Per Mondlandungen und flying helicopters ad R

File: prog-heli-moon.rev
in: /home/pwolf/lehre/rkurs/Programmieren

März 2015

Inhalt

1 Idee und Weg zum Spielen **3**

2 Flugbahn fallender Objekte **4**

2.1 Darstellung der Wegstrecke eines freien Falls 4
2.2 Darstellung des freien Falls 5
2.3 Freier Fall im Zeitablauf 6
2.4 Ein Komet mit Schweif 7
2.5 Fallbeendigung an der Oberfläche 8
2.6 Ausschmückung der Landung 9
2.7 Darstellung einer Flugkurve 10
2.8 Darstellung mehrerer Flugobjekte 12

3 Wanderungen durch eine Hügellandschaft **13**

3.1 Definition einer Hügel-Landschaft 13
3.2 Darstellung der Normalverteilungslandschaft 14
3.3 Konstruition einer Hügellandschaft 15
3.4 Ein Linienzug im Raum der Hügel-Landschaft 16
3.5 Hervorhebung des Gipfels 17
3.6 Darstellung eines Weges zum Gipfel 18

4 Hubschrauberflüge über eine Landschaft **19**

4.1 Hubschraubersteuerung 19
4.2 Benutzereingaben für Hubschrauberflug 21
4.3 Anwender gesteuerter Hubschrauberflug 23
4.4 Zielflug unter Windeinfluss 24
4.5 Eine allgemeine Beschreibung einer Oberfläche 25

4.6 Definition eines Handlungsteils und einer Oberfläche 26
4.7 Zusammenbau und Start der Hubschrauber-Simulation 27

5 Hubschrauberflüge auf einen Berggipfel 28
5.1 Hubschrauber für Landschaft definieren 28
5.2 Hubschrauber durch Landschaft steuern 29
5.3 Hubschraubersteuerung mit Cockpit-Anzeige 30
5.4 Start und Zielkreuz wählen und Flugsimulation starten 31

6 Landungen auf Monden und Himmelskörpern 32
6.1 Mondlandung – das riskante Spiel 32
6.2 Bemerkungen zum Programmieren 33
6.3 Eine grobe Struktur zur Lösung 34
6.4 Die Umsetzung der einzelnen Schritte 35
6.4.1 Argumentencheck 35
6.4.2 Bremsbefehleingabe 36
6.4.3 Konstruktion einer Ausgangssituation 38
6.4.4 Situationsanzeige 40
6.4.5 Funktionsende 41
6.4.6 Abbruch, wenn Oberfläche errreicht 41
6.4.7 Schleifenverpackung 42
6.4.8 Berechnung der Auswirkung eines Bremsmanövers 42
6.4.9 Darstellung der Höhenentwicklung 44
6.5 Auf zum Spielen 44
6.6 Aufgaben 44
6.7 Die Spielfunktion im Zusammenhang 45

7 Aufgabe: Erfinde ein Mondlandesimulationsspiel 47

8 Anhang 48

1 Idee und Weg zum Spielen

Die Entwicklung von Spielen kann gut als Medium für eine Einführung in R und seine Programmierung herhalten. In dem Kontext von Fluggeräten werden im Folgenden kleine Probleme und deren Lösung vorgestellt, die der Leser dann weiterentwickeln und so spielerisch eigene kleine Computer-Spiele zusammenbauen soll. Auf dem Wege der Lösungen werden eine Reihe von Konzepten berührt, die für die Programmierung relevant sind. Diese Konzepte werden aber nur stichwortartig genannt und müssen beispielsweise im Rahmen eines Kurses mit Leben gefüllt werden.

Damit der Leser nicht aus allen Wolken fällt, möge er sich an Alltagssituationen erinnern, in denen sich bewegende Objekte eine Rolle spielen. Auch kann er sich mit Gedankenexperimenten beschäftigen und Fragen nachgehen: Wie fallen Äpfel von Bäumen, Toastscheiben vom Tisch und Eier aus einem Nest? Dabei werden wahrscheinlich auch Erinnerungen an den Physik-Unterricht geweckt. Wie war das noch mit den einfachen Gesetzen der Kinetik? Die zurückgelegte Strecke hängt bei einem Körper, der sich mit konstanter Geschwindigkeit bewegt, linear von der Zeit ab: $s = v \cdot t$. Für einen fallenden Körper errechnet uns $s = g/2 \cdot t^2$ $\rightarrow$ die Fallstrecke nach der Zeit t.

Zur Strukturierung verwenden wir im Folgenden einige Kürzel:

P=Problem, S=Lösungsschritt, H=Hinweis, A=Aufgabe, W=Wissen, T=Technik

2 Flugbahn fallender Objekte

T: Formelumsetzung, Operationen mit Vektoren, literates Programmieren, Verfahrensparameter als Variablen, for-Schleife, Kommentare zur Erklärung, sinnvolle Namen, Graphik-Parameter, Indexzugriffe, Systemfunktionen, while-Schleife, Definition von Bedingungen, Text-Ausgabe, if-Konstruktionen, Funktionsdefinition, Funktionsargumente, Funktionkommentierung, Defaultwerte, Argumentenprüfung, Funktionsaufruf, Zufallszahlengeneratoren

2.1 Darstellung der Wegstrecke eines freien Falls

P: Wie lässt sich die Wegstrecke des freien Falls darstellen?

S: Größen fixieren

1 ⟨*setze* `zeit` 1⟩ ≡ ⊂ 3, 5, 6, 7, 8

```
zeit <- seq(from=0, to=10, length=100)
g <- 9.80665
```

H: Variablen, Vektoren, Dezimalzahlen, Funktionsaufruf, benannter Chunk

S: Fallstrecken berechnen

2 ⟨*berechne Strecke aufgrund der Fallzeit* `zeit` 2⟩ ≡ ⊂ 3, 5, 6, 7, 8

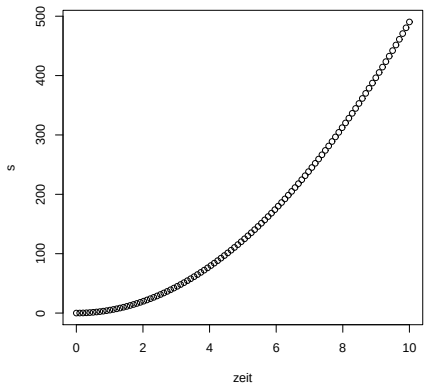
```
s <- g/2 * zeit^2
```

H: einfache Berechnungen

S: Darstellung der Fallstrecke

3 ⟨* 3⟩ ≡

```
⟨setze zeit 1⟩
⟨berechne Strecke aufgrund der Fallzeit zeit 2⟩
plot(zeit, s)
```



H: Graphischer Output.

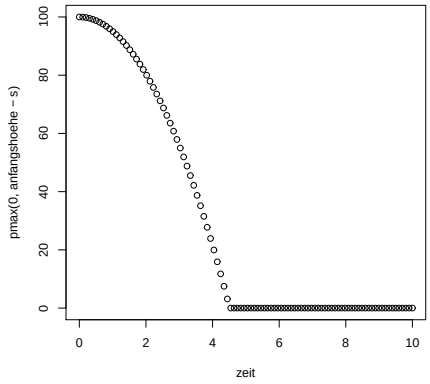
A: Was müsste man ändern, wenn sich g mit jedem Zeitschritt um den Faktor 1.05 vergrößern würde?

2.2 Darstellung des freien Falls

S: Darstellung der Höhe im Zeitablauf

4 ⟨*setze Anfangshöhe* 4⟩ ≡ ⊂ 5, 6, 7, 8
```
anfangshoege <- 100
```

5 ⟨* 3⟩+ ≡
```
⟨setze zeit 1⟩
⟨berechne Strecke aufgrund der Fallzeit zeit 2⟩
⟨setze Anfangshöhe 4⟩
plot(zeit, pmax(0, anfangshoehe - s))
```



H: Funktionsergebnis als Input für Plotfunktion.

A: Stelle die Höhenentwicklung dar, die ein nach oben geworfenes Ei, Apfel oder Ball durchlebt. Hierbei gehen wir davon aus, dass der Wurfkörper 10 Zeiteinheiten in der Luft ist.

Wie lange fliegt der Körper hoch? Wie lange hinunter?

Welche Fallhöhe ist mit der Fallzeit verbunden?

Wie lässt sich die Fallphase darstellen?

Wie lässt sich die Steigphase daraus ableiten?

Wie lassen sich die Objekte `zeit` und `hoehe` zusammensetzen?

Wie lässt sich damit die Flugphase darstellen?

A: Stelle die Höhenentwicklung eines Kometen dar. Gehe hierbei gehen davon aus, dass der Komet anfangs eine Höhe von `anfangshoehe` und eine Geschwindigkeit von `v0` hat.

Wie lange fällt ein Körper mit Geschwindigkeit `v0`? $\rightarrow v = g \cdot t \;\Rightarrow\; t = v/g$

Welche Höhe gehört zu dieser Fallzeit? $s = g/2 \cdot t^2 = g/2 \cdot (v/g)^2 = v^2/(2g)$

Wie ließe sich der Fall von dieser Höhe plus der Anfangshöhe darstellen?

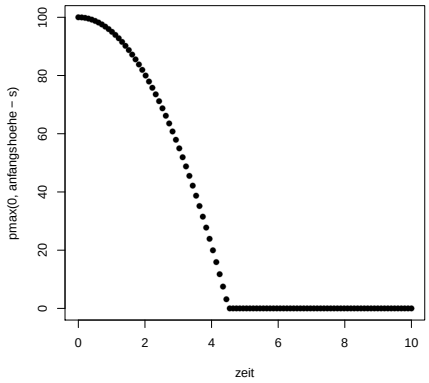
Wie ließe sich diese Darstellung auf den Teil ab der Anfangshöhe reduzieren?

2.3 Freier Fall im Zeitablauf

P: Wir wollen den freien Fall im Zeitablauf verfolgen.

S: Darstellung soll sich im Zeitablauf ändern. Lösung per for-Schleife mit einer Indexlaufvariablen.

6 ⟨ * 3⟩+ ≡
```
⟨setze zeit 1⟩
⟨berechne Strecke aufgrund der Fallzeit zeit 2⟩
⟨setze Anfangshöhe 4⟩
plot(zeit, pmax(0, anfangshoehe - s), col="gray")
for(i in 1:length(zeit)){
  points(zeit[i], pmax(0, anfangshoehe - s)[i], pch=16)
  Sys.sleep(.04)
}
```



H: Wiederholte Ausführung, for-Schleife, Zusammenfassung von Anweisungen, Zugriffe auf einzelne Elemente von Vektoren, `pmax`, `Sys.sleep`, Graphik-Parameter `pch`.

A: Welche Operation lässt sich oben einsparen?

A: Überlege den Unterschied zwischen: `1:length(zeit)`, `zeit`, `seq(zeit)` und `seq(along=Zeit)`.

A: Was ist der Unterschied zwischen `max` und `pmax`?

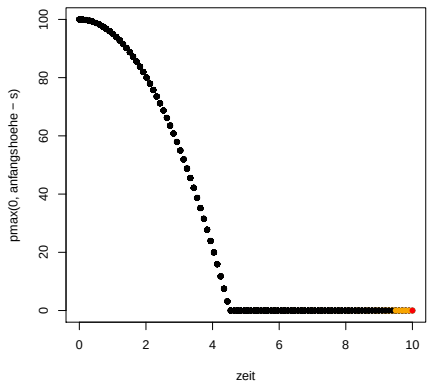
2.4 Ein Komet mit Schweif

P: Körper soll einen Schweif hinter sich herziehen.

S: Wir können schrittweise bestimmte Punkte geeignet einfärben.

7 ⟨ * 3⟩+ ≡

```
⟨setze zeit 1⟩
⟨berechne Strecke aufgrund der Fallzeit zeit 2⟩
⟨setze Anfangshöhe 4⟩
# Parametersetzung: Schweiflaenge, Pausierzeit
n.schweif <- 5; schlafzeit <- 0.04
# Plotinitialisierung
plot(zeit, pmax(0, anfangshoehe - s), col="gray")
# Schleife ueber Zeitpunkte, zuerst Fixierung der Menge zur Iteration
z.set <- seq(along=zeit)
for(z in z.set){
  # Hervorhebung der bisherigen Flugbahn
  idx <- 1:z
  points(zeit[idx], pmax(0, anfangshoehe - s)[idx], pch=16)
  # Extraktion und Darstellung der Schweif-Punkte
  idx <- max(1, z-n.schweif) : z
  points(zeit[idx], pmax(0, anfangshoehe - s)[idx], col="orange", pch=16)
  # Darstellung der Flugobjektes
  points(zeit[z],   pmax(0, anfangshoehe - s)[z],   col="red",    pch=16)
  # Wartezeit
  Sys.sleep(schlafzeit)
}
```



H: Fixierung veränderlicher Größen als Parameter, for-Schleife über Durchlauf einer Menge, Zugriff auf Teilvektoren, Kommentierung bedeutsamerSchritte, Graphik-Parameter `col`.

A: Erweitere eigene Lösung zum Eierwurf um einen Schweif und stelle Wurf schrittweise dar.

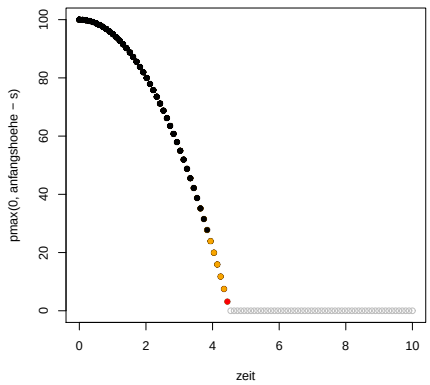
2.5 Fallbeendigung an der Oberfläche

P: Ausblendung des Stückes nach Landung

S: Abbruch der Iteration nach Höhe 0.

8 ⟨*zeige Flug mit Schweif* 8⟩ ≡ ⊂ 9

```
⟨setze zeit 1⟩
⟨berechne Strecke aufgrund der Fallzeit zeit 2⟩
⟨setze Anfangshöhe 4⟩
# Parametersetzung: Schweiflaenge, Pausierzeit
n.schweif <- 5; schlafzeit <- 0.04
# Plotinitialisierung
plot(zeit, pmax(0, anfangshoehe - s), col="gray")
# Schleife ueber Zeitpunkte, zuerst Fixierung der Menge zur Iteration
z <- 1
while( z <= length(s) && 0 < (anfangshoehe - s)[z]){
  # Hervorhebung der bisherigen Flugbahn
  idx <- 1:z
  points(zeit[idx], pmax(0, anfangshoehe - s)[idx], pch=16)
  # Extraktion und Darstellung der Schweif-Punkte
  idx <- max(1, z - n.schweif) : z
  points(zeit[idx], pmax(0, anfangshoehe - s)[idx], col="orange", pch=16)
  # Darstellung der Flugobjektes
  points(zeit[z],   pmax(0, anfangshoehe - s)[z],   col="red",    pch=16)
  # Wartezeit
  Sys.sleep(schlafzeit)
  z <- z+1
}
```



H: while-Schleife, Bedingungs-Kombination, sicherer Schleifenabbruch.

A: Verändere die Kometenlösung um die vorgestellten Ideen.

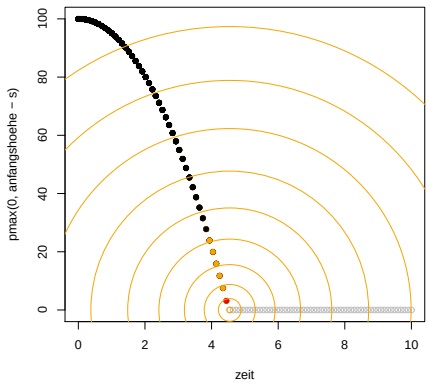
2.6 Ausschmückung der Landung

P: Zeige Aufschlagserscheinung

S: Ermittlung und Verarbeitung des Landungszeitpunktes.

9 ⟨ * 3⟩+ ≡

```
⟨zeige Flug mit Schweif 8⟩
z0 <- which(anfangshoehe - s <= 0)[1]
if( !is.na(z0) ){
  cat("Landung im Zeitpunkt", z0, "\n")
  for( i in 1:10 ){
    points(zeit[z0], 0, cex=i^2, col="orange", pch=1)
    Sys.sleep(.3)
  }
} else {
  print("Landung noch nicht erfolgt")
}
```



H: Chunk-Verwendung, `which`, `is.na`, if-Konstruktion, cat- und print-Ausgabe, for-Zählschleife, geschachtelte Kontrollstrukturen, Graphik-Parameter `cex`.

A: Baue Landungsdarstellung in die eigenen Lösungen ein.

A: Entwerfe Landungsdarstellungen, bei denen das Ei wie beim Comic in verschiedene Richtungen spritzt.

2.7 Darstellung einer Flugkurve

P: Darstellung einer Flugkurve in Abhängigkeit einer Koordinaten.

W: Da sich Bewegungsvektoren vektoriell zerlegen lassen, können wir die Richtung der Bewegung eines geworfenen Objektes in jedem Zeitpunkt in zwei Komponenten zerlegen: nämlich in die Komponente, die durch die Erdanziehung entsteht, und die zweite, die durch die Ausgangssituation verursacht ist. Letztere ist durch die Bewegungsrichtung am Anfang und die Anfangsgeschwindigkeit festgelegt. Ohne Berücksichtigung des Luftwiderstands bleibt die zweite Komponente unverändert erhalten, die erste führt zu einer beständig steigender Geschwindigkeit.

S: Berechnung des Ortes in Abhängigkeit von Anfangsort, Anfangsgeschwindigkeit, Anfangsrichtung und Zeit.

10 ⟨* 3⟩+ ≡

```
compute.flight <- function(x0 = 0, y0, winkel = 0, v0 = 0, zeit){
  # function to compute new position of a falling object after time zeit
  # Input:
  #   x0     : Anfangs-x-Koordinate
  #            oder 2-elementiger Vektor x0, y0
  #   y0     : Anfangs-y-Koordinate
  #   winkel : ist der Winkel zum Erdboden
  #   v0     : Anfangsgeschwindigkeit
  #   zeit   : Flugzeit(en)
  # Output:
  #   neue Koordinaten
  # pw 150325, version 1
  # ------------------------------------------------------------
  # Input-Check:
  if( missing(y0) ){
    if( 2 != length(x0) ) return("Error: kein y-Wert angegeben.")
    y0 <- x0[2]; x0 <- x0[1]
  }
  # ------------------------------------------------------------
  # Konstanten-Setzung
  g <- 9.80665
  # ------------------------------------------------------------
  # Fallbewegung
  s.fall <- g/2 * zeit^2
  # ------------------------------------------------------------
  # Anfangsbewegung
  s.bewegung <- v0 * zeit
  winkel.2.pi <- 2 * pi * winkel / 360
  x.del <- s.bewegung * cos(winkel.2.pi)
  y.del <- s.bewegung * sin(winkel.2.pi)
  # ------------------------------------------------------------
  # neue Koordinaten
  x <- x0 + x.del
  y <- y0 + y.del - s.fall
  # ------------------------------------------------------------
  # Output
  return( if( 1 == length(x) ) c(x,y) else cbind(x, y) )
}
```

H: Definition einer Funktion, Argumente, Default-Werte, Rumpf, Kommentierung `missing`, `return`, `cos`, `sin`.

S: Check der Funktion

11 ⟨ * 3⟩+ ≡

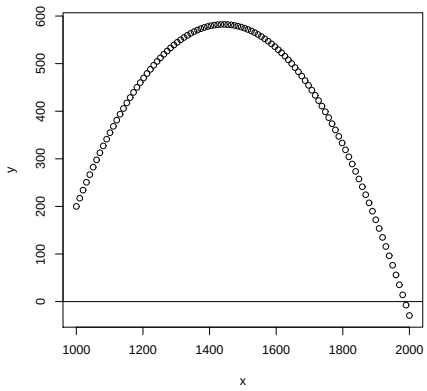
```
compute.flight( 0, 100, v0 = 10, zeit=0:5)
```

```
     x         y
[1,]  0 100.00000
[2,] 10  95.09668
[3,] 20  80.38670
[4,] 30  55.87007
[5,] 40  21.54680
[6,] 50 -22.58312
```

S: Darstellung der Flugkurve

12 ⟨ * 3⟩+ ≡

```
x <- 1000
y <- 200
v.begin <- 100
winkel <- 60
zeit <- seq(0, 20, length=100)
xy <- compute.flight(x, y, winkel, v0 = v.begin, zeit = zeit)
plot(xy)
abline(h = 0)
```



H: Formal versus actual parameter.

A: Wie lässt sich die Flugkurve in einer Funktion zusammenfassen?

Welche Parameter eignen sich für die Funktion?

Welche Checks sollte man vor der Bearbeitung durchführen?

Welche Alternativen bzgl. der Darstellung machen Sinn?

Was sollte die Funktion ausgeben?

Wie könnte eine geeignete Kommentierung aussehen?

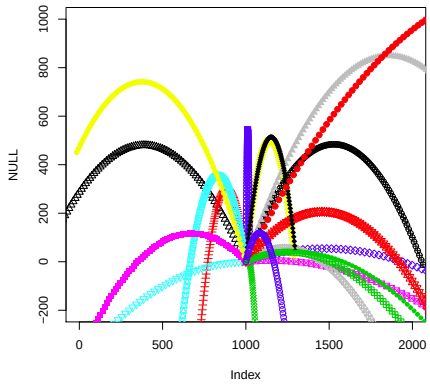
2.8 Darstellung mehrerer Flugobjekte

P: Darstellung mehrerer Flug-Objekte

S: Wiederholter Aufruf der Flugkurvenberechnung und Darstellung von mehrerer Objekte, die aus einer Quelle starten.

13 ⟨ * 3⟩+ ≡

```
n <- 20
X <- rep(1000, n)
Y <- rep(0, n)
set.seed(13)
v.begin <- rgamma(n, 5) * 20
winkel <- runif(n, 0, 180)
zeit <- seq(0, 20, length=100)
coor <- NULL
for(i in 1:n){
  co <- compute.flight(X[i], Y[i], winkel[i], v0 = v.begin[i], zeit = zeit)
  coor <- cbind(coor, co)
}
plot(NULL, xlim = c(0,2000), ylim = c(-200,1000))
for(i in 1:n){
  points(coor[, (-1:0) + 2 * i], col = i, pch = 1 + (i %% 20))
}
```



H: Zufallszahlengeneratoren, `set.seed`, Matrix-Zugriffe, `%%`, schrittweise Ergebnisspeicherung.

A: Wie lassen sich diese Anweisungen geeignet in eine Funktion gießen?

A: Wie lässt sich die zeitliche Entwicklung der Bahnen umsetzen.

3 Wanderungen durch eine Hügellandschaft

T: normalverteite Zufallszahlen, 3D-Darstellungen, geschachtelte Schleifen, Zugriffe auf Matrizen, Transformationen vom $\mathcal{R}^3$ in den $\mathcal{R}^2$, situationsabhängige Bearbeitungen

3.1 Definition einer Hügel-Landschaft

P: Wir wollen eine Landschaft kreieren.

S: Definiere eine Normallandschaft mit Hilfe einer 2-dim-Normalverteilung und berechne die Höhen an den Punkten eines Gitternetzes.

14 ⟨*erstelle Normalgebirge* 14⟩ ≡ ⊂ 15, 16

```
# Raum festlegen:
#   m Punkte in der ersten Dimension (x-Richtung)
#   n Punkte in der zweiten Dimension (y-Richtung)
m <- 50; n <- 30
# Normalgebirge berechnen und zeichnen
hill <- outer(dnorm(1:m, 3*m/4, sqrt(m/3)),
              dnorm(1:n, 2*n/3, sqrt(n/3)), "*")
# Bergspitze auf 1 normieren
hill <- hill/max(hill)
```

H: `outer` erzeugt durch elementweise Kombination der Elemente zweier Vektoren unter Verwendung der angegebenen Operation eine Matrix.

A: Baue Landschaft aus zwei oder drei Hügeln.

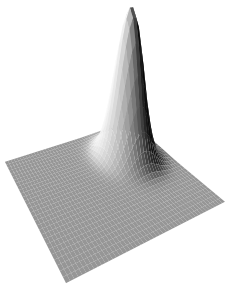
A: Wie ließe sich eine Kraterlandschaft erzeugen?

3.2 Darstellung der Normalverteilungslandschaft

P: Wie können wir die Hügellandschaft darstellen?

S: Zeichne eine perspektivische Darstellung der Landschaft.

15 ⟨*zeichne Normalgebirge* 15⟩ ≡
⟨*erstelle Normalgebirge* 14⟩
```
res <- persp(1:m, 1:n, hill, phi=30, theta=-30, shade=0.75, border=FALSE, box=FALSE)
```



H: `persp`.

A: Perspektivische Darstellungen, Studiere Hilfeseite zu `persp`.

A: Variiere Darstellung durch Variation von `phi`, `theta`, `shade`.

A: Was bewirkt `col="red"`?

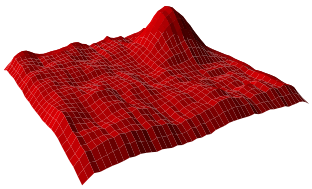
3.3 Konstruition einer Hügellandschaft

P: Eine Landschaft sieht nicht so wie eine Normalverteilung aus.

S: Addition von Störungen auf das Normalgebirge.

16 ⟨*zeige Landschaft* 16⟩ ≡ ⊂ 17, 18, 35
⟨*erstelle Normalgebirge* 14⟩

```
# kleine Landschaftsunebenheiten einbauen
unebenheiten <- matrix(rgamma(m * n, 5), m, n)
unebenheiten <- unebenheiten / max(unebenheiten)
# Stoerungen glaetten
xy.expand <- cbind(0, 0, unebenheiten, 0, 0)
xy.expand <- rbind(0, 0, xy.expand,    0, 0)
for(i in 1:m){
  for(j in 1:n){
    unebenheiten[i, j] <- mean(xy.expand[i : (i+4), j : (j+4)])
  }
}
# Stoerungen und Berg addieren
lambda <- 0.4
landschaft <- lambda * hill + (1-lambda) * unebenheiten
# Landschaft darstellen
res <- persp(1:m, 1:n, landschaft, phi = 20, theta = -30,
             zlim = c(0, 1.5 * max(landschaft)), border=FALSE,
             box=FALSE, shade = 0.75, expand = 0.5, col = "red")
```



H: `rgamma`, geschachtelte Schleifen, `rbind`, `cbind`, Linearkombinationen, Transformationsmatrix.

A: Wie ließen sich die Störungen in Abhängigkeit der Hügelhöhe einbauen?

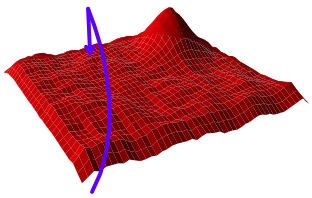
3.4 Ein Linienzug im Raum der Hügel-Landschaft

P: Wie können wir einen Linienzug in den Raum einzeichnen?

S: Definiere Koordinaten eines Pfades, transformiere und zeichne.

17 ⟨*zeichne Pfad ein* 17⟩ ≡
⟨*zeige Landschaft* 16⟩

```
n.pfad <- 10
# Bewegung linear in x
x <- (1 : n.pfad)*2
# Bewegung quadratisch in y
y <- (1 : n.pfad)^2 / 4
# Bewegung linear hoch
z <- seq(min(landschaft), max(landschaft), length = n.pfad)
# Transformation der Koordinaten
txy <- trans3d(x, y, z, pmat = res)
# Linienzug der berechneten Punkte
lines(txy, col = "blue", lwd = 5)
# Lotpfeil auf Landschaft
x <- x[n.pfad]; y <- y[n.pfad]; z <- z[n.pfad]
txy1 <- trans3d(x, y, z , pmat = res)
txy2 <- trans3d(x, y, landschaft[x,y] , pmat = res)
arrows(txy1$x, txy1$y, txy2$x, txy2$y, col = "blue", lwd = 5, length = .1)
```



H: Transformation mit `trans3d`, Zugriff auf Teile des transformierten Objekts, Listenobjekt, Pfeile.

A: Zeichne Verbindung von einer unteren Ecke zum Mittelpunkt der Landschaft und dann zu einer anderen Ecke mit Hilfe von Pfeilen.

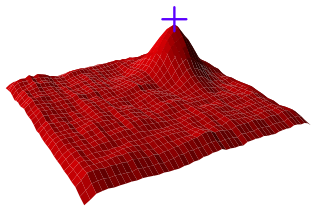
3.5 Hervorhebung des Gipfels

P: Wie können wir ein Gipfelkreuz auf dem Berg anbringen?

S: Suche Maximum, suche Stelle, transformiere Koordinaten, zeichne

18 ⟨*zeichne Bergspitze* 18⟩ ≡ ⊂ 19
⟨*zeige Landschaft* 16⟩

```
# Bergspitze suchen
idx <- which(landschaft == max(landschaft))
no.spalte <- ceiling(idx/m)
no.zeile  <- idx - ((no.spalte-1)*m)  # ident: 1 + (idx-1) %% m
hoehe <- landschaft[no.zeile, no.spalte]
# Koordinaten transformieren
txy <- trans3d(no.zeile, no.spalte, hoehe, pmat = res)
txy$y <- 1.15 * txy$y
# und zeichnen
points(txy, col="blue", cex=3, pch=3, lwd=3)
```



H: `which`, Vektorposition in Zeilen- und Spalten-Koordinaten umrechnen, `ceiling`, `%%`.

A: Markiere den tiefsten Punkt mit einem geeigneten Symbol.

A: Schreibe Funktion, die die Indizes der Zeile(n) und Spalte(n) des minimale oder maximalen Elements einer Matrix ermittelt und ausgibt.

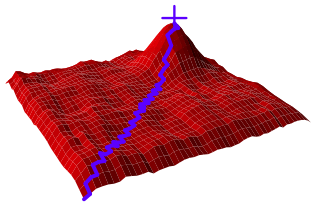
3.6 Darstellung eines Weges zum Gipfel

P: Wie können wir einen Weg zum Gipfel einzeichnen?

S: Schrittzahl berechnen, Schleife mit Entscheidungen in x- oder y-Richtung zu gehen, transformieren, zeichnen.

19 ⟨*zeichne Pfad zur Bergspitze* 19⟩ ≡
⟨*zeichne Bergspitze* 18⟩

```
n.steps <- no.zeile-1 + no.spalte-1
xn <- yn <- x <- y <- 1; z <- landschaft[1,1]
for(i in 1:n.steps){
  if( i < (n.steps - 3) ){ # zuerst Richtung Spitze
    if( no.spalte / no.zeile > y / x )     yn <- yn + 1 else xn <- xn + 1
  } else {                 # zuletzt Koordinate mit groesster Differenz
    if( (no.zeile - x) < (no.spalte - y) ) yn <- yn + 1 else xn <- xn + 1
  }
  zn <- landschaft[xn, yn]
  txy <- trans3d(c(x, xn), c(y, yn), c(z, zn), pmat = res)
  lines(txy, col="blue", lwd=5)
  x <- xn; y <- yn; z <- zn
}
```



H: geschachtelte if-Konstruktionen in for-Schleife.

A: Suche Weg des steilsten Anstiegs; wenn lokales Maximum, dann gehe zufällig in x- oder y-Richtung auf Gipfel zu.

A: Definiere Berg ohne Störungen und gehe möglichst ohne Höhenveränderung um den Berg herum.

4 Hubschrauberflüge über eine Landschaft

T: User-Eingaben, Einlesen von Zahlen, Objektkonvertierungen, Testen, reguläre Ausdrücke, Ergebnissammlungen, Oberflächenbau, Scoping, Wiederverwendbarkeit, Objektverwaltung, Umgebungen

4.1 Hubschraubersteuerung

P: Wie können wir einen Hubschrauber steuern?

S: Definiere Funktion zur Darstellung eines Hupschraubers.

20 ⟨*definiere Hubschrauber-Anzeige* 20⟩ ≡ ⊂ 21, 22, 29, 30, 34

```
show.helicopter <- function(x, y, direction = 45, col = 1, size = 1){
  # function to show the position of a helicopter in a graphics device
  # Input:
  #   x         : x coordinate of the helicopter
  #   y         : y coordinate of the helicopter
  #   direction : direction of movement
  #   col       : color of helicoper
  #   size      : size of helicoper
  # Output:
  #   graphcis output only
  # Darstellung des Hubschrauberrumpfes mit grossem Rotor
  points(x,y, cex = 2.5*size, pch=16, col = col, xpd=NA)
  points(x,y, cex = 4*size,   pch=3,  col = col, lwd=3*size, xpd=NA)
  # Umrechnung des Flugwinkels von Grad in Vielfache von 2 pi
  arc <- 2 * pi * direction / 360
  # Darstellung des Hinterteil und hinterer Rotor
  anz <- 5; usr <- par()$usr
  y.delta <- sin(arc) * 0.01 * (usr[4] - usr[3]) * (1:anz) * size
  x.delta <- cos(arc) * 0.01 * (usr[2] - usr[1]) * (1:anz) * size
  points(x - x.delta,      y - y.delta,
         cex = 1*size,   pch=16, xpd=NA, col = col)
  points(x - x.delta[anz], y - y.delta[anz],
         cex = 1.5*size, pch=3,  xpd=NA, col = col, lwd=3*size)
}
```

H: WD: Funktionsdefinition mit Kommentierung, `usr`, `xpd`, Winkelumrechnung

S: Test der Darstellungsfunktion

21 ⟨*teste Hubschrauber* 21⟩ ≡

```
⟨definiere Hubschrauber-Anzeige 20⟩
x <- y <- 15
plot(1:30, type="n")
show.helicopter(x, y, size=3)
```

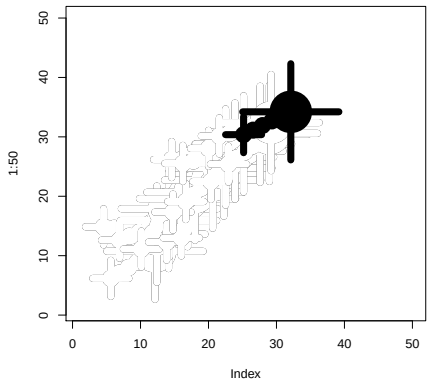
H: Überlege Zweck → entwerfe → definiere → teste → verwende → verbessere.

A: Was gefällt an der Darstellung? Was nicht? Verbessere Lösungsvorschlag!

S: Darstellung eines Zufallsflugs

22 ⟨*lasse Hubschrauber fliegen* 22⟩ ≡

```
⟨definiere Hubschrauber-Anzeige 20⟩
x <- y <- 10; dir <- 45; set.seed(13)
plot(1:50, type="n")
for( i in 1:15){
  show.helicopter(x, y, dir, size=3, col = "white")
  x <- x + runif(1,0,3)
  y <- y + runif(1,0,3)
  dir <- (dir + runif(1, -45, 45)) %% 360
  show.helicopter(x, y, dir, size=3)
  Sys.sleep(.3)
}
```



H: Löschen durch Übermalen mit weißer Farbe.

A: Überführe die Anweisungen in eine Funktion.

A: Was stimmt nicht in dem produzierten Film? Verbessere!

A: Programmiere Looping-Flug.

A: Ist die Anweisung mit `... %% 360` notwendig?

A: Was müsste man ändern, um mehrere Flugobjekte darzustellen?

A: Wie ließe sich die letzte Aufgabe rekursiv lösen?

4.2 Benutzereingaben für Hubschrauberflug

P: Wie können wir Benutzerwünsche abfragen?

S: Mit `readline()` Text einlesen und in Zahlen umwandeln.

23 ⟨*definiere* `get.number() 23`⟩ ≡ ⊂ 29, 30, 34

```
get.number <- function(message = "Input:"){
  as.numeric(readline(message))
}
```

H: Konvertierung von Objekten.

A: Was kann beim Einlesen alles schief gehen? Teste die Funktion!

A: Welche Kommentare sind angebracht?

S: leere Eingabe

24 ⟨*definiere* `get.number() 23`⟩+ ≡ ⊂ 29, 30, 34

```
get.number <- function(message = "Input:"){
  number <- readline(message)     # Einlesen
  if( 0 == nchar(number) ){       # leere Eingaben wiederholen
    number <- get.number(message)
    return(number)
  }
  as.numeric(number)              # Umwandlung in Zahl und Output
}
```

H: rekursive Funktion, Rekursion zur Behebung von Falscheingaben.

A: Wie viele Leereingaben lassen sich umsetzen?

S: Einlesewiederhoung bei Falscheingabe.

25 ⟨*definiere* `get.number() 23`⟩+ ≡ ⊂ 29, 30, 34

```
get.number <- function(message = "Input:"){
  number <- readline(message)     # Einlesen
  if( 0 == nchar(number) ){       # leere Eingaben wiederholen
    number <- get.number(message)
    return(number)
  }
  number <- as.numeric(number)    # Umwandlung in Zahl
  if( is.na(number) ){            # falsche Eingaben wiederholen
    number <- get.number(message)
  }
  return(number)                  # Output
}
```

H: Wiederholung einer Idee, explizites Ergebnis.

S: Zusammenführung der Prüfung.

26 ⟨*definiere* `get.number() 23`⟩+ ≡ ⊂ 29, 30, 34

```
get.number <- function(message = "Input:"){
  number <- readline(message)     # Einlesen
  number <- as.numeric(number)    # Umwandlung in Zahl
  if( is.na(number) ){            # falsche Eingaben wiederholen
    cat("FEHLER: keine Zahl eingelesen!\n")
    number <- get.number(message)
  }
  return(number)                  # Output
}
```

H: Verbessere Lösung schrittweise, Info über Fehler.

S: Abbruchbehandlung bzw. Notausstieg

27 ⟨*definiere* `get.number() 23`⟩+ ≡ ⊂ 29, 30, 34

```
get.number <- function(message = "Input:", exit.symbol = "X"){
  number <- readline(message)     # Einlesen
  if( 0 < length( grep( exit.symbol, number ) ) ){ # Exit?
    return(number)
  }
  number <- as.numeric(number)    # Umwandlung in Zahl
  if( is.na(number) ){            # falsche Eingaben wiederholen
    cat("FEHLER: keine Zahl eingelesen!\n")
    number <- get.number(message)
  }
  return(number)                  # Output
}
```

H: `grep`, reguläre Ausdrücke

S: Vorgabe eines Defaultwertes

28 ⟨*definiere* `get.number() 23`⟩+ ≡ ⊂ 29, 30, 34

```
get.number <- function(message = "Input:", default = 0, exit.symbol = "X"){
  number <- readline(message)     # Einlesen
  if( 0 < length( grep( exit.symbol, number ) ) ){ # Exit?
    return(number)
  }
  number <- as.numeric(number)    # Umwandlung in Zahl
  if( is.na(number) ){            # falsche Eingaben wiederholen
    cat("Hinweis: Leere Eingabe durch", default, "ersetzt!\n")
    number <- default
  }
  return(number)                  # Output
}
```

H: Fehlersituationen versus Ersatzwerte.

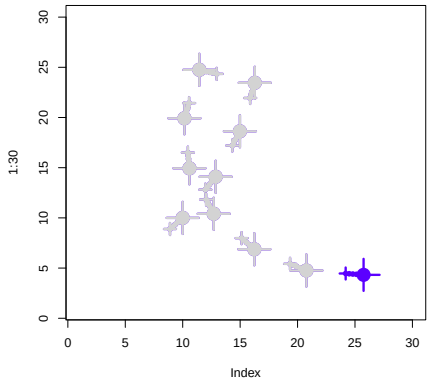
4.3 Anwender gesteuerter Hubschrauberflug

P: Wie können wir einen Hubschrauber nach Anwenderwünschen fliegen lassen?

S: Erfrage Parameter und setze Flug um.

29 ⟨*lasse Hubschrauber gemäß Anwender fliegen* 29⟩ ≡

```
⟨definiere Hubschrauber-Anzeige 20⟩
⟨definiere get.number() 23⟩
# Parameter setzen
x <- y <- 10; dir <- 45; dir.change <- 10; step <- 5; n <- 10
# Plot- und Hubschrauberinitialisierung
plot(1:30, type="n")
show.helicopter(x, y, col="blue", direction = dir)
# Schleife ueber Bewegungen
for(i in 1:n){
  # Einlesen und Aktualisierung der Richtung
  dir.change <- get.number("Richtung:", default = dir.change)
  if( 0 < length(grep("X", dir.change))) break
  dir.new <- (dir + dir.change) %% 360
  # Einlesen der Flugstrecke
  step <- get.number("Step:", default = step)
  if( 0 < length(grep("X", step))) break
  # Berechnung der neuen Koordinaten
  x.new <- x + cos(2 * pi * dir.new / 360) * step
  y.new <- y + sin(2 * pi * dir.new / 360) * step
  # Darstellung der neuen Position
  show.helicopter(x,  y,  dir,  col = "lightgray")
  show.helicopter(x.new, y.new, dir.new, col = "blue")
  # Vorbereitung der neuen Iteration
  x <- x.new; y <- y.new; dir <- dir.new
}
```



H: Voreinstellungen für Eingaben.

A: Informiere Anwender über Situation und biete Vorschlagswert an.

A: Überführe gesteuerten Flug in eine Funktion.

4.4 Zielflug unter Windeinfluss

P: Es gilt, ein Ziel anzufliegen. Dabei sollen Windeinflüsse den Flug stören.

H: Modellierung von Wind durch Zufallsbewegungen.

30 ⟨*lasse Anwender Hubschrauber zu einem Ziel fliegen* 30⟩ ≡
```
⟨definiere Hubschrauber-Anzeige 20⟩
⟨definiere get.number() 23⟩
# Parameter setzen
x <- y <- 10; dir <- 45; dir.change <- 10; step <- 5; n <- 10
y.end <- x.end <- 50
# Plot- und Hubschrauberinitialisierung
plot(1:100, type="n"); points(rep(x.end,5), rep(y.end,5), cex=1:5)
show.helicopter(x, y, col="blue", direction = dir)
# Schleife ueber Bewegungen
for(i in 1:n){
  # Einlesen und Aktualisierung der Richtung
  dir.change <- get.number("Richtung:", default = dir.change)
  if( 0 < length(grep("X", dir.change))) break
  dir.new <- (dir + dir.change) %% 360
  # Einlesen der Flugstrecke
  step <- get.number("Step:", default = step)
  if( 0 < length(grep("X", step))) break
  # Berechnung der neuen Koordinaten
  x.new <- x + cos(2 * pi * dir.new / 360) * step
  y.new <- y + sin(2 * pi * dir.new / 360) * step
  # Windeinfluss
  x.new <- x.new + runif(1, -1, 1) * step / 2
  y.new <- y.new + runif(1, -1, 1) * step / 2
  # Darstellung der neuen Position
  show.helicopter(x,  y,  dir,  col = "lightgray")
  show.helicopter(x.new, y.new, dir.new, col = "blue")
  # Vorbereitung der neuen Iteration
  x <- x.new; y <- y.new; dir <- dir.new
}
```

H: Wiederverwertbare Elemente, Auslagerungsmöglichkeiten in Funktionen.

A: Was könnte zur Information ausgegeben werden?

A: Baue ein Hindernis ein, das umflogen werden muss, sowie ein Sicherheitssystem, das dessen Umfliegen erzwingt.

4.5 Eine allgemeine Beschreibung einer Oberfläche

P: Verbesserung der Eingabe über eigenes Fenster.

S: Eine Funktion zur Eingabe.

31 ⟨*entwerfe Struktur für eine Oberflächenlösung* 31⟩ ≡

```
# definiere Funktionen -------------------------
what.to.do <- function(...){
  # hole Input zum ersten Parameter
  #     verarbeite ersten Parameter
  # hole Input zum zweiten Parameter
  #     verarbeite zweiten Parameter
  # setze Handlungen um
}
start.window <- function(...){
  # baue Steuerungspanel mit
  # integriere Erfragung Parameter 1
  # integriere Erfragung Parameter 2
  # integriere Abarbeitungstartknopf, der what.to.do aufruft
}
# starte Handlungen ----------------------------
#   initialisiere Prozess
#   starte Steuerungspanel
start.window()
```

H: Zunächst immer strukturelle Vorstellungen entwickeln.

A: Was ist zu ändern, wenn auch noch die Flughöhe zu berücksichtigen ist?

A: Wie könnte man ein Cockpit für den Piloten einbauen?

A: Was gehört alles zur Handlungsumsetzung?

A: Was muss alles zur Initialisierung geschehen?

4.6 Definition eines Handlungsteils und einer Oberfläche

P: Entwerfe Steuerungspanel für Helikopter-Flug.

S: Definiere `what.to.do()`; Parameter-Austausch per `slider()`.

32 ⟨*definiere* `what.to.do()` 32⟩ ≡ ⊂ 34

```
what.to.do <- function(...){
  # hole Input zum ersten Parameter: Richtungsvariation
  dir.change <- slider(no = 1)
  # verarbeite ersten Parameter
  if( is.na( dir.change <- as.numeric(dir.change) ) ) dir.change <- 0
  dir <- slider(obj.name = "dir")
  dir.new <- (dir + - dir.change) %% 360 # Achtung +/- Tausch
  # hole Input zum zweiten Parameter: Flugstrecke
  step <- slider(no = 2)
  # verarbeite zweiten Parameter
  if( is.na( step <- as.numeric(step) ) ) step <- 0
  # setze Handlungen um
  x <- slider(obj.name = "x")
  y <- slider(obj.name = "y")
  x.new <- x + cos(2 * pi * dir.new / 360) * step
  y.new <- y + sin(2 * pi * dir.new / 360) * step
  # Windeinfluss
  x.new <- x.new + runif(1, -1, 1) * step / 2
  y.new <- y.new + runif(1, -1, 1) * step / 2
  # Darstellung der neuen Position
  show.helicopter(x,  y,  dir,  col = "lightgray")
  show.helicopter(x.new, y.new, dir.new, col = "blue")
  # Vorbereitung der neuen Iteration
  x <- x.new; y <- y.new; dir <- dir.new
  slider(obj.name = "x", obj.value = x)
  slider(obj.name = "y", obj.value = y)
  slider(obj.name = "dir", obj.value = dir)
}
```

H: lokale versus globale Infos, Environments, Speicher von Infos zum globen Zugriff.

S: Definiere Oberfläche.

33 ⟨*definiere* `start.window()` 33⟩ ≡ ⊂ 34

```
start.window <- function(){
  do.nothing <- function(...) print("relax")
  slider(
  # baue Steuerungspanel mit Schiebern zur Parametereingabe
          do.nothing, sl.names = c("change of direction", "length of step"),
          sl.mins = c(-45, 0), sl.maxs = c(45, 20),
          sl.deltas = c(1,1), sl.defaults = c(0,0),
  # integriere Abarbeitungstartknopf, der what.to.do ruft
          but.functions = what.to.do, but.names = "doit")
}
```

H: tcltk, slider als Service-Funktion, Definition von Oberflächenelementen.

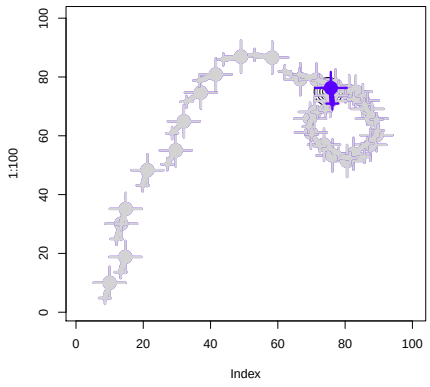
A: Was könnte die Oberfläche noch aufnehmen?

4.7 Zusammenbau und Start der Hubschrauber-Simulation

S: Einsatz der definierten Größen.

34 ⟨ * 3⟩+ ≡

```
# definiere Funktionen -------------------------
⟨definiere Hubschrauber-Anzeige 20⟩
⟨definiere get.number() 23⟩
⟨definiere what.to.do() 32⟩
⟨definiere start.window() 33⟩
# starte Handlungen ----------------------------
#   initialisiere Prozess
  # Parameter setzen
  x <- y <- 10; dir <- 75; dir.change <- -10; step <- 5; n <- 10
  y.end <- x.end <- 75
  slider(obj.name = "x", obj.value = x)
  slider(obj.name = "y", obj.value = y)
  slider(obj.name = "dir", obj.value = dir)
  # Plot- und Hubschrauberinitialisierung
  plot(1:100, type="n"); points(rep(x.end,5), rep(y.end,5), cex=1:5)
  show.helicopter(x, y, col="blue", direction = dir)
#   starte Steuerungspanel
start.window()
```



H: lokale versus globale Variablen, Scoping, Namensräume, Eventverarbeitung.

A: Teste Simulation: Versuche möglichst schnell zum Zielpunkt zu gelangen.

A: Wie können wir Landschaftsparameter berücksichtigen?

A: Wie können wir Geschwindigkeit als Parameter einbauen?

A: Wie können wir Flughöhen integrieren?

A: Wie könnte ein Cockpit aussehen?

5 Hubschrauberflüge auf einen Berggipfel

T: S3-Klassen- und Methoden, Programmierstil, Debugging, ...

5.1 Hubschrauber für Landschaft definieren

P: Wie können wir einen Helikopter in der Landschaft darstellen.

S: Vorschlag einer geeigneten Methode, um Objekte der Klasse Helikoper darzustellen.

35 ⟨*zeige Helikoper in Landschaft* 35⟩ ≡ ⊂ 36, 37
⟨*zeige Landschaft* 16⟩

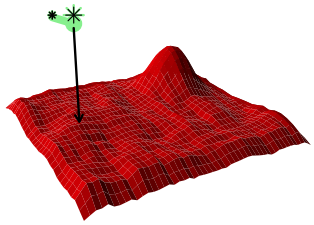
```
# Koordinaten definieren und zusammenfassen
z <- 1; x <- 10; y <- 15; xyz <- c(x, y, z)
# aktuelle Projektion an das Objekt kleben
attributes(xyz) <- list(resmat = res)
# Objekt als helicopter deklarieren
class(xyz) <- "helicopter"
# Darstellungsmethode verfassen
fly.helicopter <- function(x, ...){
  # Transformations-Matrix extrahieren
  res <- attributes(x)$resmat
  # Koordinaten beschaffen und transformieren
  xyz <- rbind(x); x <- xyz[, 1]; y <- xyz[, 2]; z <- xyz[, 3]
  txyz <- trans3d(x, y, z, pmat = res)
  Lot.punkt <- trans3d(x, y, landschaft[x, y], pmat = res)
  # Heli darstellen
  points(txyz, cex = 3, pch = 16, ..., xpd = NA)
  segments(txyz$x, txyz$y, txyz$x - .05, txyz$y + .015, ..., lwd = 10, xpd = NA)
  points(txyz$x - .05, txyz$y + .02, ..., lwd = 3, pch = 8, xpd = NA)
  points(txyz$x, txyz$y + 0.02, ..., lwd = 3, pch = 8, cex = 2.5, xpd = NA)
  points(txyz$x - .05, txyz$y + .02, lwd = 2, pch = 8, xpd = NA)
  points(txyz$x, txyz$y + 0.02, lwd = 1.5, pch = 8, cex = 2.0, xpd = NA)
  arrows(txyz$x, txyz$y - .01, Lot.punkt$x, Lot.punkt$y, length = 0.1, lwd = 3, xpd = NA)
}
# generische Methode verfassen
fly <- function(xyz, ...) UseMethod("fly")
# Default-Methode verfassen
fly.default <- function(x, ...){
  points(x, ...)
}
```

36 ⟨* 3⟩+ ≡
⟨*zeige Helikoper in Landschaft* 35⟩

```
fly(xyz, col="lightgreen")
```

H: neue S3-Klassen und Methoden definieren, Attribute ergänzen, Objekt mit Methode darstellen, `xpd`.

A: Verbessere Helikopter-Darstellung.



5.2 Hubschrauber durch Landschaft steuern

P: Wie können wir Hubschrauber durch die Landschaft steuern?

S: Kombiniere Panel-Ideen mit Hubschrauber-in-Landschaft-Darstellung.

37 ⟨*steuere Hubschrauber* 37⟩ ≡

```
⟨zeige Helikoper in Landschaft 35⟩
fly(xyz, col="lightgreen"); slider(obj.name = "xyz", obj.value = xyz)
control.heli <- function(...){
  # copy xyz of class helicopter + neue Koordinaten ermitteln
  delta <- c(slider(no=1), slider(no=2), slider(no=3))
  xyz.new <- xyz <- slider(obj.name = "xyz"); xyz.new[] <- xyz + delta
  slider(obj.name = "xyz", obj.value = xyz.new)
  # Darstellung an neuer Position
  res <- persp(1:m, 1:n, landschaft, phi = 20, theta = -30,
              zlim = c(0, 1.5 * max(landschaft)), border=FALSE,
              box=FALSE, shade = 0.75, expand = 0.5, col = "red")
  # fly(xyz, col="lightgray")
  fly(xyz.new, col="lightgreen")
  if( exists("show.heli.state") ) show.heli.state()
}
show.heli.state <- function(){
  cat("Flughoehe:", slider(obj.name = "xyz")[3], "\n")
}
do.nothing <- function(...) print("relax")
slider(
  # baue Steuerungspanel mit Schiebern zur Parametereingabe
          do.nothing, sl.names = c("delta x", "delta y", "delta z"),
          sl.mins = c(-2, -2, -.2), sl.maxs = c(2, 2, .2),
          sl.deltas = c(1,1,.1), sl.defaults = c(0,0,0),
  # integriere Abarbeitungstartknopf, der what.to.do ruft
          but.functions = control.heli, but.names = "fly"
)
```

H: Kombination Darstellung und Steuerung.

A: Überlege Mechanismus, um Lot vom Hubschrauber auf die Landschaft an oder auszuschalten.

5.3 Hubschraubersteuerung mit Cockpit-Anzeige

P: Wie lassen sich wichtige Infos im Darstellungsfeld einblenden.

S: Teile Graphik-Fenster mit Hilfe von `layout` in Bereiche ein.

38 ⟨ * 3⟩+ ≡

```
# definiere show.heli.state
show.heli.state <- function(){
  plot(1:8, type="n", axes = FALSE, xlab="", ylab="")
  text(2, 1, paste("Hoehe:", slider(obj.name = "xyz")[3]), adj=0, xpd = NA)
  text(4, 1, paste("x:",     slider(obj.name = "xyz")[1]), adj=0, xpd = NA)
  text(6, 1, paste("y:",     slider(obj.name = "xyz")[1]), adj=0, xpd = NA)
  title("Cockpit -- Infos:")
}
layout(rbind(1,2), heights = c(4,1.5))
control.heli()
slider(
  # baue Steuerungspanel mit Schiebern zur Parametereingabe
          do.nothing, sl.names = c("delta x", "delta y", "delta z"),
          sl.mins = c(-2, -2, -.2), sl.maxs = c(2, 2, .2),
          sl.deltas = c(1,1,.1), sl.defaults = c(0,0,0),
  # integriere Abarbeitungstartknopf, der what.to.do ruft
          but.functions = control.heli, but.names = "fly"
)
```

H: `layout`

A: Als wesentliche Info sollte im Cockpit auch die Höhe über dem Boden angezeigt werden. Ergänzen Sie diese.

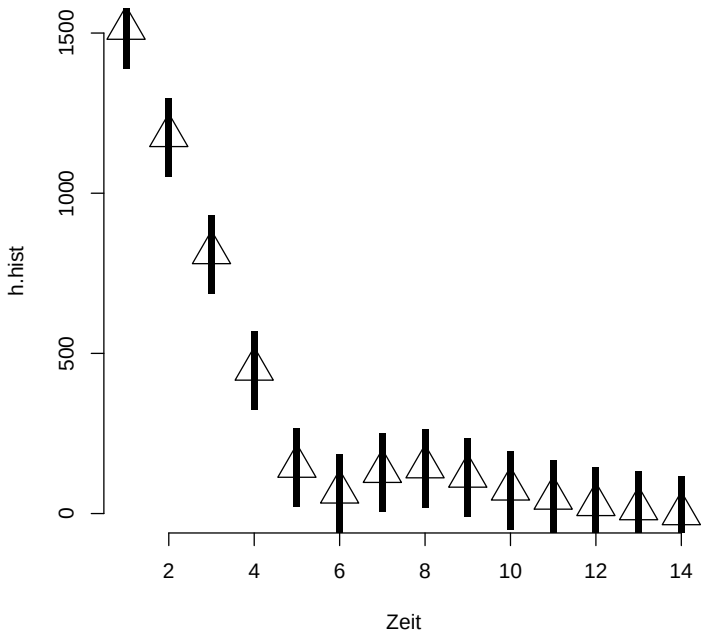
A: Zeige in der Landschaft den zurückgelegten Weg.

5.4 Start und Zielkreuz wählen und Flugsimulation starten

Dieser Unterabschnitt besteht nur aus einer großen Konstruktionsaufgabe.

A: Verwende die oben gegebenen Bestandteile und baue eine Flugsimulation für Hubschrauber. Dabei sollten verschiedene Gelände wählbar sein. Es sollten zufällig Start- und Landesituationen gezogen werden können. Es lässt sich Wind und Wetter einbauen. Auch sollten verschiedene der angesprochenen Verbesserungen integriert werden.

6 Landungen auf Monden und Himmelskörpern



6.1 Mondlandung – das riskante Spiel

Im Folgenden wird ein Spiel vorgestellt, in dem der Spieler mit einem Raumschiff auf einem fremden Himmelskörper eine Notlandung machen muss. Neben dem erhofften Spaßfaktor hat das Spiel den Zweck, den Prozess einer Lösung mit Hilfe von R zu verfolgen und dabei den Einsatz einiger Programmier-Werkzeuge zu erleben.

Rahmenbedingungen des Spiels

Mit dem Spielstart wird zufällig eine Landesituation erzeugt. Diese zeichnet sich durch folgende Parameter aus:

- Geschwindigkeit v_t, mit der sich das Raumschiff dem Himmelskörper nähert (`v_t`),
- Gravitationsstärke g des Himmelskörpers (`GRAVITATION`),
- Treibstoffvorrat (`tank`) auf dem Raumschiff,
- Höhe des Raumschiffs über der Oberfläche des Himmelskörpers (`hoehe`),
- maximale Landegeschwindigkeit `vmax`, mit der das Raumschiff sicher, also gerade noch ohne Crash landet.

Die in `Textschrift` eingefügten Namen finden sich später als Variablen in der Umsetzung wieder. Der Kapitän des Raumschiffs, der die wichtigsten Parameter kennt, kann als einzige Option das Raumschiff schrittweise abzubremsen. Dazu kann er für jedes Bremsmanöver:

- die einzusetzende Treibstoffmenge `treibstoffmenge` wählen.

Ausgehend von dieser Situation lassen sich natürlich schnell Erweiterungen einbauen, die den Landeprozess verkomplizieren, jedoch das Spiel interessanter machen können. Man könnte:

- die Dauer des Bremsmanövers wählbar machen,
- die Anzeige für den Kapitän stören,
- variierende Gravitationsfelder unterstellen,
- unterschiedliche Bremstechniken ergänzen,
- unterschiedliche Bodenbeschaffenheiten unterstellen,
- Bremswirkungen durch eine Gas-Atmosphäre einbauen
- und und und ...

Doch zunächst soll es einmal um die Grundkonstruktion gehen. Ziel des Spiels ist es, in der speziellen Landesituation das Raumschiff sicher zu landen. Das Ergebnis muss natürlich dem Spieler mitgeteilt werden.

6.2 Bemerkungen zum Programmieren

Damit man etwas zu Programmieren hat, benötigt man ein Ziel. Ein solches ist mit der sicheren Landung des Raumschiffs gegeben. Gleichfalls sind Bedingungen der Aufgabenstellung zu beachten. Zu diesen zählen hier neben den angegebenen Größen auch die Gesetze der Physik. Während der Programmierung müssen wir die Fakten, Abläufe und Zusammenhänge geeignet modellieren.

Programmier-Paradigma : Die Programmiersprache R legt die Erstellung von funktionalen Lösungen nahe. Denn hinter jeder Anweisung stehen letztlich Funktionsaufrufe. Für die Lösung von Teilaufgaben sind Aufgaben zu extrahieren, die aufgrund von Inputs die zugehörigen Outputs erarbeiten. Die Lösung einer Aufgabe kann dann von Funktionen ermittelt werden, die es zu programmieren gilt.

Vorgehensstrategie : Einigkeit besteht in der Empfehlung, schwierige Lösungen in Teillösungen zu zerlegen und die Lösung schwerer Aufgaben durch Zusammenfügung von Teillösungen zu entwickeln. Als Strategien zum Ziel werden oft unterschieden:

- top down
- bottom up
- Mischung dieser

Im ersten Fall versucht man, zunächst das schwierige Problem zu zerlegen, um dann handhabbare, kleinere zu erhalten. Diese Vorgehensweise setzt oft eine große Übersicht und auch Erfahrung voraus. Überlegt man sich erst die Lösung einzelner mit hoher Sicherheit brauchbarer Puzzle-Steine liegt der zweite Fall vor. Hierbei kann man klein anfangen und der großen Lösung entgegenstreben. Möglicherweise ist dieser Ansatz für Anfänger leichter, besitzt jedoch den Nachteil, dass eventuell unpassende Steine entwickelt werden.

Da erst nach Fertigstellung das Ergebnis beurteilt werden kann, lässt sich vorher kaum abschätzen, wie gut die Lösung ausfallen wird. Es hat sich gezeigt, dass sehr oft erst eine Wiederholung der Aufgabenlösung aufgrund der Erfahrungen aus dem ersten Durchgang zu einer guten Lösung führt. Wie auch immer man diese Frage sieht, ist eine Lösung hinterher nur dann wirklich brauchbar,

- wenn sie wie gewünscht funktioniert,
- wenn sie so dokumentiert ist, dass auch Projektfremde diese verstehen können und
- wenn die Lösung mit moderatem Aufwand an etwas veränderte Bedingungen angepasst werden kann.

Literates Programmieren : Um besonders dem Dokumentationswunsch zu entsprechen, bietet sich der literate Programmierstil an, bei dem neben der explizierten Zerlegung von Problemen bzw. der expliziten Zusammenfügung von Teillösungen jeder einzelne Schritt verbal dargelegt wird.

6.3 Eine grobe Struktur zur Lösung

Wir wollen den Versuch wagen, das Spiel mit einer Funktion umzusetzen. In der ersten Entwicklungsstufe sollen große Teilaufgaben extrahiert werden.

Was ist zu tun?

- Nach dem Aufruf der Spiel-Funktion müssen zufällig Parameter für den Himmelkörper und die Raumschiffsituation ermittelt werden.
- Die Abarbeitung der Kapitänsbefehle lässt sich gut im Rahmen einer einfachen Schleife umsetzen.
  - Innerhalb eines Schleifendurchgangs müssen dem Raumschiffchef wichtige Parameter angezeigt werden.
  - Dann muss der Kapitän seine Entscheidung eingeben.
  - Aufgrund des Befehls an die Bremseinheiten muss die Bremsphase umgesetzt werden.
  - Es ist hilfreich, wenn der Kommander die Landung graphisch verfolgen kann.
  - Anschließend ist zu prüfen, ob die Oberfläche während des Bremsmanövers erreicht wird.
- Zum Schluss sollte eine Meldung über Erfolg, Landungsvorgang oder Misserfolg ausgegeben werden.

Es hat sich gezeigt, dass zu Beginn immer noch ein paar Größen eingeführt werden müssen. Deshalb ergänzen wir noch eine Initialisierungssektion und kommen zu folgender groben Struktur:

39 ⟨*definiere mondandung* 39⟩ ≡ ⊂ 61

```
mondlandung <- function(level_of_gamer = c("Beginner", "Normalo", "Experte")[1], zufall=TRUE){
  ⟨checke Argumente 40⟩
  ⟨initialisiere Variablen und Funktionen 44⟩
  ⟨begrüße Spieler und ziehe Landesituation 46⟩
  ⟨beginne Schleife über Befehle des Kapitäns 54⟩
    ⟨zeige Situation an 49⟩
    ⟨frage Brems-Befehl ab 41⟩
    ⟨berechne Auswirkungen des Bremsbefehls 58⟩
    ⟨stelle Verlauf der Landung dar 59⟩
    ⟨beende die Schleife, wenn Oberfläche erreicht 53⟩
  ⟨beende Schleife über Befehle des Kapitäns 55⟩
  ⟨erstelle Abschlussbericht 50⟩
}
```

Das Argument `zufall` stellt einen Schalter dar, mit dem eine Standardsituation erzeugt wird. Mit dem anderen Argument (`level_of_gamer`) kann die Spielerstärke gewählt werden.

Auch wenn an dieser Stelle die einzelnen Einheiten nicht ganz exakt charakterisiert worden sind, sollte der Leser doch Grundidee und Logik der Lösung verstehen können. Aus Sicht der Programmentwicklung sollten die einzelnen Einheiten so sein, dass sie sich isoliert voneinander entwickeln, testen und verbessern lassen.

6.4 Die Umsetzung der einzelnen Schritte

6.4.1 Argumentencheck

Es ist empfehlenswert, die an eine Funktion übergebenen Argumente zu schecken. Dieses wollen wir anhand des Arguments `level_of_gamer`

demonstrieren. Wenn ein Objekt der Länge 0 eingegeben wurde, wird das Beginner-Level eingestellt. Das gilt auch für eine Zeichenkette, die nicht mit den vorgegebenen Möglichkeiten übereinstimmt. Wir wandeln ggf. die Zeichenkette in eine Zahl aus $\{1, 2, 3\}$ um. Andere Zahlen werden durch eine 1 ersetzt, welche das Anfänger-Level repräsentiert.

40 ⟨*checke Argumente* 40⟩ ≡ ⊂ 39

```
if(0 == length(level_of_gamer)) level_of_gamer <- 1
if(is.character(level_of_gamer)) level_of_gamer <-
  c(which(c("Beginner", "Normalo", "Experte") == level_of_gamer),1)[1]
if( !(level_of_gamer %in% (1:3))) level_of_gamer <- 1
```

6.4.2 Bremsbefehleingabe

Beginnen wir mit der Befehlseingabe des Kapitäns. Diese lässt sich durch Aufruf einer passenden Funktion umsetzen:

41 ⟨*frage Brems-Befehl ab* 41⟩ ≡ ⊂ 39

```
if(state["hoehe"] < 100) cat("hoehe < 100  =>  Bremsdauer 0.25 ZE !!")
treibstoffmenge <-
lese_zahl_ein("Welche Treibstoffmenge soll eingesetzt werden?",
              zahlmin = 0, zahlmax = state["tank"])
cat("    ->", treibstoffmenge, "Einheiten Treibstoff an Antrieb!\n")
```

Damit haben wir die Lösung des Einleseproblems verschoben, aber wir sind sicher, dass man eine solche Funktion `lese_zahl_ein` erstellen kann. Da es in vielen Problemkreisen Einleseaufgaben gibt, macht es an dieser Stelle Sinn, eine Funktion zu entwerfen. Bei dem Entwurf von Funktionen muss man sich immer wieder folgende Fragen stellen:

- Was soll die Funktion tun und was muss beachtet werden?
- Was soll herauskommen und was muss die Funktion ausgeben?
- Was ist notwendig, um den Job zu erledigen, welche Informationen müssen der Funktion bereitgestellt werden?
- Welche der Infos sollen als Argumente und welche sonstwie bereitgestellt werden?
- Welche allgeinen Verwendungen sind für die Funktion denkbar?
- Auf welche nützlichen Funktionen kann bei der Umsetzung zurückgegriffen werden?

Unsere Funktion soll nicht nur einen Wert einlesen, der Wert soll auch als Zahl ausgegeben werden und soll sich innerhalb eines Intervalls befinden. Denn es ist zu beachten, dass nicht mehr für das Bremsen eingesetzt werden kann als der Tank hergibt. Anhand dieses Umwandlungsprozesses wird der Leser daran erinnert, dass sich Variablen nach Typen klassifizieren lassen. Zeichenketten sind keine Zahlen oder Kategorien. In R werden zudem Klassen von Objekten unterschieden, was aber an dieser Stelle nicht so offensichtlich ist.

Damit der spätere Spieler weiß, was er zu tun hat, muss ihm mitgeteilt werden, was einzugeben ist. Damit haben wir die Parameter der Funktion hinreichend motiviert und kommen zu folgender einfachen Umsetzung. Wir schlagen zunächst eine ganz einfache Funktion vor:

42 ⟨*definiere* `lese_zahl_ein` 42⟩ ≡ ⊂ 44

```
lese_zahl_ein <- function(mitteilung, zahlmin = 0, zahlmax = Inf ){
  cat(mitteilung, "\n")
  zahl <- readline()
  zahl <- as.numeric(zahl)
  zahl <- max(zahlmin, min(zahl, zahlmax))
  return(zahl)
}
```

Eine Kritik an dieser Funktion besteht darin, dass der Kapitän vor lauter Aufregung eventuell keine korrekte Zahl eingibt. Ein solcher Fall sollte abgefangen werden. Eine Lösung sehen wir in der Funktion, die im Folgenden definiert wird. Ihre Idee besteht darin, die Umwandlung zu *probieren*. Wenn ein Fehler auftritt, wird ein Default-Wert (hier 0) eingesetzen sowie eine Fehlermeldung ausgegeben. Für einen schnellen Ausstieg wurde außerdem – sozusagen als short cut – eingebaut: Wenn ein "X" eingegeben wird, wird eine große negative Bremsmenge abgeliefert, die dann unweigerlich zum Absturz führt.

43 ⟨*definiere* `lese_zahl_ein` 42⟩+ ≡ ⊂ 44

```
lese_zahl_ein <- function(mitteilung, zahlmin = 0, zahlmax = Inf ){
  cat(mitteilung, "\n")
  zahl <- readline()
  cat("\n ->", zahl, "eingelesen \n")
  if(0 < length(grep("X",zahl))) return(-99999)
  zahl <- try(as.numeric(zahl))
  if(class(zahl) == "try-error" || is.na(zahl)){
    cat("FEHLER: leider war die Eingabe unbrauchbar,",
        "es wurde 0 eingesetzt!!!\n")
    zahl <- 0
  }
  zahl <- max(zahlmin, min(zahl, zahlmax))
  return(zahl)
}
```

Es sei noch eine ergänzende Erklärung zu der `try()`-Konstruktion gestattet. Dieser Funktion können R-Anweisungen übergeben werden, die *zunächst nur probiert* werden. Bei Fehlerlosigkeit kann dann das Ergebnis weiter verwendet werden. Falls es jedoch zu einem Fehler kommt, wird ein Output erzeugt, an dem der Fehlerfall abgelesen werden kann: In diesem Fall ist das Ergebnis von der Klasse `try-error`. Diese Eigenschaft wird in der Einlese-Funktion ausgenutzt und mit Hilfe einer `if`-Konstruktion auf die fehlerhafte Eingabe hingewiesen. Weiter wird ein Ersatzwert so eingesetzt, dass der weitere Ablauf reibungslos verläuft.

Die Einlesefunktion können wir im Initialisierungsteil vereinbaren.

44 ⟨*initialisiere Variablen und Funktionen* 44⟩ ≡ ⊂ 39

```
⟨definiere lese_zahl_ein 42⟩
```

Ein weitergehender Vorschlag wäre, fehlerhafte Eingaben erneut neben einer

Warnung-Meldung mit einer erneuten Abfrage zu begegnen. Auch könnte man die Anzahl der Einleseversuche begrenzen und einen Default-Wert einsetzen. Eine Umsetzung dieses Vorschlags findet der Leser in der Funktion `lese_zahl_ein_plus()`, die eine `while`-Schleife für die wiederholten Einlesevorgänge benutzt.

45 ⟨*definiere* `lese_zahl_ein_plus` 45⟩ ≡

```
lese_zahl_ein_plus <- function(mitteilung, zahlmin = 0, zahlmax = Inf,
                               max_versuche = 10, default = 0 ){
  korrekt <- FALSE
  zahl <- NA
  while(korrekt == FALSE || zahlmin <= zahl || zahl <= zahlmax ){
    max_versuche <- max_versuche - 1
    if(max_versuche == 0){
      cat("WARNUNG: maximale Eingabeversuchszahl erreicht, Default-Wert",
          default,"wird eingesetzt\n")
      zahl <- default
      break
    }
    cat(mitteilung, "\n")
    zahl <- readline()
    zahl <- try(as.numeric(zahl))
    if(class(result) == "try-error"){
      cat("FEHLER: leider war die Eingabe unbrauchbar,",
          "bitte noch einmal versuchen!!!\n")
      next
    }
    if( zahl < zahlmin ){
      cat("FEHLER: Eingabe zu klein, bitte noch einmal versuchen!!!\n")
      next
    }
    if( zahlmax < zahl ){
      cat("FEHLER: Eingabe zu klein, bitte noch einmal versuchen!!!\n")
      next
    }
    break
  }
  return(zahl)
}
```

Da nicht gesichert ist, wie lange der Spieler falsche Eingaben tätigt, macht an dieser Stelle eine `while`-Schleife Sinn. In dieser werden verschiedene Situationen, die eintreten können, untersucht und abgehandelt. Bei genauerer Betrachtung fallen uns sicher noch weitere Verbesserungsvorschläge ein. Hieran lässt sich erkennen, dass in der Regel unklar ist, wie weit die Sonderfallabhandlungen und Ausgestaltungen gehen sollen. Oft wird der Kompromiss der Abhandlung dann durch Zeitbegrenzungen bestimmt.

6.4.3 Konstruktion einer Ausgangssituation

Jetzt gilt es noch, den Anfangszustand zu definieren. Es sind die Situationsparameter zu ziehen. Um eine zufällige Situation zu erhalten, setzen wir den Seed per Datum.

Für die Gravitation wird eine ganzzahlige Größe zwischen 10 und 100 gewählt. Die Höhe variiert zwischen 1000 und 3000. Die Anfangsgeschwindigkeit bekommt anfangs einen Wert zwischen 10 und 500. Als letzte spannende Größe bleibt der Treibstoffvorrat. Dieser sollte so sein, dass (meistens) eine erfolgreiche Landung möglich ist, eine solche jedoch nicht zu einfach ist. Deshalb überlegen wir, welche Energie das Raumschiff beim Spielstart besitzt und wählen dann einen entsprechenden Vorrat.

Anfangs besitzt das Raumschiff Lage- und Bewegungsenergie. Die erste Form ist proportional zur Gravitationsbeschleunigung und zur Höhe über der Oberfläche: $E_{pot} = m \cdot g \cdot h$. (Vervierfachen wir den Wert von g, dann halbiert sich die Fallzeit und die Aufprall-Geschwindigkeit verdoppelt sich. Doppelte Geschwindigkeit bedeutet aber eine Vervierfachung der Bewegungsenergie. Damit wächst die Energie linear mit der Gravitationsbeschleunigung.) Zu dieser Lage-Energie müssen wir noch die Energie der Anfangsgeschwindigkeit v_0 addieren. Hierfür gilt mit m als Masse:

$$E_{v_0} = m \cdot g \cdot s_{v_0} = m \cdot g \cdot \frac{g}{2} \cdot t^2 = \frac{mg^2}{2} \cdot \left(\frac{v_0}{g}\right)^2 = \frac{mv_0^2}{2}$$

Damit folgt für die Gesamtenergie zu Beginn der Bremsmanöver:

$$E_0 = m \cdot \left(g \cdot h + \frac{v_0^2}{2}\right)$$

Ausgehend von der Höhe, der Anfangsgeschwindigkeit und der Gravitationsbeschleunigung können wir nun einen geeigneten Vorrat wählen.

Die vier veränderlichen Größen `tank, hoehe, delta.s, v_t` fassen wir in einer Liste zusammen. Das dritte Element soll den jeweils letzten Höhenverlust verwalten. Die Treibstoffmenge wird mit 7.7 multipliziert, um für den Spieler den Zahlenzusammenhang zwischen Treibstoff und Bremswirkung ein wenig zu verschleiern.

Die Spielerstärke nimmt Einfluss auf die Treibstoffmenge.

46 ⟨*begrüße Spieler und ziehe Landesituation* 46⟩ ≡ ⊂ 39

```
cat("\n================================================\n")
cat("\n                   MONDLANDUNG                  \n")
cat("\n  -- ein Spiel fuer angehende Raumschiff-Chefs --  \n")
cat(paste("                 Spielerlevel:", level_of_gamer,"\n"))
cat("\n                   viel Erfolg                  \n")
cat("\n================================================\n")
seed <- date(); seed <- sub(" [0-9]*$","",seed)
seed <- sub("^.* ","",seed); seed <- gsub("[:]","",seed) # ; seed <- 7
set.seed(seed)
GRAVITATION <- 10 *  sample(1:10,1)
hoehe       <- 100*  sample(10:30,1)
v_t         <- 10 *  sample(1:50,1) #; v_t <- 0
energie     <- GRAVITATION * hoehe + v_t^2/2
tank <- round(energie * .004 * runif(1, 1, 1.2))
if(!zufall){
  v_t <- 0; GRAVITATION <- 50; hoehe <- 2000; tank <- 1500
}
delta.s <- 0
factor <- 7.7 * c(3.0, 2.0, 1.0)[level_of_gamer]
state <- c(round(tank * factor), hoehe, delta.s, v_t)
names(state) <- c("tank", "hoehe", "delta.s", "v_t")
```

6.4.4 Situationsanzeige

Dieses Teilproblem lösen wir ganz einfach durch eine Ausgabe der relevanten Größen. Hier könnte eine graphische Darstellung viel Verschönerungen bringen.

47 ⟨*erstelle Ausgabe zur Darstellung der Lage* 47⟩ ≡ ⊂ 48

```
cat("aktuelle Hoehe          :", round(state["hoehe"]),  "\n")
cat("letzter Hoehenverlust   :", round(state["delta.s"]),"\n")
cat("aktuelle Geschwindigkeit:", round(state["v_t"]),    "\n")
cat("aktuelle Treibstoffmenge:", round(state["tank"]),   "\n")
cat("GRAVITATION             :", GRAVITATION,            "\n")
```

Diese Ausgabe könnten wir direkt in unsere Spielfunktion einbauen. Wir wollen jedoch auch hier lieber eine Funktion verwenden. Diese soll `show_state()` heißen und wird im Initialisierungsteil definiert.

48 ⟨*initialisiere Variablen und Funktionen* 44⟩+ ≡ ⊂ 39

```
show_state <- function(){
  ⟨erstelle Ausgabe zur Darstellung der Lage 47⟩
}
```

Zur Anzeige für den Kapitän können wir jetzt diese Funktion verwenden.

49 ⟨*zeige Situation an* 49⟩ ≡ ⊂ 39

```
show_state()
```

6.4.5 Funktionsende

Am Ende der Funktion werden die wichtigen Daten noch einmal angezeigt und über Erfolg bzw. Misserfolg Auskunft gegeben. Natürlich sollte man sich immer fragen, was von der Arbeit innerhalb einer Funktion nach außen gelangen soll. Was soll explizit per `return`, per Druck-Anweisungen oder beispielsweise durch das Schreiben einer Datei nach draußen dringen? Daneben gibt es auch noch weitere Wege über besondere Zuweisungen in spezielle Speicherbereiche zu schreiben. Doch sollte man die direkten Wege solange nicht verlassen, bis überzeugende Gegenargumente aufgetaucht sind.

50 ⟨*erstelle Abschlussbericht* 50⟩ ≡ ⊂ 39
```
print.result()
cat("==== end of game =======\n")
```
In der Funktion `print.result()` fassen wir die Anweisung zur Erstellung der Endausgabe zusammen. In der Tat ist der Einwand berechtigt, dass die Funktion kaum an anderer Stelle einsetzbar ist und der Rumpf der Funktion auch direkt in unsere Spielfunktion hätte eingetragen werden können. Jedoch bleibt durch diese Konstruktion der Handlungsteil unserer Spielfunktion sehr übersichtlich.

51 ⟨*initialisiere Variablen und Funktionen* 44⟩+ ≡ ⊂ 39
```
print.result <- function(){
  cat("================================\n")
  cat("Oberflaeche des Himmelskoerpers erreicht!\n")
  cat("Hier sind die aktuellen Daten:\n")
  show_state()
  if(state["v_t"] <= vmax) {
    cat("Bravo: ordentliche Landung!\n")
  } else {
    cat("Leider gecrasht!!\n")
    lines(seq(0, length(h.hist),length=500), runif(500)*2500,type="l")
  }
}
```
Bislang ist die maximale Landegeschwindigkeit noch nicht gesetzt worden. Das wollen wir nun nachholen.

52 ⟨*initialisiere Variablen und Funktionen* 44⟩+ ≡ ⊂ 39
```
vmax <- 50
```

6.4.6 Abbruch, wenn Oberfläche errreicht

Wenn die Oberfläche erreicht ist, muss der Landeprozess (hoffentlich erfolgreich) als abgeschlossen betrachtet werden.

53 ⟨*beende die Schleife, wenn Oberfläche erreicht* 53⟩ ≡ ⊂ 39
```
if( state["hoehe"] <= 0 ){
  cat("Landung abgeschlossen!\n")
  break
}
```
An dieser Stelle kann darauf hingewiesen werden, dass es möglich ist, die große Schleife aus einer `if`-Konstruktion heraus mit dem Zauberwort `break` zu verlassen.

6.4.7 Schleifenverpackung

Der Beginn und das Ende der großen Schleife sind noch umzusetzen. Wir wählen eine `for`-Schleife mit einer Zählvariablen, damit nicht versehentlich eine unendliche Wiederholung eintreten kann. Die maximale Zahl setzen wir in der Initialisierungssektion.

54 ⟨*beginne Schleife über Befehle des Kapitäns* 54⟩ ≡ ⊂ 39
```
for(befehlsanzahl in 1:max_befehlsanzahl){ # begin of for loop
```

55 ⟨*beende Schleife über Befehle des Kapitäns* 55⟩ ≡ ⊂ 39
```
} # of for loop
```

Hier folgt die Setzung der maximalen Befehlsanzahl.

56 ⟨*initialisiere Variablen und Funktionen* 44⟩+ ≡ ⊂ 39
```
max_befehlsanzahl <- 30
```

6.4.8 Berechnung der Auswirkung eines Bremsmanövers

Dieser Abschnitt modelliert die Auswirkung von Bremsmanövern und stellt somit den interessanten Kern des Spiels dar. Idealisierend nehmen wir an, dass sich während des Anflugs das Gravitationsfeld nicht ändert. Auch gehen wir davon aus, dass die Reduktion der Treibstoffmenge keinen Einfluss auf die Bremswirkung hat.

Für den freien Fall wissen wir, wie sich die Geschwindigkeit bei einer Graviation von g erhöht und welche Strecke zurückgelegt wird:

$$v_t = g \cdot t, \quad s = \frac{g}{2} \cdot t^2 = \frac{g}{2} \cdot \left(\frac{v_t}{g}\right)^2 = \frac{v_t^2}{2g}$$

Eine weitere Zeiteinheit führt zu einer Geschwindigkeitserhöhung von:

$$dv = g \cdot (t+1) - g \cdot t = g \cdot [ZE]$$

und zu einer vergrößerten Strecke von:

$$\begin{aligned} ds = \frac{g}{2} \cdot (t + 1[ZE])^2 - \frac{g}{2} \cdot t^2 &= \frac{g}{2} \cdot (2t \cdot [ZE] + 1 \cdot [ZE]^2) \\ &= \frac{g}{2} \cdot \left(2\frac{v_t}{g} \cdot [ZE] + 1 \cdot [ZE]^2\right) \\ &= v_t \cdot [ZE] + \frac{g}{2} \cdot [ZE]^2 \end{aligned}$$

Die Kürzel $[ZE]$ erinnern an die Zeitheiten, die bei inhaltlichen Interpretationen nicht vergessen werden dürfen.

Was ändert sich, wenn beim freien Fall bereits eine Anfangsgeschwindigkeit v_0 existiert? Dann können wir fiktiv zurückrechnen und eine Fallzeit t_0 eines freien Falls mit Ausgangsgeschwindigkeit 0 ermitteln:

$$t_0 = \frac{v_0}{g}, \quad s_0 = \frac{g}{2} \cdot t_0^2 = \frac{g}{2} \cdot \left(\frac{v_0}{g}\right)^2 = \frac{v_0^2}{2g}$$

Damit folgt im Zeitpunkt t eine Addition der Geschwindigkeiten:

$$v_{t,v_0} = g \cdot (t + t_0) = g \cdot \left(t + \frac{v_0}{g}\right) = g \cdot t + v_0$$

und für die Strecke ergibt sich ebenfalls eine Additionsregel:

$$\begin{aligned} s_{t,v_0} &= \frac{g}{2} \cdot (t + t_0)^2 - \frac{g}{2} \cdot t_0^2 \\ &= \frac{g}{2} \cdot \left(t^2 + 2tt_0\right) = \frac{g}{2} \cdot \left(t^2 + 2t \cdot \frac{v_0}{g}\right) \\ &= \frac{g}{2} \cdot t^2 + tv_0 \end{aligned}$$

Abschließend wollen wir die Wirkung des Abbremsens betrachten. Die Gravitation übt eine beständige Kraft auf alle Massen inklusive des Raumschiffs aus. Gleiches gilt für den Abbremsschub. In Erinnerung an $F = m \cdot a$ ist die Kraft proportional zur Beschleunigung. Wird der Raketenantrieb eingeschaltet, dann wirkt also nicht nur die Anziehungs-, sondern auch die Bremskraft auf den Flugkörper. Wenn beide Kräfte wirken, müssen wir in die Formeln statt g eine Differenz: $g - b$ einsetzen. Wir erhalten, wenn b die Beschleunigung beschreibt, die aus dem Abbremsen resultiert, die Formel

$$s_{t,v_0,b} = \frac{g - b}{2} \cdot t^2 + tv_0$$

sowie

$$v_{t,v_0,b} = (g - b) \cdot t + v_0$$

Der Wert von b muss proportional zu der für das Bremsen eingesetzten Treibstoffmenge in einer Zeiteinheit sein. Damit kommen wir zu folgender Umsetzung, die wieder in einer Funktion eingepackt ist:

57 ⟨*initialisiere Variablen und Funktionen* 44⟩+ ≡ ⊂ 39

```
compute.brake.effect <- function(state, treibstoffmenge, GRAVITATION, ZE=1){
  if(missing(ZE)) ZE <- if(state["hoehe"] < 100) 0.25 else 1
  tank <- state["tank"]; v_t <- state["v_t"]; hoehe <- state["hoehe"]
  rest.vorrat <- round(tank - treibstoffmenge)
  b <- treibstoffmenge / ZE / 7.7   # Umrechnung: treibstoff -> beschleunigung
  delta.s <- ZE * v_t + (GRAVITATION - b) / 2 * ZE^2
  hoehe <- hoehe - delta.s
  v_t <- v_t + (GRAVITATION - b) * ZE
  state[] <- c(rest.vorrat, hoehe, delta.s, v_t)
  return(state)
}
```

Nach ihrer Definition können wir die Berechnungsfunktion verwenden. Es sei bemerkt, dass diese Funktion auch für andere Spiele verwendbar ist, sofern die Syntax und Semantik der Parameter beachtet wird.

58 ⟨*berechne Auswirkungen des Bremsbefehls* 58⟩ ≡ ⊂ 39

```
state <- compute.brake.effect(state, treibstoffmenge, GRAVITATION)
```

6.4.9 Darstellung der Höhenentwicklung

Um den Landeanflug zu verdeutlichen, lässt sich die Entwicklung der Höhe darstellen. Dazu muss anfangs die Variable `h.hist` eingeführt werden. Diese dient als Speicher der Vergangenheitswerte der Flughöhen. Eventuell könnte es interessant sein, auch andere Größen des Verlaufs zu protokollieren. Dann könnten Flugschüler später den Landeprozess noch eingehender studieren.

59 ⟨*stelle Verlauf der Landung dar* 59⟩ ≡ ⊂ 39
```
delta.zeit <- c(delta.zeit, if(state["hoehe"] < 100) 0.25 else 1)
h.hist <- c(h.hist, state["hoehe"])
plot.history()
```

Die Funktion `plot.history()` muss natürlich auch definiert werden.

60 ⟨*initialisiere Variablen und Funktionen* 44⟩+ ≡ ⊂ 39
```
plot.history <- function(){
  zeit <- cumsum(delta.zeit)
  plot(zeit, h.hist, ylim=c(0, max(h.hist)),
       xlab="Zeit", axes=FALSE, pch=2, cex=3)
  points(zeit, h.hist, pch = "|", cex=4)
  axis(1); axis(2)
}
delta.zeit <- h.hist <- NULL
```

6.5 Auf zum Spielen

Wer möchte ein Spielchen wagen?

61 ⟨* 3⟩+ ≡
⟨*definiere mondandung* 39⟩
```
# print(noquote(deparse(mondlandung)))
mondlandung(level_of_gamer = "Beginner", zufall = !FALSE)
```

6.6 Aufgaben

Leicht können wir aus dem vorliegendem Spiel interessante Aufgaben ableiten:

- Schreibe eine Funktion, mit der man je nach Situation sehr gute Bremsmanöver berechnen kann.
- Überlege eine schönere Darstellung des Landeprozesses.
- Je nach Höhe könnte es sehr hilfreich sein, die Länge der Bremsmanöver einzugeben. Mache dazu Vorschläge und setze diese um.
- Natürlich will jeder mehrere Spiele absolvieren. Schreibe ein Programm zur Verwaltung von Spielergebnissen, am besten für mehrere Spieler.
- Greife einen der obigen Erweiterungsvorschläge auf und setze ihn um.
- Schreibe das Spiel um, so dass es ein anlegendes Schiff abbildet.

6.7 Die Spielfunktion im Zusammenhang

```
function (level_of_gamer = c("Beginner", "Normalo", "Experte")[1],
    zufall = TRUE)
{
    if (0 == length(level_of_gamer))
        level_of_gamer <- 1
    if (is.character(level_of_gamer))
        level_of_gamer <- c(which(c("Beginner", "Normalo", "Experte") ==
            level_of_gamer), 1)[1]
    if (!(level_of_gamer %in% (1:3)))
        level_of_gamer <- 1
    lese_zahl_ein <- function(mitteilung, zahlmin = 0, zahlmax = Inf) {
        cat(mitteilung, "\\n")
        zahl <- readline()
        zahl <- as.numeric(zahl)
        zahl <- max(zahlmin, min(zahl, zahlmax))
        return(zahl)
    }
    lese_zahl_ein <- function(mitteilung, zahlmin = 0, zahlmax = Inf) {
        cat(mitteilung, "\\n")
        zahl <- readline()
        cat("\\n ->", zahl, "eingelesen \\n")
        if (0 < length(grep("X", zahl)))
            return(-99999)
        zahl <- try(as.numeric(zahl))
        if (class(zahl) == "try-error" || is.na(zahl)) {
            cat("FEHLER: leider war die Eingabe unbrauchbar,",
                "es wurde 0 eingesetzt!!!\\n")
            zahl <- 0
        }
        zahl <- max(zahlmin, min(zahl, zahlmax))
        return(zahl)
    }
    show_state <- function() {
        cat("aktuelle Hoehe          :", round(state["hoehe"]),
            "\\n")
        cat("letzter Hoehenverlust   :", round(state["delta.s"]),
            "\\n")
        cat("aktuelle Geschwindigkeit:", round(state["v_t"]),
            "\\n")
        cat("aktuelle Treibstoffmenge:", round(state["tank"]),
            "\\n")
        cat("GRAVITATION             :", GRAVITATION, "\\n")
    }
    print.result <- function() {
        cat("================================\\n")
        cat("Oberflaeche des Himmelskoerpers erreicht!\\n")
        cat("Hier sind die aktuellen Daten:\\n")
        show_state()
        if (state["v_t"] <= vmax) {
            cat("Bravo: ordentliche Landung!\\n")
        }
        else {
            cat("Leider gecrasht!!\\n")
            lines(seq(0, length(h.hist), length = 500), runif(500) *
                2500, type = "l")
        }
    }
    vmax <- 50
    max_befehlsanzahl <- 30
    compute.brake.effect <- function(state, treibstoffmenge,
        GRAVITATION, ZE = 1) {
        if (missing(ZE))
            ZE <- if (state["hoehe"] < 100)
                0.25
            else 1
        tank <- state["tank"]
        v_t <- state["v_t"]
        hoehe <- state["hoehe"]
        rest.vorrat <- round(tank - treibstoffmenge)
        b <- treibstoffmenge/ZE/7.7
        delta.s <- ZE * v_t + (GRAVITATION - b)/2 * ZE^2
        hoehe <- hoehe - delta.s
        v_t <- v_t + (GRAVITATION - b) * ZE
        state[] <- c(rest.vorrat, hoehe, delta.s, v_t)
        return(state)
    }
    plot.history <- function() {
        zeit <- cumsum(delta.zeit)
        plot(zeit, h.hist, ylim = c(0, max(h.hist)), xlab = "Zeit",
            axes = FALSE, pch = 2, cex = 3)
        points(zeit, h.hist, pch = "|", cex = 4)
        axis(1)
        axis(2)
    }
    delta.zeit <- h.hist <- NULL
    cat("\\n=================================================\\n")
    cat("\\n                   MONDLANDUNG                   \\n")
    cat("\\n  -- ein Spiel fuer angehende Raumschiff-Chefs --  \\n")
    cat(paste("                  Spielerlevel:", level_of_gamer,
        "\\n"))
    cat("\\n                    viel Erfolg                  \\n")
    cat("\\n=================================================\\n")
    seed <- date()
    seed <- sub(" [0-9]*$", "", seed)
    seed <- sub("^.* ", "", seed)
    seed <- gsub("[:]", "", seed)
```

```
 [97]     set.seed(seed)
 [98]     GRAVITATION <- 10 * sample(1:10, 1)
 [99]     hoehe <- 100 * sample(10:30, 1)
[100]     v_t <- 10 * sample(1:50, 1)
[101]     energie <- GRAVITATION * hoehe + v_t^2/2
[102]     tank <- round(energie * 0.004 * runif(1, 1, 1.2))
[103]     if (!zufall) {
[104]         v_t <- 0
[105]         GRAVITATION <- 50
[106]         hoehe <- 2000
[107]         tank <- 1500
[108]     }
[109]     delta.s <- 0
[110]     factor <- 7.7 * c(3, 2, 1)[level_of_gamer]
[111]     state <- c(round(tank * factor), hoehe, delta.s, v_t)
[112]     names(state) <- c("tank", "hoehe", "delta.s", "v_t")
[113]     for (befehlsanzahl in 1:max_befehlsanzahl) {
[114]         show_state()
[115]         if (state["hoehe"] < 100)
[116]             cat("hoehe < 100  =>  Bremsdauer 0.25 ZE !!")
[117]         treibstoffmenge <- lese_zahl_ein("Welche Treibstoffmenge soll eingesetzt werden?",
[118]             zahlmin = 0, zahlmax = state["tank"])
[119]         cat("    ->", treibstoffmenge, "Einheiten Treibstoff an Antrieb!\\n")
[120]         state <- compute.brake.effect(state, treibstoffmenge,
[121]             GRAVITATION)
[122]         delta.zeit <- c(delta.zeit, if (state["hoehe"] < 100) 0.25 else 1)
[123]         h.hist <- c(h.hist, state["hoehe"])
[124]         plot.history()
[125]         if (state["hoehe"] <= 0) {
[126]             cat("Landung abgeschlossen!\\n")
[127]             break
[128]         }
[129]     }
[130]     print.result()
[131]     cat("==== end of game =======\\n")
[132] }
```

7 Aufgabe: Erfinde ein Mondlandesimulationsspiel

Jetzt ist der geneigte Leser wieder am Zuge. Und er bekommt hiermit die Aufgaben,

- das Kapitel *Mondlandung* zu studieren,
- die Darstellung zu verstehen und Verbesserungen umzusetzen
- und als Krönung die Entwurfsgedanken aus der Helikopter-Simulation zu einem neuen Simulationsspiel in die Mondlandung aufzunehmen.

Auf geht es! Viel Spaß dabei!

8 Anhang

Code Chunk Index

⟨ * 3 ∪ 5 ∪ 6 ∪ 7 ∪ 9 ∪ 10 ∪ 11 ∪ 12 ∪ 13 ∪ 34 ∪ 36 ∪ 38 ∪ 61 ∪ 62 ∪ 63 ∪ 64⟩ p4
⟨*beende die Schleife, wenn Oberfläche erreicht* 53⟩ ⊂ 39 ..... p41
⟨*beende Schleife über Befehle des Kapitäns* 55⟩ ⊂ 39 ..... p42
⟨*beginne Schleife über Befehle des Kapitäns* 54⟩ ⊂ 39 ..... p42
⟨*begrüße Spieler und ziehe Landesituation* 46⟩ ⊂ 39 ..... p40
⟨*berechne Auswirkungen des Bremsbefehls* 58⟩ ⊂ 39 ..... p43
⟨*berechne Strecke aufgrund der Fallzeit* `zeit` 2⟩ ⊂ 3, 5, 6, 7, 8 ..... p4
⟨*checke Argumente* 40⟩ ⊂ 39 ..... p36
⟨*definiere Hubschrauber-Anzeige* 20⟩ ⊂ 21, 22, 29, 30, 34 ..... p19
⟨*definiere mondandung* 39⟩ ⊂ 61 ..... p35
⟨*definiere* `get.number()` 23 ∪ 24 ∪ 25 ∪ 26 ∪ 27 ∪ 28⟩ ⊂ 29, 30, 34 ..... p21
⟨*definiere* `lese_zahl_ein` 42 ∪ 43⟩ ⊂ 44 ..... p37
⟨*definiere* `lese_zahl_ein_plus` 45⟩ p38
⟨*definiere* `start.window()` 33⟩ ⊂ 34 ..... p26
⟨*definiere* `what.to.do()` 32⟩ ⊂ 34 ..... p26
⟨*entwerfe Struktur für eine Oberflächenlösung* 31⟩ p25
⟨*erstelle Abschlussbericht* 50⟩ ⊂ 39 ..... p41
⟨*erstelle Ausgabe zur Darstellung der Lage* 47⟩ ⊂ 48 ..... p40
⟨*erstelle Normalgebirge* 14⟩ ⊂ 15, 16 ..... p13
⟨*frage Brems-Befehl ab* 41⟩ ⊂ 39 ..... p36
⟨*initialisiere Variablen und Funktionen* 44 ∪ 48 ∪ 51 ∪ 52 ∪ 56 ∪ 57 ∪ 60⟩ ⊂ 39 p37
⟨*lasse Anwender Hubschrauber zu einem Ziel fliegen* 30⟩ p24
⟨*lasse Hubschrauber fliegen* 22⟩ p20
⟨*lasse Hubschrauber gemäß Anwender fliegen* 29⟩ p23
⟨*setze Anfangshöhe* 4⟩ ⊂ 5, 6, 7, 8 ..... p5
⟨*setze* `zeit` 1⟩ ⊂ 3, 5, 6, 7, 8 ..... p4
⟨*stelle Verlauf der Landung dar* 59⟩ ⊂ 39 ..... p44
⟨*steuere Hubschrauber* 37⟩ p29
⟨*teste Hubschrauber* 21⟩ p19
⟨*zeichne Bergspitze* 18⟩ ⊂ 19 ..... p17
⟨*zeichne Normalgebirge* 15⟩ p14
⟨*zeichne Pfad ein* 17⟩ p16
⟨*zeichne Pfad zur Bergspitze* 19⟩ p18
⟨*zeige Flug mit Schweif* 8⟩ ⊂ 9 ..... p8
⟨*zeige Helikoper in Landschaft* 35⟩ ⊂ 36, 37 ..... p28
⟨*zeige Landschaft* 16⟩ ⊂ 17, 18, 35 ..... p15
⟨*zeige Situation an* 49⟩ ⊂ 39 ..... p40

Object Index

anfangshoege ∈ 4
anz ∈ 20
arc ∈ 20
co ∈ 13
compute.brake.effect ∈ 57, 58
compute.flight ∈ 10, 11, 12, 13
control.heli ∈ 37, 38
coor ∈ 13
delta ∈ 37, 38
delta.s ∈ 46, 47, 57
delta.zeit ∈ 59, 60
dir ∈ 22, 29, 30, 32, 34
dir.change ∈ 29, 30, 32, 34
dir.new ∈ 29, 30, 32
do.nothing ∈ 33, 37, 38
energie ∈ 46
factor ∈ 46
fly ∈ 35, 36, 37, 38
fly.default ∈ 35
fly.helicopter ∈ 35
get.number ∈ 23, 24, 25, 26, 27, 28, 29, 30, 34
GRAVITATION ∈ 46, 47, 57, 58
h.hist ∈ 51, 59, 60
hill ∈ 14, 15, 16
hoehe ∈ 18, 41, 46, 47, 53, 57, 59
idx ∈ 7, 8, 18
korrekt ∈ 45
lambda ∈ 16
landschaft ∈ 16, 17, 18, 19, 35, 37
lese_zahl_ein ∈ 41, 42, 43, 44, 45
lese_zahl_ein_plus ∈ 45
Lot.punkt ∈ 35
max_befehlsanzahl ∈ 54, 56
max_versuche ∈ 45
mondlandung ∈ 39, 61
nobreak ∈ 64
no.spalte ∈ 18, 19
no.zeile ∈ 18, 19
n.pfad ∈ 17
n.schweif ∈ 7, 8
n.steps ∈ 19
number ∈ 24, 25, 26, 27, 28
plot.history ∈ 59, 60
print.result ∈ 50, 51
res ∈ 15, 16, 17, 18, 19, 35, 37
rest.vorrat ∈ 57
s.bewegung ∈ 10
schlafzeit ∈ 7, 8
seed ∈ 46
s.fall ∈ 10
show.helicopter ∈ 20, 21, 22, 29, 30, 32, 34
show.heli.state ∈ 37, 38
show_state ∈ 48, 49, 51
simple.dokuwiki.to.rev ∈ 64
start.window ∈ 31, 33, 34
state ∈ 41, 46, 47, 48, 49, 51, 53, 57, 58, 59
step ∈ 29, 30, 32, 33, 34
tank ∈ 41, 46, 47, 57
treibstoffmenge ∈ 41, 57, 58
tworkwin ∈ 64
txt ∈ 64

txy ∈ 17, 18, 19
txy1 ∈ 17
txy2 ∈ 17
txyz ∈ 35
unebenheiten ∈ 16
usr ∈ 20
v.begin ∈ 12, 13
vmax ∈ 51, 52
v_t ∈ 46, 47, 51, 57
what.to.do ∈ 31, 32, 33, 34, 37, 38
winkel ∈ 10, 12, 13
winkel.2.pi ∈ 10
worktext ∈ 64
x0 ∈ 10
x.del ∈ 10
x.delta ∈ 20
xn ∈ 19
x.new ∈ 29, 30, 32
xy ∈ 12
xy.expand ∈ 16
xyz ∈ 35, 36, 37, 38
xyz.new ∈ 37
y0 ∈ 10
y.del ∈ 10
y.delta ∈ 20
y.end ∈ 30, 34
y.new ∈ 29, 30, 32
z0 ∈ 9, 64
zahl ∈ 41, 42, 43, 44, 45
zeit ∈ 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 60
zn ∈ 19
z.set ∈ 7