: Manchmal will man in einer Funktion wissen, was genau beim Aufruf übergeben wurde. Studiere dazu:

```
in 78: | myplot<-function(x,y){
      |   x.name<-deparse(substitute(x))
      |   y.name<-deparse(substitute(y))
      |   plot(x,y, xlab=x.name, ylab=y.name, main=y.name)
      | }
      | reiner.zufall<-rnorm(10)
      | myplot(reiner.zufall, reiner.zufall * reiner.zufall)
```

In dem von `myplot()` generierten Plot ist die x -Achse mit `reiner.zufall` und die y -Achse mit `reiner.zufall * reiner.zufall` beschriftet. `substitute()` betrachtet das übergebene Objekt ohne vorherige Auswertung als `expression` und erstellt von diesem einen Parsebaum. Dieses kann beispielsweise untersucht werden an:

```
in 79: | as.character(substitute(2*3))
```

```
out 79: | [1] "*" "2" "3"
```

Die Funktion `deparse()` erstellt aus dem Baum eine Zeichenkette.

```
in 80: | deparse(substitute(2*3))
```

```
out 80: | [1] "2 * 3"
```

Eine zweite Verwendung von `substitute()` besteht darin, in `expressions` bestimmte Objekte gegen Werte auszutauschen.

```
in 81: | substitute(a*b, list(a=10,b=20))
```

liefert:

```
out 81: | 10 * 20
```


Schnittpunktlösung mit oo-Check: `cut.function.3()`. Jetzt müssen wir nur noch die zu testenden Parameter der zugehörigen Klasse zuordnen und können dann nach dem oo-Standpunkt aufrufen: *hier sind die Parameter, bitte führe Check durch!*

```
82 <start 44>+≡
  cut.function.3<-
  function(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max=4, x.n=40) {
    show.info("== cut.function.3 startet ==")
    <checke die Inputargumente nach S3-Konzept 83>
    <lege R-Funktionen oder Ausdrücke als f1 und f2 ab 70>
    <ermittle Schnittpunkte graphisch 61>
  }
```

Check mit S3. Zum Check werden die Parameter gemäß der Funktionsnamen Klassen zugeordnet. Dann wird jeweils die generische Funktion `check.par()` aufgerufen.

```
83 <checke die Inputargumente nach S3-Konzept 83>≡
  # Inputcheck
  if(!is.null(f1.par)) {
    class(f1.par) <- f1.name
    if("ok" != (msg <- check.par(f1.par, "-> Funktion 1"))) return(msg)
  }
  if(!is.null(f2.par)) {
    class(f2.par) <- f2.name
    if("ok" != (msg <- check.par(f2.par, "-> Funktion 2"))) return(msg)
  }
```

Test. Zur Überprüfung: Ohne Probleme erhalten wir den erwarteten Plot beim Aufruf von:

```
in 84: | cut.function.3(f1.name="dnorm", f2.name="cos", f1.par=c(0,+1),
  |         f2.par=NULL, xmin=-4, xmax=5, i.max=4, x.n=40)
```

In folgendem Fall wird jedoch nur eine Fehlermeldung ausgegeben:

```
in 85: | cut.function.3(f1.name="dnorm", f2.name="cos", f1.par=c(0,-1),
  |         f2.par=NULL, xmin=-4, xmax=5, i.max=4, x.n=40)
```

```
out 85: | [1] "ERROR: sigma nicht positiv -> Funktion 1"
```

5.3 S version 4 – wie programmieren wir auf Basis von S4?

Mit dem Standardisierungsvorschlag S4 wird S3 weiterentwickelt, indem einerseits Anforderungen an Klassen explizit definiert werden können, andererseits Methoden mit speziellen Werkzeugen eingerichtet werden müssen. Um einen Vergleich zu ermöglichen, lösen wir das Problem aus dem letzten Absatz noch einmal mit S4. Zur Unterscheidung tauschen wir die Zeichen `check` gegen `teste` aus.

Als Vorübung definieren wir die neue Klasse `par.norm`. Objekte (Parametervektoren) dieser Klasse besitzen zwei numerische Elemente mit den Namen `mean` und `sd`, diese Einträge werden als *slots* bezeichnet.

```
in 86: | setClass("par.norm", representation(mean="numeric", sd="numeric"))
```

Ein neues Objekt dieser Klasse erhalten wir durch:

```
in 87: | parnorm.obj <- new("par.norm", mean=0, sd=1)
```

```
out 87: | An object of class "par.norm"
        | Slot "mean":
        | [1] 0
        | Slot "sd":
        | [1] 1
```

Auf die einzelnen Slots können wir zugreifen mittels des Operators @:

```
in 88: | cat("Mittelwert:", parnorm.obj@mean,
        |      "/ Standardabweichung:", parnorm.obj@sd, "\n")
```

```
out 88: | Mittelwert: 0 / Standardabweichung: 1
```

Nun schreiben wir als Vorübung eine generische Funktion `probe()` und legen sie in der globalen Umgebung ab. Wichtig ist, dass später kein anderes Objekt gleichen Names gefunden wird. Deshalb weisen wir sicherheitshalber `probe` explizit mit `assign` zu.

```
in 89: | xyz<-function(x, msg="") {
        |   print(paste("keine Check-Funktion zu Funktionsklasse gefunden"))
        |   "ok"
        | }
        | assign("probe", xyz, .GlobalEnv)
```

Die Vereinbarung als generische Funktion geschieht in dem Moment, in dem wir die erste spezielle Methode einrichten. Dazu schreiben wir für die Klasse `par.norm` eine spezielle Methode:

```
in 90: | setMethod("probe", "par.norm", function(x, msg="") {
        |   er<-"ERROR: "
        |   if(length(x@mean)<1) return(paste(er, "Mittelwert fehlt", msg))
        |   if(length(x@sd)<1) return(paste(er, "SD fehlt", msg))
        |   if(is.na(x@mean) || is.na(x@sd))
        |     return(paste(er, "NA in Parametervektor", msg))
        |   if(x@sd<=0) return(paste(er, "sigma nicht positiv", msg))
        |   return("ok")
        | })
```

und testen:

```
in 91: | cat("Test 1:")
        | parnorm.obj<-new("par.norm", mean=0, sd=1)
        | print(probe(parnorm.obj))
        | cat("Test 2:")
        | parnorm.obj<-new("par.norm", mean=0, sd=-1)
        | print(probe(parnorm.obj))
        | cat("Test 3:")
        | print(probe(1:5))
```

```
out 91: | Test 1:
        | [1] "ok"
        | Test 2:
        | [1] "ERROR: sigma nicht positiv "
        | Test 3:
        | [1] "keine Check-Funktion zu Funktionsklasse gefunden"
```

Schnittpunktuche mit S4-Check. Damit sind die Vorübungen abgeschlossen und wir kehren zu unserem Schnittpunktproblem zurück. Zur Demonstration entwerfen wir: `cut.function.4()`.

Die Vorarbeit. Wir implementieren Default und Spezialmethode.

```
in 92: | setClass("dnorm", representation(mean="numeric",sd="numeric"))
      | xyz<-function(x,msg="") {
      |   print(paste("keine Check-Funktion zu Funktionsklasse gefunden"))
      |   "ok"
      | }
      | assign("teste",xyz,.GlobalEnv)
      | setMethod("teste","dnorm", function(x,msg="") {
      |   er<-"ERROR:"
      |   if(length(x@mean)<1) return(paste(er,"Mittelwert fehlt",msg))
      |   if(length(x@sd)<1)   return(paste(er,"SD fehlt",msg))
      |   if(is.na(x@mean) || is.na(x@sd))
      |     return(paste(er,"NA in Parametervektor",msg))
      |   if(x@sd<=0)         return(paste(er,"sigma nicht positiv",msg))
      |   return("ok")
      | })
```

cut.function.4(). Für die Definition von cut.function.4() können wir fast alles von cut.function.3 übernehmen. Deshalb brauchen wir auch nur den neuen Abschnitt – dank literatem Programmierstil – gesondert erklären.

```
93 | (start 44)+≡
    | cut.function.4<-
    | function(f1.name,f2.name,f1.par,f2.par,xmin,xmax,i.max=4,x.n=40) {
    |   show.info("== cut.function.4 startet ==")
    |   (checke die Inputargumente mittels S4 94)
    |   (lege R-Funktionen oder Ausdrücke als f1 und f2 ab 70)
    |   (ermittle Schnittpunkte graphisch 61)
    | }
```

Input-Check mit S4.

```
94 | (checke die Inputargumente mittels S4 94)≡
    | if(!is.null(f1.par)) {
    |   p.obj<-new(f1.name,mean=f1.par[1],sd=f1.par[2])
    |   if("ok"!=(msg<-teste(p.obj,"-> Funktion 1"))) return(msg)
    | }
    | if(!is.null(f2.par)) {
    |   p.obj<-new(f2.name,mean=f2.par[1],sd=f2.par[2])
    |   if("ok"!=(msg<-teste(p.obj,"-> Funktion 2"))) return(msg)
    | }
```

Test. Zur Überprüfung: Ohne Probleme erhalten wir den erwarteten Plot beim ersten Aufruf und eine Fehlermeldung beim zweiten:

```
in 95: | cut.function.4(f1.name="dnorm",f2.name="cos",f1.par=c(0,+1),
      |               f2.par=NULL,xmin=-4,xmax=5,i.max=4,x.n=40)
```

```
96 | (error 38)+≡
    | cut.function.4(f1.name="dnorm",f2.name="cos",f1.par=c(0,-1),
    |               f2.par=NULL,xmin=-4,xmax=5,i.max=4,x.n=40)
```

```
out 95: | [1] "almost completed"
      | [1] "ERROR: sigma nicht positiv -> Funktion 1"
```

Eine letzte Verbesserung? Dieses kann nur eine Zwischenlösung sein, da für die Testaufrufe nach Funktionsart unterschieden werden muss. Allenfalls sind wir bereit eine allgemeine Umwandlungsfunktion zu akzeptieren, die die Existenz der entsprechenden Klassen voraussetzt:

Neue Methode und eine Konvertierungsfunktion. Die Konvertierungsfunktion ist ohne Tricks entworfen.

```

in 97: setClass("dexp", representation(rate="numeric"))
      setMethod("teste", "dexp", function(x,msg=""){
        er<-"ERROR:"
        if(length(x@rate)<1) return(paste(er,"rate fehlt",msg))
        if(is.na(x@rate)) return(paste(er,"NA in Parametervektor",msg))
        if(x@rate<=0) return(paste(er,"rate nicht positiv",msg))
        return("ok") })
      from.vec.to.type<-function(vec,type) {
        if(type=="dnorm") return(new("dnorm",mean=vec[1],sd=vec[2]))
        if(type=="pnorm") return(new("dnorm",mean=vec[1],sd=vec[2]))
        if(type=="dexp") return(new("dexp",rate=vec[1]))
        if(type=="pexp") return(new("dexp",rate=vec[1]))
        vec
      }
      from.vec.to.type->>from.vec.to.type

```

Die Definition von cut.function.5. Wir können nun einen neuen Prüfteil einbauen:

```

98 (start 44)+≡
   cut.function.5<-
   function(f1.name,f2.name,f1.par,f2.par,xmin,xmax,i.max=4,x.n=40){
     show.info("== cut.function.5 startet ==")
     (checke die Inputargumente mittels S4 und from.vec.to.type 99)
     (lege R-Funktionen oder Ausdrücke als f1 und f2 ab 70)
     (ermittle Schnittpunkte graphisch 61)
   }

```

Parameter-Check.

```

99 (checke die Inputargumente mittels S4 und from.vec.to.type 99)≡
   if(!is.null(f1.par)) {
     p.obj<-from.vec.to.type(f1.par,f1.name)
     if("ok"!=(msg<-teste(p.obj,"-> Funktion 1")) return(msg)
     print(msg)
   }
   if(!is.null(f2.par)) {
     p.obj<-from.vec.to.type(f2.par,f2.name)
     if("ok"!=(msg<-teste(p.obj,"-> Funktion 2")) return(msg)
     print(msg)
   }

```

Test. ... und der Test:

```

in 100: | cut.function.5(f1.name="dnorm",f2.name="dexp",f1.par=c(2:1),
           |           f2.par=2,xmin=-4,xmax=5,i.max=4,x.n=40)
in 101: | cut.function.5(f1.name="dnorm",f2.name="cos",f1.par=c(0:1),
           |           f2.par=NULL,xmin=-4,xmax=5,i.max=4,x.n=40)
in 102: | cut.function.5(f1.name="dnorm",f2.name="dexp",f1.par=c(1,-2),
           |           f2.par=2,xmin=-4,xmax=5,i.max=4,x.n=40)

```

Fazit. Weitere Verbesserungen bleiben dem Leser überlassen.

6 Library – wie erstellen wir ein einfaches R-package?

Zur Beantwortung dieser Frage verpacken wir eine Version von `cut.function`, die ohne Argument-Prüfung funktioniert, in einem Paket namens `fcut`. Zunächst üben wir den Umgang mit Bibliotheken und Paketen. Dann verschnüren wir eine `cut`-Funktion in einem Paket und installieren es an einer anderen Stelle. Den Abschluss bildet ein kurzer Gebrauchstest.

6.1 Bibliotheken und Pakete – wie gehen wir damit um?

Zuerst frischen wir unser Wissen im Umgang mit Paketen auf. Fertige Funktionen werden in Paketen (packages) zusammengefasst. (Mehrere) Pakete werden in einer Bibliothek (library) abgelegt. Zum Beispiel enthält das Basispaket elementare R-Funktionen wie `ls()`, graphische Routinen sind in dem Paket `graphics` und statistische in `stats` zu finden. Diese Pakete befinden sich in der Bibliothek `lib`. Welche Bibliotheken erreichbar sind, ist der Funktion `.libPaths()` zu entnehmen, mit der auch eine weitere Bibliothek dem Suchpfad der Bibliothek hinzugefügt werden kann.

```
in 103: | .libPaths()
```

```
out 103: | [1] "/usr/lib/R/library"
```

Welche Pakete in einer Bibliothek vorhanden sind, zeigt uns `library()` an.

```
in 104: | library()
```

Das Ergebnis könnte so aussehen:

```
out 104: | Packages in library '/usr/lib/R/library':
          | KernSmooth           Functions for kernel smoothing for Wand &
          |                   Jones (1995)
          | MASS                 Main Package of Venables and Ripley's MASS
          | base                 The R Base Package
          | boot                 Bootstrap R (S-Plus) Functions (Canty)
          | class                Functions for Classification
          | cluster              Functions for clustering (by Rousseeuw et al.)
          | ctest                Defunct Package for Classical Tests
          | eda                  Defunct Package for Exploratory Data Analysis
          | foreign              Read data stored by Minitab, S, SAS, SPSS,
          |                   Stata, ...
          | graphics            The R Graphics Package
          | ...
```

Falls die Bibliothek an anderer Stelle zu finden ist, kann das über das Argument `lib.loc` der Funktion `library()` vermittelt werden:

```
in 105: | library(lib.loc="mylib")
```

Mit `library()` werden auch Pakete geladen. `laden`. So lädt folgender Befehl das Paket `boot`.

```
in 106: | library(boot)
```

Hierdurch wird der Suchpfad, anhand dessen ggf. nach Objekten gesucht wird, verlängert.

```
in 107: | search()
```

```
out 107: | [1] ".GlobalEnv"      "package:boot"      "package:methods"  "package:stats"
          | [5] "package:graphics" "package:utils"     "Autoloads"        "package:base"
```

Die zugehörigen Pfade liefert:

```
in 108: | searchpaths()
```

Die Pfade zu den Paketen erhalten wir durch:

```
in 109: | .path.package()
```

```
out 109: | [1] ".GlobalEnv"                "/usr/lib/R/library/boot"
          | [3] "/usr/lib/R/library/methods" "/usr/lib/R/library/stats"
          | [5] "/usr/lib/R/library/graphics" "/usr/lib/R/library/utils"
```

Eine geladenes Paket wird wieder entfernt über den Namen oder die Position im Suchpfad durch die Anweisung:

```
in 110: | detach("package:boot")
        | search()
```

```
out 110: | [1] ".GlobalEnv"                "package:methods" "package:stats" "package:graphics"
          | [5] "package:utils"           "Autoloads"       "package:base"
```

Von der Position im Suchpfad zur Umgebung führt uns die Funktion:

```
in 111: | library(boot)
        | print(pos.to.env(2))
```

```
out 111: | [1] ".GlobalEnv"                "/usr/lib/R/library/boot"
          | [3] "/usr/lib/R/library/methods" "/usr/lib/R/library/stats"
          | [5] "/usr/lib/R/library/graphics" "/usr/lib/R/library/utils"
          | [7] "Autoloads"
```

6.2 Objekte – wie erstellen wir Objekte für ein Paket?

Wir erstellen Objekte für ein Paket, wie wir üblicherweise Objekte erstellen. Zur Demonstration schreiben wir die neue Funktion `cut.function`.

```
in 112: | cut.function.6<-
        | function(f1.name,f2.name,f1.par,f2.par,xmin,xmax,i.max=4,x.n=40){
        |   print("== cut.function.6 startet ==")
        |   # Inputcheck
        |   # Funktionen fixieren
        |   if(exists(f1.name)) f1<-function(x) do.call(f1.name,c(list(x),as.list(f1.par)))
        |   else f1<-eval(parse(text=paste("function(x)", f1.name)))
        |   if(exists(f2.name)) f2<-function(x) do.call(f2.name,c(list(x),as.list(f2.par)))
        |   else f2<-function(x) x; body(f2)<-parse(text=f2.name)
        |   x.extr.new<-rbind(xmin,xmax)
        |   res<-try(for(i in 1:i.max){
        |     x.extr<-x.extr.new; x.extr.new<-NULL
        |     for(j in 1:ncol(x.extr)){
        |       x<-seq(x.extr[1,j],x.extr[2,j],length=x.n); y1<-f1(x); y2<-f2(x)
        |       index<-which(diff(sign(y1-y2))!=0)
        |       if(0<length(index))x.extr.new<-cbind(x.extr.new,rbind(x[index],x[index+1]))
        |       if(i==1){
        |         par(mfrow=c(i.max,length(index)))
        |         for(k in seq(index)[-1])frame()
        |       }
        |       plot(x,y1,type="l",ylim=c(min(y1,y2),max(y1,y2)))
        |       lines(x,y2,lty=2,col="red"); title(paste("stufe",i,"bereich",j))
        |     }
        |   })
        |   par(mfrow=c(1,1))
        |   if(class(res)=="try-error"){
        |     cat("Fehler: kein Schnittpunkt gefunden!")
        |     plot(x,y1,type="l",ylim=c(min(y1,y2),max(y1,y2)))
        |     lines(x,y2,lty=2,col="red"); title(paste("stufe",i,"bereich",j))
        |   }
        |   return("almost completed")
        | }
```

6.3 Help-Page – wie erstellen wir zu einer Funktion eine Hilfeseite?

Nach der Definition einer Funktion muss an die Hilfe-Seite gedacht werden. Also erstellen wir einen Rohling.

```
in 113: | ## prompt("cut.function.6") ## auskommentiert, damit Rd-file nicht ueberschrieben wird!
```

Es entsteht cut.function.6.Rd:

```
\name{cut.function.6}
\alias{cut.function.6}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{ function to do ... }
\description{
  A concise (1-5 lines) description of what the function does.
}
\usage{
cut.function.6(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max = 4, x.n = 40)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{f1.name}{ Describe \code{f1.name} here }
  \item{f2.name}{ Describe \code{f2.name} here }
  \item{f1.par}{ Describe \code{f1.par} here }
  \item{f2.par}{ Describe \code{f2.par} here }
  \item{xmin}{ Describe \code{xmin} here }
  \item{xmax}{ Describe \code{xmax} here }
  \item{i.max}{ Describe \code{i.max} here }
  \item{x.n}{ Describe \code{x.n} here }
}
\details{
  If necessary, more details than the __description__ above
}
\value{
  Describe the value returned
  If it is a LIST, use
  \item{comp1 }{Description of 'comp1'}
  \item{comp2 }{Description of 'comp2'}
  ...
}
\references{ put references to the literature/web site here }
\author{ who you are }
\note{ further notes }

Make other sections like Warning with \section{Warning }{...}

\seealso{ objects to See Also as \code{\link{fun}}, }
\examples{
##---- Should be DIRECTLY executable !! ----
##-- ==> Define data, use random,
##-- or do help(data=index) for the standard data sets.

## The function is currently defined as
function(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max=4, x.n=40) {
  print("== cut.function.6 startet ==")
  # Inputcheck
  # Funktionen fixieren
  if(exists(f1.name)) f1<-function(x) do.call(f1.name,c(list(x),as.list(f1.par)))
  else f1<-eval(parse(text=paste("function(x) ", f1.name)))
  if(exists(f2.name)) f2<-function(x) do.call(f2.name,c(list(x),as.list(f2.par)))
  else f2<-function(x) x; body(f2)<-parse(text=f2.name)
  x.extr.new<-rbind(xmin,xmax)
  res<-try(for(i in 1:i.max){
    x.extr<-x.extr.new; x.extr.new<-NULL
    for(j in 1:ncol(x.extr)){
      x<-seq(x.extr[1,j],x.extr[2,j],length=x.n); y1<-f1(x); y2<-f2(x)
      index<-which(diff(sign(y1-y2))!=0)
      if(0<length(index))x.extr.new<-cbind(x.extr.new,rbind(x[index],x[index+1]))
      if(i==1){
        par(mfrow=c(i.max,length(index)))
        for(k in seq(index)[-1])frame()
      }
      plot(x,y1,type="l",ylim=c(min(y1,y2),max(y1,y2)))
      lines(x,y2,lty=2,col="red"); title(paste("stufe",i,"bereich",j))
    }
  })
  par(mfrow=c(1,1))
  if(class(res)=="try-error"){
    cat("Fehler: kein Schnittpunkt gefunden!")
    plot(x,y1,type="l",ylim=c(min(y1,y2),max(y1,y2)))
    lines(x,y2,lty=2,col="red"); title(paste("stufe",i,"bereich",j))
  }
  return("almost completed")
}
}
\keyword{ kwd1 }% at least one, from doc/KEYWORDS
\keyword{ kwd2 }% ONLY ONE keyword per line
```

Wir modifizieren geeignet den Inhalt.

```
\name{cut.function.6}
\alias{cut.function.6}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{ function to do ... }
\description{
  Funktion zur graphischen Bestimmung von Funktionsschnittpunkten.
}
\usage{
cut.function.6(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max = 4, x.n = 40)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{f1.name}{ Name oder Ausdruck zur Beschreibung der ersten Funktion }
  \item{f2.name}{ Name oder Ausdruck zur Beschreibung der zweiten Funktion }
  \item{f1.par}{ Parameter zur ersten Funktion }
  \item{f2.par}{ Parameter zur zweiten Funktion }
  \item{xmin}{ Minimum des Untersuchungsbereiches }
  \item{xmax}{ Maximum des Untersuchungsbereiches }
  \item{i.max}{ Anzahl der Zooming-Schritte }
  \item{x.n}{ Anzahl der Stuetzstellen }
}
\details{
  Siehe R-Steilkurs
}
\value{
  Es wird eine Graphik erstellt und der Text "almost completed ausgegeben.
}
\references{ R-Steilkurs }
\author{ H.P.Wolf }
\note{ --- }

\section{Warning }{In dieser Version findet kein Input-Check statt}

\seealso{ \code{\link{uniroot}} }
\examples{
##---- Should be DIRECTLY executable !! ----
##-- ==> Define data, use random,
##-- or do help(data=index) for the standard data sets.
cut.function.6("dnorm", "x^2", 0:1, NULL, -3, 3, 100)

## The function is currently defined as
function(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max=4, x.n=40){
  print("== cut.function.6 startet ==")
  # Inputcheck
  # Funktionen fixieren
  if(exists(f1.name)) f1<-function(x) do.call(f1.name,c(list(x),as.list(f1.par)))
  else f1<-eval(parse(text=paste("function(x)", f1.name)))
  if(exists(f2.name)) f2<-function(x) do.call(f2.name,c(list(x),as.list(f2.par)))
  else f2<-function(x) x; body(f2)<-parse(text=f2.name)
  x.extr.new<-rbind(xmin,xmax)
  res<-try(for(i in 1:i.max){
    x.extr<-x.extr.new; x.extr.new<-NULL
    for(j in 1:ncol(x.extr)){
      x<-seq(x.extr[1,j],x.extr[2,j],length=x.n); y1<-f1(x); y2<-f2(x)
      index<-which(diff(sign(y1-y2))!=0)
      if(0<length(index))x.extr.new<-cbind(x.extr.new,rbind(x[index],x[index+1]))
      if(i==1){
        par(mfrow=c(i.max,length(index)))
        for(k in seq(index)[-1])frame()
      }
      plot(x,y1,type="l",ylim=c(min(y1,y2),max(y1,y2)))
      lines(x,y2,lty=2,col="red"); title(paste("stufe",i,"bereich",j))
    }
  })
  par(mfrow=c(1,1))
  if(class(res)=="try-error"){
    cat("Fehler: kein Schnittpunkt gefunden!")
    plot(x,y1,type="l",ylim=c(min(y1,y2),max(y1,y2)))
    lines(x,y2,lty=2,col="red"); title(paste("stufe",i,"bereich",j))
  }
  return("almost completed")
}
}
\keyword{ hplot }
```

6.4 Paketverzeichnis – wie richten wir es ein und wie checken wir es?

Für den Generierungsprozess erstellen wir ein neues Verzeichnis. Da wir verschiedentlich mit Pfaden hantieren müssen, schreiben wir eine Hilfsfunktion, die an ein Heimatverzeichnis (z.B. `getwd()`) Pfadstücke anhängt.

```
in 114: | mypath<-getwd()
        | bilde.pfad<-function(...){
        |   file.path(mypath,...)
        | }
```

Wir legen unter unserer Arbeitsstelle `mypath` ein Testverzeichnis `test` und in diesem ein Arbeitsverzeichnis mit dem Namen des zu entstehenden Paketes (`fcut`) an.

```
in 115: | dir.create(bilde.pfad("test"))
        | dir.create(bilde.pfad("test", "fcut"))
```


und legen in diesem einen File mit Namen DESCRIPTION an: Beispiel:

```
in 116: | DESC<-c(  
        | "Package: fcut",  
        | "Version: 0.1",  
        | "Date: 2004-07-07",  
        | "Title: Funktionsschnitt",  
        | "Author: pw <pwolf@wiwi.uni-bielefeld.de>",  
        | "Maintainer: Peter Wolf <pwolf@wiwi.uni-bielefeld.de>",  
        | "Depends: R (>= 1.9)",  
        | "Description: Paket enthaelt eine Funktion zum Schnitt von Funktionen",  
        | "License: GPL version 2 or newer",  
        | "URL: http://www.wiwi.uni-bielefelde.de/wolf"  
        | )  
        | cat(DESCR, sep="\n", file=bilde.pfad("test", "fcut", "DESCRIPTION"))
```

Zur Sicherheit wagen wir einen ersten Check.

```
in 117: | setwd(bilde.pfad("test"))  
        | system("R CMD check fcut")
```

Wir beobachten folgende Ausgaben.

```
out 117: | * checking for working latex ... OK  
        | * using log directory '/home/wiwi/pwolf/R/R-kurs/R-steilkurs/test/fcut.Rcheck'  
        | * checking for file 'fcut/DESCRIPTION' ... OK  
        | * checking if this is a source package ... OK  
  
        | * Installing *source* package 'fcut' ...  
        | No man pages found in package 'fcut'  
        | * DONE (fcut)  
  
        | * checking package directory ... OK  
        | * checking for portable file names ... OK  
        | * checking for sufficient/correct file permissions ... OK  
        | * checking DESCRIPTION meta-information ... OK  
        | * checking package dependencies ... OK  
        | * checking index information ... OK  
        | * checking package subdirectories ... OK  
        | * checking DVI version of manual ... OK
```

6.5 R-Code – wie bringen wir unsere Objekte ein?

Jetzt bringen wir den Code ein. Die Aufgabe lautet: Speichere den Code an passender Stelle. Der R-File ist in einem Unterverzeichnis mit Namen R abzulegen.

```
in 118: | dir.create(bilde.pfad("test", "fcut", "R"))  
        | dump("cut.function.6",  
        |       bilde.pfad("test", "fcut", "R", "cut.function.6.R"))
```

Für alle Fälle führen wir erneut einen Check durch.

```
in 119: | setwd(bilde.pfad("test"))  
        | system("R CMD check fcut")
```

Output des Checks:

```
out 119: * checking for working latex ... OK
* using log directory '/home/wiwi/pwolf/R/R-kurs/R-steilkurs/test/fcut.Rcheck'
* checking for file 'fcut/DESCRIPTION' ... OK
* checking if this is a source package ... OK

* Installing *source* package 'fcut' ...
** R
No man pages found in package 'fcut'
* DONE (fcut)

* checking package directory ... OK
* checking for portable file names ... OK
* checking for sufficient/correct file permissions ... OK
* checking DESCRIPTION meta-information ... OK
* checking package dependencies ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking R files for syntax errors ... OK
* checking R files for library.dynam ... OK
* checking S3 generic/method consistency ... WARNING
cut:
  function(x, ...)
cut.function.6:
  function(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max, x.n)

* checking for replacement functions with final arg not named 'value' ... OK
* checking foreign function calls ... OK
* checking DVI version of manual ... OK

WARNING: There was 1 warning, see
  /home/wiwi/pwolf/R/R-kurs/R-steilkurs/test/fcut.Rcheck/00check.log
for details
```

In unserem neuen Paket befinden sich jetzt:

```
in 120: | setwd(bilde.pfad("test", "fcut"))
        | system("ll")
```

```
out 120: | insgesamt 4
        | -rw-r--r--  1 pwolf  wiwi          331 2004-07-01 13:19 DESCRIPTION
        | drwxr-xr-x  2 pwolf  wiwi          80 2004-07-01 13:27 R
```

```
in 121: | setwd(bilde.pfad("test", "fcut", "R"))
        | system("ll")
```

```
out 121: | insgesamt 4
        | -rw-r--r--  1 pwolf  wiwi          1317 2004-07-01 13:30 cut.function.6.R
```

In dem beim Checken entstandenen Verzeichnis fcut.Rcheck sind jetzt vorrätig:

```
in 122: | setwd(bilde.pfad("test", "fcut.Rcheck"))
        | system("ls -R")
```

```
out 122: | .:
        | 00check.log R.css fcut fcut-manual.dvi
        |
        | ./fcut:
        | DESCRIPTION Meta R
        |
        | ./fcut/Meta:
        |
        | ./fcut/R:
        | fcut
```

6.6 Help – wie verankern wir unsere Hilfeseite?

Jetzt ist auch noch die Beschreibung zu ergänzen.

```
in 123: | dir.create(bilde.pfad("test", "fcut", "man"))
        | file.copy(from=bilde.pfad("cut.function.6.Rd"),
        |           to=bilde.pfad("test", "fcut", "man", "cut.function.6.Rd"))
```

Und wieder wird ein Check fällig.

```
in 124: | setwd(bilde.pfad("test"))
        | system("R CMD check fcut")
```

```
out 124: | * checking for working latex ... OK
        | * using log directory '/home/wiwi/pwolf/R/R-kurs/R-steilkurs/test/fcut.Rcheck'
        | * checking for file 'fcut/DESCRIPTION' ... OK
        | * checking if this is a source package ... OK
        |
        | * Installing *source* package 'fcut' ...
        | ** R
        | ** help
        | >>> Building/Updating help pages for package 'fcut'
        |       Formats: text html latex example
        |       cut.function.6          text      html      latex      example
        | * DONE (fcut)
        |
        | * checking package directory ... OK
        | * checking for portable file names ... OK
        | * checking for sufficient/correct file permissions ... OK
        | * checking DESCRIPTION meta-information ... OK
        | * checking package dependencies ... OK
        | * checking index information ... OK
        | * checking package subdirectories ... OK
        | * checking R files for syntax errors ... OK
        | * checking R files for library.dynam ... OK
        | * checking S3 generic/method consistency ... WARNING
        | cut:
        |   function(x, ...)
        | cut.function.6:
        |   function(f1.name, f2.name, f1.par, f2.par, xmin, xmax, i.max, x.n)
        |
        | * checking for replacement functions with final arg not named 'value' ... OK
        | * checking foreign function calls ... OK
        | * checking Rd files ... OK
        | * checking for missing documentation entries ... OK
        | * checking for code/documentation mismatches ... OK
        | * checking Rd \usage sections ... OK
        | * creating fcut-Ex.R ... OK
        | * checking examples ... OK
        | * creating fcut-manual.tex ... OK
        | * checking fcut-manual.tex ... OK
        |
        | WARNING: There was 1 warning, see
        | /home/wiwi/pwolf/R/R-kurs/R-steilkurs/test/fcut.Rcheck/00check.log
        | for details
```

Wir können schauen, ob die neuen help-Formate in dem Checkverzeichnis generiert worden sind.

```
in 125: | setwd(bilde.pfad("test", "fcut.Rcheck"))
        | system("ls -R")
```

```

out 125: | .:
          | 00check.log fcut fcut-Ex.Rout fcut-manual.aux fcut-manual.log
          | R.css fcut-Ex.R fcut-Examples.ps fcut-manual.dvi fcut-manual.tex
          |
          | ./fcut:
          | CONTENTS DESCRIPTION INDEX Meta R R-ex help html latex man
          |
          | ./fcut/Meta:
          | Rd.rds hsearch.rds
          |
          | ./fcut/R:
          | fcut
          |
          | ./fcut/R-ex:
          | cut.function.6.R
          |
          | ./fcut/help:
          | AnIndex cut.function.6
          |
          | ./fcut/html:
          | 00Index.html cut.function.6.html
          |
          | ./fcut/latex:
          | cut.function.6.tex
          |
          | ./fcut/man:
          | fcut.Rd

```

scheint alles im grünen Bereich zu sein.

6.7 Paketerstellung – wie erstellen, verschnüren und installieren wir unser Paket?

Wunderbar. Was fehlt noch? Richtig: Einpacken und Auspacken. Beginnen wir damit, das Paket zu erstellen und zu verschnüren.

```

in 126: | setwd(bilde.pfad("test"))
          | system("R CMD build fcut")

```

Die ausgegebenen Meldungen sind durchweg positiv.

```

out 126: | * checking for file 'fcut/DESCRIPTION' ... OK
          | * preparing 'fcut':
          | * removing junk files
          | * building 'fcut_0.1.tar.gz'

```

Wir haben also einen TAR-File mit allen Zutaten generiert.

Im zweiten Schritt müssen wir das Paket dort auspacken, wo es hin soll. Als experimenteller Zielort stehen wir ein Verzeichnis `mylib` zur Verfügung.

```

in 127: | setwd(bilde.pfad("test"))
          | dir.create(bilde.pfad("test", "mylib"))
          | system(paste("R CMD INSTALL --library=",
          |         bilde.pfad("test", "mylib"), " ",
          |         "fcut_0.1.tar.gz", sep=""))

```

Als Rückmeldungen erhalten wir:

```

out 127: | * Installing *source* package 'fcut' ...
          | ** R
          | ** help
          | >>> Building/Updating help pages for package 'fcut'
          |       Formats: text html latex example
          |       cut.function.6          text      html      latex      example
          | * DONE (fcut)

```

In der Tat, es ist ein neues Verzeichnis entstanden. Prüfen wir den Inhalt.

```

in 128: | setwd(bilde.pfad("test", "mylib"))
          | system("ls -R")

```

```

out 128: | .:
          | R.css fcut
          |
          | ./fcut:
          | CONTENTS DESCRIPTION INDEX Meta R R-ex help html latex man
          |
          | ./fcut/Meta:
          | Rd.rds hsearch.rds
          |
          | ./fcut/R:
          | fcut
          |
          | ./fcut/R-ex:
          | cut.function.6.R
          |
          | ./fcut/help:
          | AnIndex cut.function.6
          |
          | ./fcut/html:
          | 00Index.html cut.function.6.html
          |
          | ./fcut/latex:
          | cut.function.6.tex
          |
          | ./fcut/man:
          | fcut.Rd

```

Das sieht gut aus.

6.8 Test – wie testen wir unser Paket?

Nach der Installation können wir hoffentlich unsere Funktion einsetzen. Schauen wir zunächst, ob Bibliothek und Paket gefunden werden.

```

in 129: | library(lib.loc=bilde.pfad("test", "mylib"))

```

Wir erhalten:

```

out 129: | Packages in library '/home/wiwi/pwolf/R/R-kurs/R-steilkurs/test/mylib':
          | fcut                Funktionsschnitt

```

Nun löschen wir mutig unsere Funktion. (Achtung, wer sich nicht sicher ist, sollte das natürlich nicht machen.) Jetzt öffnen unser privates Paket.

```

in 130: | rm(cut.function.6)
          | library(fcut, lib.loc=bilde.pfad("test", "mylib"))
          | search()

```

```

out 130: | [1] ".GlobalEnv"      "package:fcut"    "package:methods" "package:stats"
          | [5] "package:graphics" "package:utils"  "Autoloads"      "package:base"

```

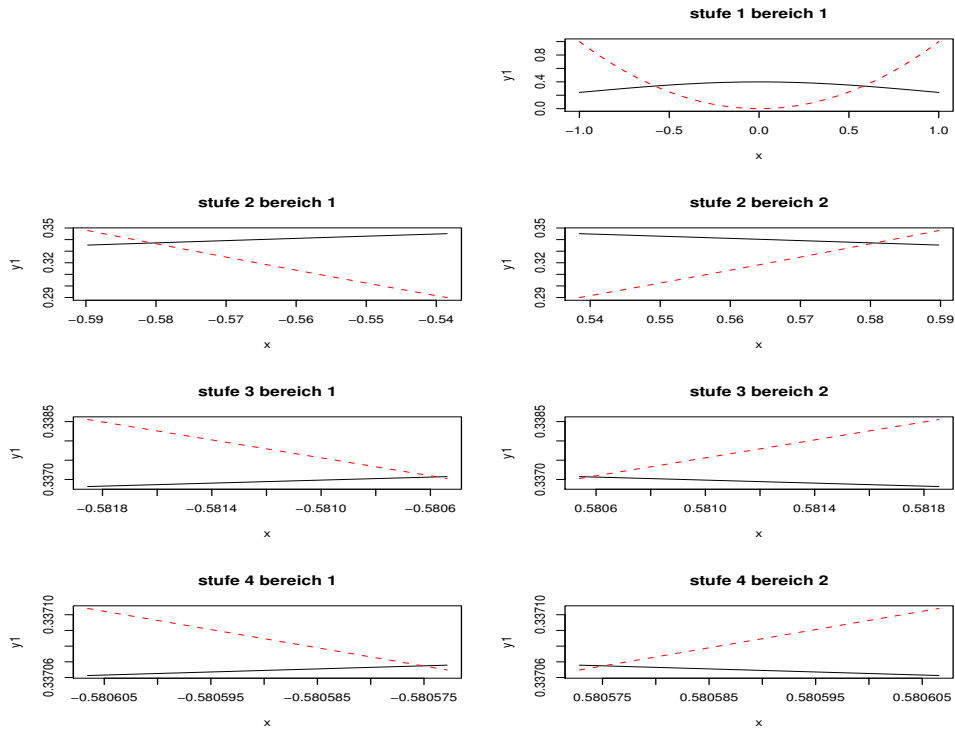
Der Einsatz folgt:

```

in 131: | cut.function.6("dnorm", "x^2", 0:1, NULL, -1, 1)

```

Wir erhalten:



Weiter funktioniert:

```
in 132: | help(cut.function.6)
```

sowie auch

```
in 133: | help.start(browser="mozilla")  
        | help(cut.function.6)
```

Wer hätte das gedacht! Nach dem Studium von nur 133 Code-Chunks hat der Leser einen Überblick über das Programmieren mit R erhalten. Weitergehende Fragen müssen Sie jedoch über andere Quellen oder durch eigene Experimente herausfinden. Zum Abschluss sei das Rätsel des Auffindens von Schnittstellen noch einmal explizit gelöst: Sind die Normalverteilungsdichten mit den Parametervektoren $(\mu = 0, \sigma = 1)$ und $(\mu = 3, \sigma = 4)$ gegeben, dann erhalten wir den Schnittpunkt im Intervall $[0, 4]$ leicht durch Aufruf von:

```
in 134: | uniroot(function(x) dnorm(x, 0, 1) - dnorm(x, 3, 4), c(0, 4))
```

7 Literatur

Als Start für das Literaturstudium werden nur wenige Quellen genannt. Weitere lassen sich über die R-Projektseite <http://cran.at.r-project.org> finden.

H. P. Wolf (2004): Kleiner R-Steilkurs,
<http://www.wiwi.uni-bielefeld.de/~wolf/lehre/ss04/liptex/Rkurs.pdf>

W. N. Venables, D. M. Smith and the R Development Core Team (2004): An Introduction to R,
<http://cran.at.r-project.org/doc/manuals/R-intro.pdf>

R Development Core Team (2004): Writing R Extensions,
<http://cran.at.r-project.org/doc/manuals/R-exts.pdf>