R-Package zur Verarbeitung von Pixel-Umgebungen

Bakkarbeit, Tobias Krause

April 2018

In dieser Bakkarbeit wird eine Reihe von Funktionen vorgestellt, welche das Verarbeiten und Manipulieren von Pixel-Umgebungen vereinfacht und effizienter gestaltet.

Contents

1 Abstrakt

Diese Arbeit beschäftigt sich mit der Aufgabe, das Anwenden von Funktionen auf Pixel-Umgebungen zu vereinfachen und effizienter zu gestalten. Solche Anwendungen finden in vielen Bereichen Verwendung, zum Beispiel bei Bildverarbeitung, Pixel-Spielen, sowie in etlichen Simulationen. Ein Problem stellt auf die aufwendige Implementierung von flexiblen und effizienten Lösungen dar. Die naheliegendste Lösung, mit einer Doppelt-For Schleife durch alle Elemente zu iterieren ist höchst ineffizient und unflexibel und alternative Lösungen sind aufwendig zu implementieren. Außerdem gibt es derzeit auf CRAN kein Package, welches sich mit dieser Problemstellung auseinandersetzt. Mit diesem Problem beschäftigt sich diese Arbeit und stellt eine effiziente und flexible Lösung mithilfe von einer Reihe von Funktionen vor.

Definitionen: Ein Pixel ist hier definiert als ein Element in einer 2-Dimensionalen Matrix. Die Pixel-Umgebung beschreibt umliegende Elemente eines Pixels. Welche umliegenden Elemente eine Nachbarschaft inkludiert ist abhängig von der gewählten Nachbarschaft. Einige bekannte Nachbarschaften sind zum Beispiel Moore-, Von-Neumann- und Diamantförmige-Nachbarschaften. (Mehr dazu in 4.2).

Lösungsansatz: In dieser Arbeit wird ddie neue Funktion nhapply() vorgestellt, welche eine Funktion auf Pixel-Umgebungen anwendet und der apply-Familie (sapply, tapply, lapply,...) in der Anwendung sehr ähnelt. Auch bei der hier eingeführten Funktion nhapply() wird zuerst die Matrix eingegeben und als zweiter

Parameter die gewünschte Funktion. Nun wird die Funktion nicht wie bei apply Spalten oder Zeilenweise auf die Matrix angewendet, sondern auf jede Pixel-Umgebung. Der Folgezustand jedes Pixels ist der Ausgabewert der Funktion auf jeder Nachbarschaft.

nhapply() unterstützt beim Programmieren von zellulären Automaten, Pixelspielen, Simulationen und weiteren Lösungen, welche Vorteile aus dem effizienten und flexiblen Anwenden von Funktionen auf Nachbarschaften ziehen können.

Im Ordner "Krause_Bakkarabeit" findet man folgende Dokumente:

- **nhapply:** Dies ist das Package, welches auf CRAN submitted wird. Im Ordner ist der Ordner R zu finden, in dem die R-Codes liegen.
- **Grafiken:** Hier findet man Code, der die Grafiken der Arbeit generiert.
- **GameOfLife, Zell_Auto und GaussFilter:** Dies sind Codes für Beispielanwendungen.
- **simple_apply:** Dieser Code enthält eine einfache Schleifen Lösung zum Performenz-Vergleich mit nhapply.
- **Performenz:** Indem Dokument findet man den Code für das Testen der Effizienz.
- **Durer.jpg und duck.jpg:** Sind Bilder die für Effizienzvergleich und GaussFilter verwendet wurden.

Ein Beispiel, bei dem jeder Pixel die Summe der umliegenden Pixel annimmt:

```
source("nhapply_manipulation.R")
source("nhapply_functions.R")
source("nhapply_master.R")
source("error_handling.R")

mat <- matrix(c(1, 1, 1, 1,
                2, 2, 2, 2,
                3, 3, 3, 3,
                4, 4, 4, 4), ncol=4)

mat_new <- nhapply(mat, sum)

print(mat_new)

##      [,1] [,2] [,3] [,4]
## [1,]   20   16   24   20
## [2,]   20   16   24   20
## [3,]   20   16   24   20
## [4,]   20   16   24   20
```

Berechnung des Elements (2, 2): Summe der umliegenden Elemente genommen: $1 + 2 + 3 + 1 + 2 + 3 + 1 + 2 + 3 = 18$

2 Problemstellung

Der User soll bei passenden Anwendungen (siehe späterer Absatz) einen klaren Mehrwert haben, wenn er mit dem Paket nhapply arbeitet. Pixel-Nachbarschaften zu definieren und auf jede dieser Funktionen anzuwenden ist ein relativ aufwendiger Prozess, sowohl aus der Sicht der Programmierung, als auch der Rechenleistung. Der naheliegende Ansatz, Schleifen für jeden Pixel zu verwenden ist höchst ineffizient und verschiedene Nachbarschaften zu definieren aufwendig.

Diese Probleme lösen die Funktionen des Pakets nhapply. Die drei Grundsätze, nachdem die Funktionen programmiert wurden, sind:

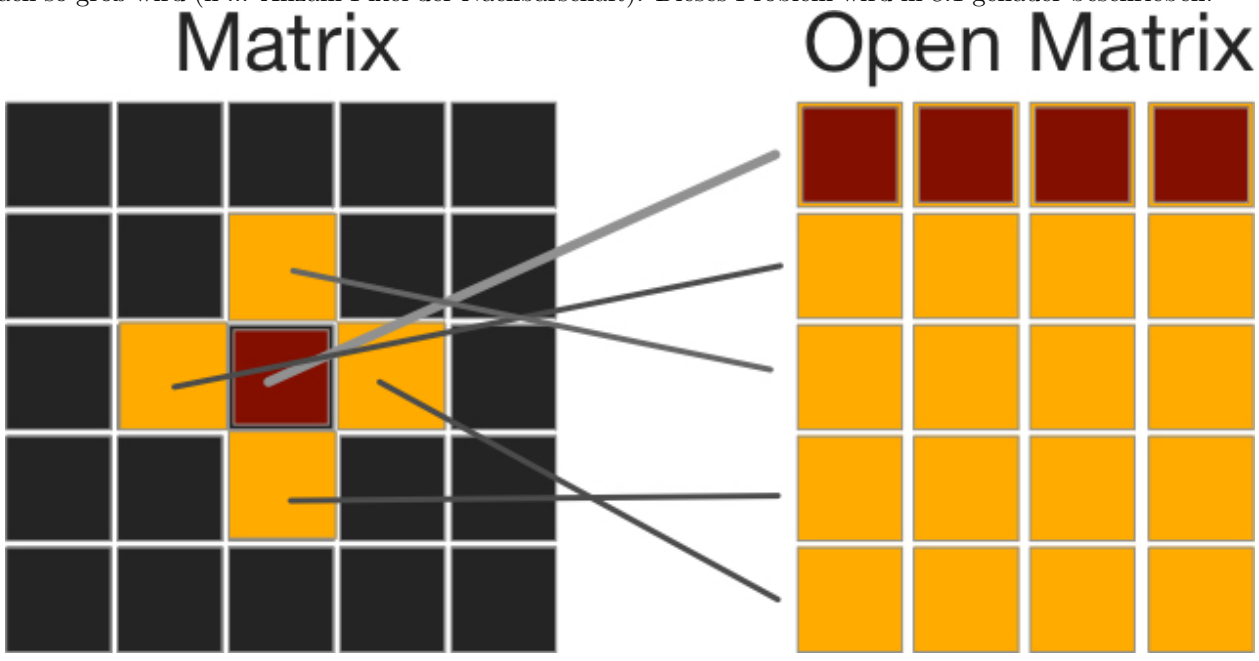
- **Einfachheit:** Die Funktionen sollen ebenso einfach anzuwenden sein, wie die apply-Familie im Base-Package (z.B.: sapply), mit welchen folgender Code ausreicht, um ein gewünschtes Ergebniss zu erhalten: sapply(df, mean)!
- **Flexibilität:** Es sollen benutzerdefinierte Funktionen zusätzlich zu den Base-Funktionen zugelassen werden (wie bei apply) und darüber hinaus sollen Parameter, wie die Nachbarschaft, Übergänge, etc. einstellbar sein.
- **Effizienz:** Die Effizienz soll durch das Benutzen von vektorwertigen Funktionen, welche in R sehr effizient sind, Rechenzeit bestmöglich optimieren. Bekannte Funktionen, wie `count()`, `sum()`, `max()` sollen mit eigenen Funktionen beschleunigt werden. Um die Effizienz zu messen, vergleiche ich die Lösung mit einer einfachen, relativ effizienten Doppel-For-Schleifen Lösung.

Im Kapitel "Erfüllung der Problemstellung" werde ich auf alle drei Punkte und die implementierten Lösungen eingehen.

3 Lösungsansatz

3.1 Die Idee: Erstellen einer "Open-Matrix"

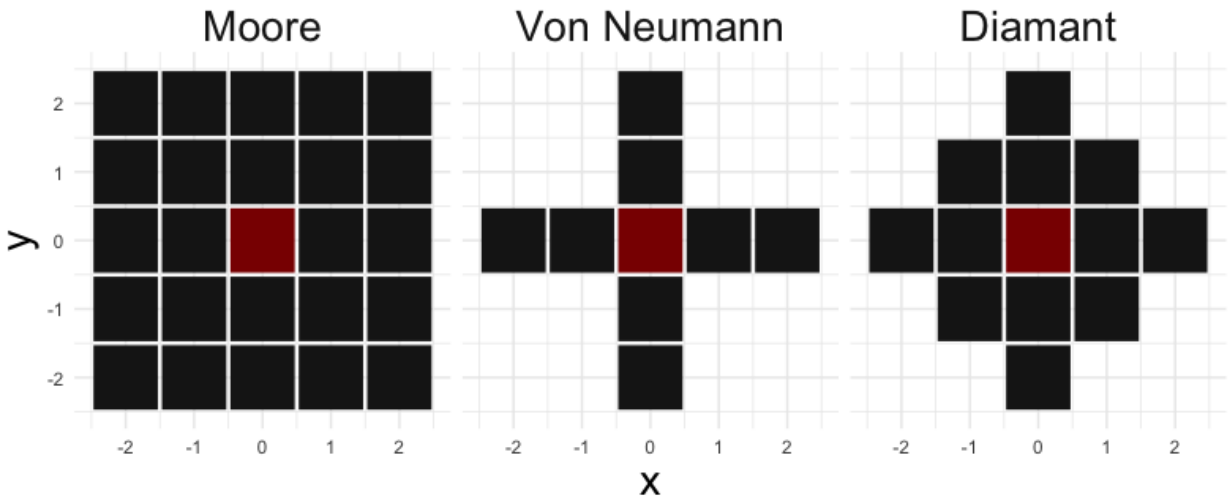
Um die Problemstellung bestmöglich zu lösen, muss die Input Matrix in ein neues Format transformiert werden, sodass Vektorwertige Lösungen möglich werden. Die transformierte Matrix nenne ich **"Open-Matrix"** und schaut wie folgt aus: Jede **Spalte entspricht einem Pixel** der Input Matrix, und jede **Zeile einem Pixel** der Umgebung. Diese Form vereinfacht das Anwenden von Funktionen und erlaubt effiziente Lösungen durch vektorwertige Funktionen. Eine Herausforderung hierbei ist, dass dabei der benötigte Speicherplatz n-fach so groß wird (n ... Anzahl Pixel der Nachbarschaft). Dieses Problem wird in 8.1 genauer beschrieben.



3.2 Betrachtung der einzelnen Schritte

Folgend sind die notwendigen Schritte gelistet, um eine Eingabe zu transformieren, dann zu verarbeiten und letztendlich eine verarbeitete Matrix auszugeben. All diese Schritte sind in einzelnen Funktionen umgesetzt und in der Masterfunktion nhapply() vereint.

1. **Definieren der Nachbarschaft:** Zuerst wird definiert, welche Nachbarschaft von der Funktion betrachtet werden sollen. Implementierte Nachbarschaften sind die bekannten Moore Nachbarschaft, Von-Neumann Nachbarschaft und eine Diamantförmige Nachbarschaft. Es ist auch zulässig, eine eigene Nachbarschaft zu definieren.



   Die Nachbarschaft muss als Liste mit x, y Koordinaten gegeben sein, welche angeben, welcher Nachbar-Pixel relativ zu jedem Pixel zur Nachbarschaft zählt. Funktionen zu dieser Manipulation sind generate_coords() und generate_coords_default() im Dokument nhapply_manipulation.R.

2. **Öffnen der Matrix:** Um die Matrix in die Form der im vorigem Abschnitt beschriebenen "Open-Matrix" zu bringen, verwendet der Code "Lags". Diese Beschreiben, wie die Matrix verschoben werden muss, damit jeweils der gewünschte Nachbar-Pixel genau über dem Pixel liegt. Wenn z.B.: der Pixel links vom mittleren Pixel zur Nachbarschaft angehört, muss die Matrix um eine Spalte links erweitert werden und erzeugt daher einen Lag. Letztendlich sind alle Parameter bestimmt um die Matrix zu öffnen. Dies wird einfach durch rbind(c(matrix), c(matrix[lags], ...) erreicht.

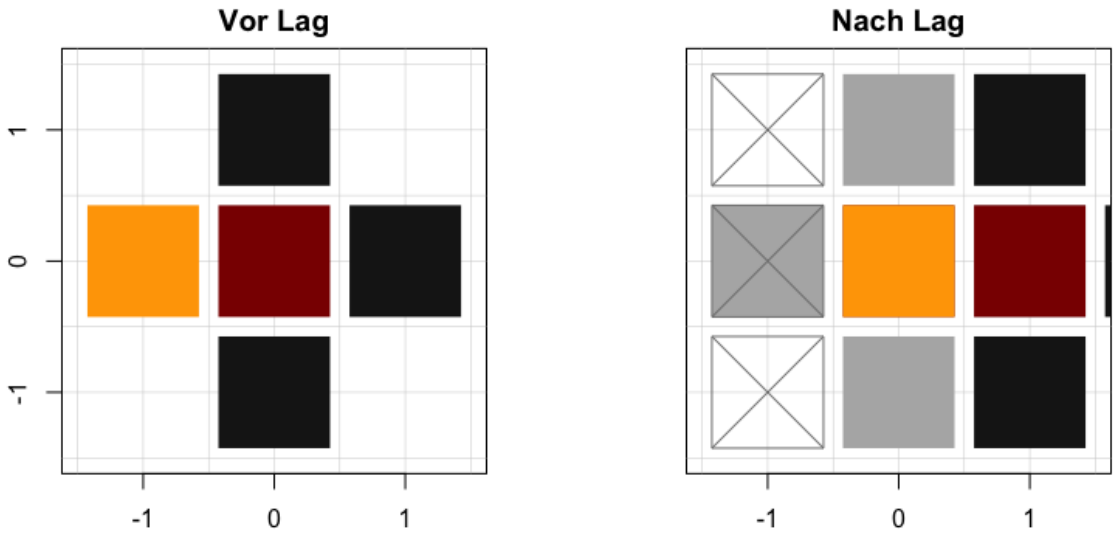
   In der Generate Lag Funktion werden ebenfalls Überläufe geregelt. Sind Überläufe deaktiviert, werden NAs anstelle der eigentlichen Pixel-Werte eingefügt, während wenn Überläufe aktiviert sind, werden Pixel von der anderen Seite in die Funktion inkludiert!

   Zum Beispiel schauen die Lags einer Moore Nachbarschaft mit Breite 1 auf einer 3x3 Matrix wie folgt aus:

```
generate_lags(generate_coords_default("VonNeumann"), 3, 3)

## $rows
## $rows[[1]]
## [1] 1 2 3
##
## $rows[[2]]
## [1] 3 1 2
##
## $rows[[3]]
## [1] 2 3 1
##
## $rows[[4]]
## [1] 1 2 3
```

```
##
##
## $cols
## $cols[[1]]
## [1] 3 1 2
##
## $cols[[2]]
## [1] 1 2 3
##
## $cols[[3]]
## [1] 1 2 3
##
## $cols[[4]]
## [1] 2 3 1
```

Die Open-Matrix hat letztendlich 5 Zeilen (Größe der Nachbarschaft) und 9 Spalten (Anzahl der Pixel).



Lag.png: Auf der Grafik ist erkennbar, wie durch geeigneten Lag, der linke Nachbarpixel durch Hinzufügen einer leeren Spalte, über den mittleren geschoben wird.

Zugehörige Funktionen sind generate_lags() und open_matrix() im Dokument nhapply_manipulation.R.

3. **Verarbeiten der Daten:** Nun können auf die geöffnete Matrix Spaltenweise Funktionen angewendet werden, zum einen bereits vorhandene, für Spalten optimierte Funktionen wie colSums() und colMeans(), weiters die in diesem Paket optimierten Funktionen min, max, maxcount, mincount, which.max und which.min, als auch benutzerdefinierte Funktionen mit apply(). Der Output ist ein Vektor, dessen Länge ident mit der Anzahl der Pixel der Input Matrix ist.

   All diese Funktionen werden von der überstehenden Funktionen mat_apply() verwaltet, welche erkennt ob eine benutzerdefinierte oder optimierte Funktion vorliegt. Zugehörige Funktionen sind extreme(), count_(), w_sums() und mat_apply() im Dokument nhapply_functions.R.

4. **Transformation zu Matrix:** Der resultierende Output Vektor wird nun wieder zurück zu einer Matrix derselben Dimension der Input Matrix transformiert. Dies ist in R ein einfacher, ein-zeiliger Code im Dokument nhapply_functions.R unter Punkt 5.

4 Beschreibung aller Funktionen

4.1 generate_coords() [_manipulation]

Parameter: (mat)

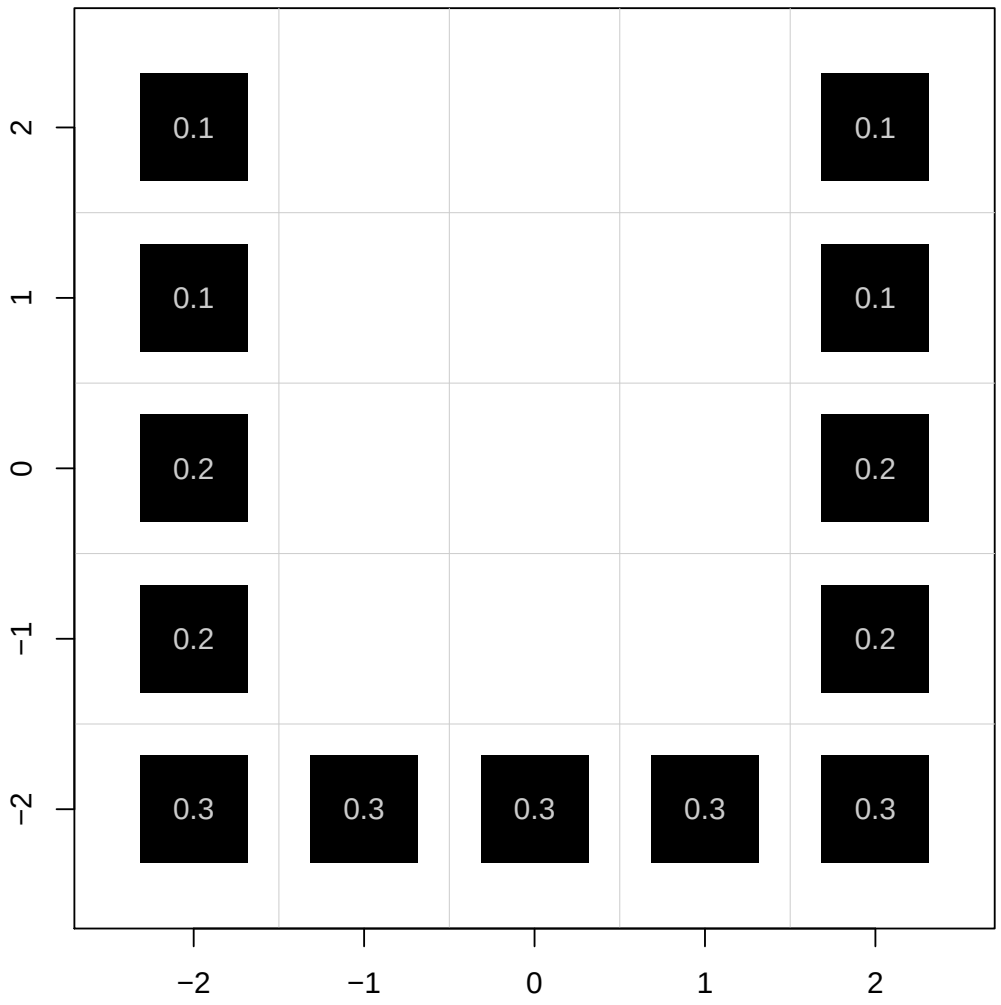
Die Funktion wandelt einer 2-Dimensionale Matrix in eine, für die anderen Funktionen verständliche, verständliche Koordinaten-Matrix um.

Der Input ist hierbei eine nxn Matrix, wobei n ungerade sein muss, und jene Felder, die zur Nachbarschaft gehören sollen, müssen einen numerischen Werte ungleich null besitzen. Diese Werte werden ebenfalls als Gewichte interpretiert! Die Ausgabe ist eine Matrix mit 2 Spalten, welche die x und y Koordinaten der Nachbarspixel relative zum mittleren Pixel ausgeben!

- **Input Parameter:**
  - **mat:** Eine 2-Dimensionale Matrix mit ungeraden Anzahl an Spalten und Zeilen, da der mittlere Punkt den jeweiligen Pixel beschreibt. Felder mit Werten ungleich 0 werden in die Nachbarschaft aufgenommen, jene gleich 0, nicht.
- **Output:** Die Ausgabe besteht aus einer Matrix mit 2 Spalten, welche die x und y Koordinaten der Nachbarn ausgibt.
- **Beispiel:**

  Zuerst werden Koordinaten von einer benutzerdefinierten Nachbarschaft mit verschiedenen Gewichten generiert, daraufhin wird der Input zu Charakters geändert, welches nicht erlaubt ist!

```
# Erstellen einer U-förmigen Matrix
U <-  rbind(c(.1, 0, 0, 0, .1),
                c(.1, 0, 0, 0, .1),
                c(.2, 0, 0, 0, .2),
                c(.2, 0, 0, 0, .2),
                c(.3, .3, .3, .3, .3))
U_cords <- generate_coords(U)
c <- U_cords$coords

## Plotten als Proof of concept
par(pty = "s")
plot(c[, 1], c[, 2], pch = 15, cex = 8,
     xlim = c(-2.5, 2.5), ylim = c(-2.5, 2.5), xaxt = "n",
     yaxt = "n", xlab = "", ylab = "")
abline(h = -1.5:1.5, v = -1.5:1.5, col = "grey80", lwd = .5)
text(c[, 1], c[, 2], U_cords$weights, col = "grey80")
axis(1, at = -2:2)
axis(2, at = -2:2)
```




```
## Input nun als Character
U <-  rbind(c("A", 0, 0, 0, "A"),
                c("A", 0, 0, 0, "A"),
                c("B", 0, 0, 0, "B"),
                c("B", 0, 0, 0, "B"),
                c("C", "C", "C", "C", "C"))

# Wird erfolgreich abgefangen
U_cords <- generate_coords(U)

## Error in generate_coords(U): Input matrix must be numeric and and have odd number of rows
and
## columns
```

- **Error-Handling:**
  - **Stop10:** Tritt ein, falls die Input Matrix nicht ungerade Zeilen oder Spalten aufweist.

4.2 generate_coords_default() [_manipulation]

Parameter: (neighb_type = 1, width = 1, include.own = FALSE)

Die Funktion erzeugt automatisch folgende, gut bekannte Nachbarschaften: 1 Von-Neumann, 2 Moore und 3 Diamantförmig. (https://www.researchgate.net/figure/Common-representations-of-CA-models-and-fi-rst-and-second-order-neighborhood_fig1_258628072)

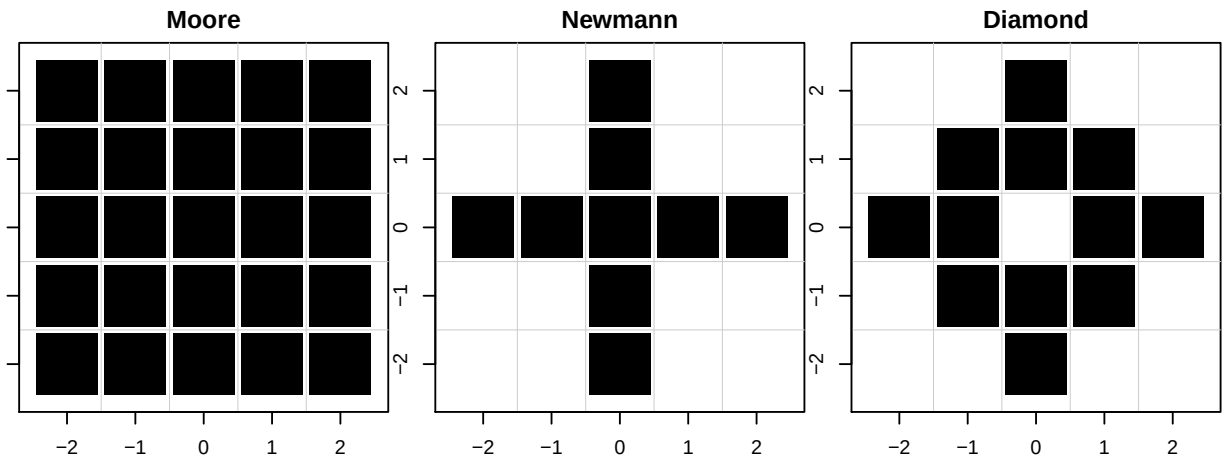
- **Input Parameter:**
  - **neighb_type:** Input kann eine der Zahlen 1, 2, oder 3 sein, als auch dazugehörig ein String namens 1:"Moore", 2:"VonNeumann" oder 3:"Diamond" sein.
  - **width:** Beschreibt die Breite in alle vier Richtungen des Mittelpunkts. So geht eine Von-Neumann Nachbarschaft mit width = 2 bis zwei Pixel nach links, oben, rechts und unten von der Mitte und besitzt daher eine 5x5 Fläche.
  - **include.own:** Besagt, ob der mittlere Punkt selbst zur Nachbarschaft angehören soll.
- **Output:** Die Ausgabe, wie bei generate_coords(), besteht aus einer Matrix mit 2 Spalten, welche die x und y Koordinaten der Nachbarn ausgibt.
- **Beispiel:**

Hier werden alle Defaults gerendert und anschließend fehlerhafte Inputs getestet!

```
moo <- generate_coords_default("moore", width=2, TRUE)
new <- generate_coords_default(2, width=2, TRUE)
dia <- generate_coords_default("Diamond", width=2, FALSE)

print("Moore Koordinaten")

## [1] "Moore Koordinaten"

print(moo)

##       [,1] [,2]
##  [1,]   -2    2
##  [2,]   -2    1
##  [3,]   -2    0
##  [4,]   -2   -1
##  [5,]   -2   -2
##  [6,]   -1    2
##  [7,]   -1    1
##  [8,]   -1    0
##  [9,]   -1   -1
## [10,]   -1   -2
## [11,]    0    2
## [12,]    0    1
## [13,]    0    0
## [14,]    0   -1
## [15,]    0   -2
## [16,]    1    2
```

```
## [17,]    1    1
## [18,]    1    0
## [19,]    1   -1
## [20,]    1   -2
## [21,]    2    2
## [22,]    2    1
## [23,]    2    0
## [24,]    2   -1
## [25,]    2   -2
```

```
par(pty = "s", mfrow = c(1,3), mar=c(3, 1, 2, 1))
ptn <- function(mat, tit){
  plot(mat[, 1], mat[, 2], pch = 15, cex = 7,
       xlim = c(-2.5, 2.5), ylim = c(-2.5, 2.5), xaxt = "n", yaxt = "n",
       xlab = "", ylab = "", main = tit)
  abline(h = -1.5:1.5, v = -1.5:1.5, col = "grey80", lwd = .5)
  axis(1, at = -2:2)
  axis(2, at = -2:2)
}
ptn(moo, "Moore")
ptn(new, "Newmann")
ptn(dia, "Diamond")
```




```
# Falscher Input
generate_coords_default("Diamond", width="2", FALSE)

## Error in 2 * width:  non-numeric argument to binary operator

generate_coords_default(4, width=2, FALSE)

## Error in generate_coords_default(4, width = 2, FALSE): The inputed default neighbourhood
is not known.  Please choose one
## of the following 3:
## 1 - 'Moore'
## 2 - 'VonNeumann'
## 3 - 'Diamond'

generate_coords_default("Diamont", width=2, FALSE)
```

```
## Error in generate_coords_default("Diamont", width = 2, FALSE): The inputed default neighbourhood
is not known.  Please choose one
## of the following 3:
## 1 - 'Moore'
## 2 - 'VonNeumann'
## 3 - 'Diamond'
```

- **Error-Handling:**
  - **stop7:** Eingegebener Nachbarschaftstyp, neighb_type, ist unbekannt.

4.3 generate_lags() [_manipulation]

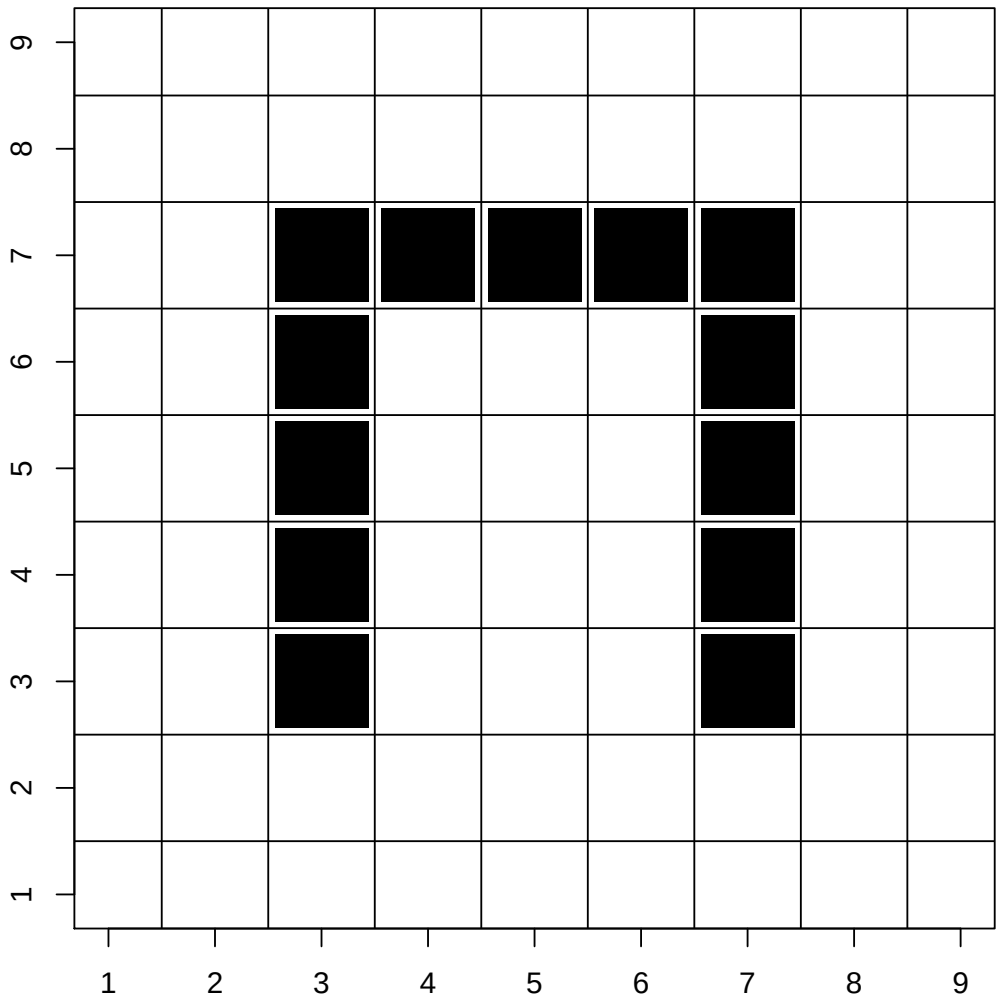
Parameter: (neighb, rows, cols, transition = TRUE)

Die Funktion transformiert Nachbarschaftskoordinaten in eine Liste, welche die Matrix-Lags ausgibt, um Nachbar Pixel über den mittleren Pixel zu legen. Die Ausgabe wird in der Funktion open_matrix() verwendet, um die Matrix zu öffnen. *Anm.: Input kann auch ein Dataframe sein, dieser Fall wird jedoch nicht näher betrachtet.*

- **Input Parameter:**
  - **neighb:** Dieser Input ist eine Matrix (oder Dataframe) mit 2 Spalten, wie von generate_coords() generiert, welche die Nachbar-Pixel in x und y Koordinaten relativ vom mittleren Pixel ausdrückt.
  - **rows und cols:** Diese Parameter geben die Anzahl der Zeilen und Spalten der Matrix an, auf welche die Funktion angewendet werden soll.
  - **transition:** Besagt, ob die Nachbarschaft, wenn ein Ende der Matrix erreicht wird, auf der anderen Seite fortfließt oder einfach keine Nachbar-Pixel außerhalb der Matrix verwendet werden sollen.
- **Output:** Die Ausgabe ist eine Liste mit jeweils zwei Elementen: Die Verschiebungen der Zeilen (lags$rows) und Verschiebungen der Spalten(lags$cols).
- **Beispiel:**

  Hier wird die vorhergegangene U-Matrix (aus. 4.1) in Lags umgewandelt. Dabei wird die Matrix korrekt verschoben, damit richtige Pixel aufeinander aufliegen. Ausgegeben werden die Lag und Gewichte der Nachbarspixel. Die Lags müssen logischer-weise um den Nullpunk gespiegelt sein.

```
par(pty="s", mfrow=c(1,1))
mat <- matrix(0, ncol = 9, nrow = 9)
mat[5, 5] <- 1 # mittlerer Punkt ist 1
l <- generate_lags(c, 9, 9)

mat_temp <- matrix(0, ncol = 9, nrow = 9)
n <- length(l$rows)
for (i in 1:n) {
  mat_temp <- mat_temp + mat[l$rows[[i]], l$cols[[i]]]
}
plot(c(col(mat_temp)), rev(row(mat_temp)), col = mat_temp, pch = 15,
     xaxt="n", yaxt="n", xlab="", ylab="", cex = 7)
abline(h=.5:8.5, v=.5:8.5)
axis(1, 1:9)
axis(2, 1:9)
```




```
# Falscher Input: Matrix mit Nachbarschaft
generate_lags(U, 9, 9)

## Error in generate_lags(U, 9, 9):  Input of parameter 'neighb' must be a numeric 2-dim matrix
with two
## columns, representing x and y coordinates respectively.
```

- **Error-Handling:**
  - **stop11:** Input Matrix neighb hat nicht die richtige Anzahl an Spalten, welche x und y Koordinaten der Nachbarn beschreiben, bzw. ist Input Matrix weder Matrix noch Dataframe.
  - **stop12:** Matrix enthält NAs.
  - **warning7:** Der Input Nachbarschaft ist eine Dataframe mit ausschließlich numerischen Werten, obwohl es eine Matrix sein sollte. Das Programm versucht dennoch fortzufahren!

4.4 open_matrix() [_manipulation]

Parameter: (mat, lags)

Funktion transformiert eine Matrix und zugehörige lags zu einer "Open-Matrix".

- **Input Parameter:**
  - **mat:** Hier wird die Matrix, die geöffnet werden soll, angegeben. Dies kann tatsächlich eine 2-Dimensionale Matrix beliebiger Dimension sein.
  - **lags:** Input ist die ausgegebene Liste der Funktion generate_lags(), welche die Verschiebungen der Matrix entlang der Zeilen und Spalten angibt.
- **Output:** Der Output ist eine Matrix mit sovielen Spalten wie Elemente in der Input Matrix. Jede Spalte repräsentiert ein Element und seine Nachbarschaft, welche die Zeilen darstellen.
- **Beispiel:**

```
mat = matrix(1:25, ncol=5)
lags = generate_lags(generate_coords_default("Moore"))

## Error in generate_lags(generate_coords_default("Moore")):  argument "rows" is missing, with
no default

open_matrix(mat, lags)

## Error in open_matrix(mat, lags):  object 'lags' not found
```

- **Error-Handling:**
  - **Kein Error-Handling**

4.5 extreme() [_functions]

Parameter: (mat_open, sub.FUN = "max")

Die Funktion gibt den maximalen, bzw. minimalen Wert jeder Spalte einer beliebigen Matrix aus. Diese Funktion ist optimiert und arbeitet Vektorweise mit pmin/pmax.

- **Input Parameter:**
  - **mat_open:** Input ist eine Matrix beliebiger Dimension. Bei Anwenden der Funktion nhapply() wird hier die offene Matrix eingesetzt. Type sollte Numeric sein.
  - **sub.FUN:** Dieser Parameter gibt an, ob das extreme "max" oder "min" verwendet werden soll.
- **Output:** Der Output ist ein Verktor selber Länge wir Spalten der Matrix.
- **Beispiel:**

```
o_mat <- rbind(1:10, 10:1, rep(5, 10), rep(2:6,2), rep(4:5, 5), rep(5:6, 5))
print(o_mat)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## [2,]   10    9    8    7    6    5    4    3    2     1
## [3,]    5    5    5    5    5    5    5    5    5     5
## [4,]    2    3    4    5    6    2    3    4    5     6
## [5,]    4    5    4    5    4    5    4    5    4     5
## [6,]    5    6    5    6    5    6    5    6    5     6
```

```
extreme(o_mat, "max")
```

```
##  [1] 10  9  8  7  6  6  7  8  9 10
```

```
extreme(o_mat, "min")
```

```
##  [1] 1 2 3 4 4 2 3 3 2 1
```

- **Error-Handling:**
  - **stop5:** Wird ausgegeben, falls eingegebene Matrix nicht numerisch ist und der Versuch, es zu Numeric umzuwandeln, fehlschlägt (ausgenommen Factors werden nicht automatisch umgeformt)
  - **stop6:** Warnung, dass eingabe nicht ein Faktor sein darf.

4.6 count_() [_functions]

Parameter: (mat_open, sub.FUN = "max", count.output = "value")

Die Funktion gibt den häufigsten oder seltensten Wert oder Anzahl jeder Spalte zurück. Der Algorithmus funktioniert fast ausschließlich Vektorwertig, welches die Effizienz im Vergleich zum klassischen apply-Ansatz steigert.

- **Input Parameter:**
  - **mat_open:** Die eingegeben Matrix ist beliebiger Dimension und vom Typ Numeric.
  - **sub.FUN:** Gibt an, ob der häufigste oder seltenste Wert gezählt werden soll.
  - **count.output** Kann "count" und "value" sein. Beschreibt ob die Ausgabe die Anzahl oder dern Wert des häufigsten/seltensten Spaltenelement sein soll.
- **Output:** Der Output ist ein Verktor selber Länge wir Spalten der Matrix.
- **Beispiel:**

```
o_mat <- rbind(1:10, 10:1, rep(5, 10), rep(2:6,2), rep(4:5, 5), rep(5:6, 5))
print(o_mat)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## [2,]   10    9    8    7    6    5    4    3    2     1
## [3,]    5    5    5    5    5    5    5    5    5     5
## [4,]    2    3    4    5    6    2    3    4    5     6
## [5,]    4    5    4    5    4    5    4    5    4     5
## [6,]    5    6    5    6    5    6    5    6    5     6
```

```
count_(o_mat, "max", "count")
```

```
##  [1] 2 2 2 3 3 3 2 2 3 2

count_(o_mat, "max", "value")

##  [1] 5 5 4 5 5 5 4 5 5 6
```

- **Error-Handling:**
  - **warning5:** Parameter count.output wurde nicht entweder als "value", oder als "count" definiert.
  - **warning6:** sub.FUN wurde nicht richtig als "min", "max", "pmin", oder "pmax" definierst.

4.7 w_sums() [_functions]

Parameter: (mat_open, weights=1)

Die Funktion berechnet die Spaltensummen, sowie colSums(), nur erlaubt diese Funktion zusätzlich, gewichtete Summen zu berechnen.

- **Input Parameter:**
  - **mat_open:** Input ist eine Matrix beliebiger Dimension. Bei Anwenden der Funktion nhapply() wird hier die offene Matrix eingesetzt. Type sollte Numeric sein.
  - **weights:** Vektor der selben Länge wie Anzahl Zeilen (=Anzahl Elemente in Neighbourhood) welcher die Gewichtung der Nachbarn für die Summenbildung angibt.
- **Output:** Ausgabe ist natürlich abermals ein Vektor derselben Länge wie Spalten der Open-Matrix.
- **Beispiel:**

```
o_mat <- rbind(1:10, 10:1, rep(5, 10), rep(2:6,2), rep(4:5, 5), rep(5:6, 5))
weights <- c(rep(1, 3), rep(2, 3))
print(o_mat)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## [2,]   10    9    8    7    6    5    4    3    2     1
## [3,]    5    5    5    5    5    5    5    5    5     5
## [4,]    2    3    4    5    6    2    3    4    5     6
## [5,]    4    5    4    5    4    5    4    5    4     5
## [6,]    5    6    5    6    5    6    5    6    5     6

w_sums(o_mat, weights = weights)

##  [1] 38 44 42 48 46 42 40 46 44 50

# Fehler
w_sums(o_mat, weights = c(2, 3))

## Error in w_sums(o_mat, weights = c(2, 3)):  Weights-vector is not same size as neighbourhood
size.
##
## Required length:  6
```

- **Error-Handling:**
  - **stop8:** Eingabe weight ist kein Vektor.
  - **stop9:** Länge des weights-Vektors ist nicht ident mit Zeilenanzahl von open_matrix.

4.8 mat_apply() [_functions]

Parameter: (mat_open, FUN, weights=1)

Die Funktion wendet eine Funktion auf jede Spalte einer Matrix an und retourniert einen Vektor. Zuerst entscheidet sie, ob eine optimierte Funktion vorliegt, ansonsten greift sie auf die Funktion apply() zu.

Eine benutzerdefinierte, nicht optimierte Funktion wird zuerst an einer kleinen Matrix getestet, um sicherzustellen, dass der Output korrektes Format hat.

- **Input Parameter:**
  - **mat_open:** Input ist eine Matrix beliebiger Dimension. Bei Anwenden der Funktion nhapply() wird hier die offene Matrix eingesetzt.
  - **weights:** Ist der Vektor mit gewichten der Pixel, der bei der Summenfunktion in Verwendung kommt. Üblicherweise werden beim generieren von Nachbarn zuerst die Koordinaten von links oben Spaltenweise ausgewählt. Z.B.: Bei quadratischer Moore Nachbarschaft im 3x3 würde das erste Element des Vektors den Nachbarn links über der Mitte, das zweite links von der Mitte und das vierte gerade über der Mitte beschreiben
- **Output:** Ausgabe ist ein Vektor mit identer Länge wie Spalten der Input Matrix.
- **Beispiel:**

  Hier teste ich unter anderem verschiedene Input Typen: Character, Funktionen und Calls.

```
o_mat <- rbind(1:10, 10:1, rep(5, 10), rep(2:6,2), rep(4:5, 5), rep(5:6, 5))
print(o_mat)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## [2,]   10    9    8    7    6    5    4    3    2     1
## [3,]    5    5    5    5    5    5    5    5    5     5
## [4,]    2    3    4    5    6    2    3    4    5     6
## [5,]    4    5    4    5    4    5    4    5    4     5
## [6,]    5    6    5    6    5    6    5    6    5     6

mat_apply(o_mat, "max")

##  [1] 10  9  8  7  6  6  7  8  9 10

mat_apply(o_mat, max) # funktioniert für min/max

##  [1] 10  9  8  7  6  6  7  8  9 10

mat_apply(o_mat, function(x) max(x))

##  [1] 10  9  8  7  6  6  7  8  9 10
```

```r
# mat_apply with optimised functions
fs <- c("max", "min", "which.max", "which.min",
        "mean", "sum", "count", "maxcount", "mincount")
for (f in fs) {
  print(f)
  print(mat_apply(o_mat, f))
}
```

```
## [1] "max"
##  [1] 10  9  8  7  6  6  7  8  9 10
## [1] "min"
##  [1] 1 2 3 4 4 2 3 3 2 1
## [1] "which.max"
##  [1] 2 2 2 2 2 6 1 1 1 1
## [1] "which.min"
##  [1] 1 1 1 1 5 4 4 2 2 2
## [1] "mean"
##  [1] 4.500000 5.000000 4.833333 5.333333 5.166667 4.833333 4.666667
##  [8] 5.166667 5.000000 5.500000
## [1] "sum"
##  [1] 27 30 29 32 31 29 28 31 30 33
## [1] "count"
##  [1] 2 2 2 3 3 3 2 2 3 2
## [1] "maxcount"
##  [1] 2 2 2 3 3 3 2 2 3 2
## [1] "mincount"
##  [1] 2 2 2 3 3 3 2 2 3 2
```

```r
# mat_apply with custom function
mat_apply(o_mat, function(x) length(unique(x)))
```

```
##  [1] 5 5 4 4 3 3 4 5 4 4
```

```r
# Fehler
mat_apply(o_mat, "function(x) length(unique(x))")
```

```
## Error in value[[3L]](cond):  An error occured!  Make sure your function returns only one value,
## when input is a vector!
##  Error in get(as.character(FUN), mode = "function", envir = envir):  object 'function(x) length(unique(x))' of mode 'function' was not found
```

- **Error-Handling:**
  - **stop13:** Die Ausgabe der benutzerdefinierten Funktion gibt nicht einen Vektor aus, bzw. gibt apply() einen Fehler aus.

4.9 nhapply() [_master]

Parameter: (mat, FUN, restriction="TRUE", weights=1, density=FALSE, neighb_type=1, width=1, include.own=FALSE, custom_neighb, FUN_advanced, transition TRUE, force=FALSE)

Eine Funktion wird auf die Nachbarschaft jedes Pixels angewendet.

- **Input Parameter:**
  - **mat:** Input ist eine 2-Dimensionale Matrix, auf dessen Elemente die Funktion angewendet werden soll.
  - **FUN:** Ist die Funktion, welche auf jede Nachbarschaft angewendet werden soll.
    * Optimierte Funktionen "min", "max", "which.min", "which.max", "maxcount", "mincount", "sum" müssen als Zeichenkettte angegeben werden.
    * Vorhandene Funktionen: Müssen einen Vektor auf ein Element abbilden können, da die Funktion auf die Nachbarschaft angewendet wird, welche als Vektor übergeben wird.
    * Benutzerdefinierte Funktionen: Müssen ebenfalls einen Vektor auf ein Element abbilden können. z.B.: FUN = function(x) length(unique(x))
      Optimierte Funktionen müssen als Zeichenkette angegeben werden, ansonsten sollten Funktionen ohne Zeichenkette angegeben werden.
  - **restriction:** Restringiert die Input Matrix aufgrund einer Bedingungen. Input muss eine Zeichenkette sein und kann sich auch auf eine andere Matrix selber Größe beziehen. (z.B.: nhapply(mat, restriction = "mat ¿ 9")).
  - **weights:** Ist ein Vektor mit selber Länge, wie Pixel in der Nachbarschaft, welcher die Gewichte für Summen festlegt. Die Pixel einer Nachbarschaft werden Spalten weise von links nach rechts, jeweils von oben nach unten ausgewertet. So ist zum Beispiel bei einer Moore-Nachbarschaft das erste Element der Pixel links oben, das zweite links Mitte, das dritte Links unten, das vierte oben in der Mitte und so weiter! In der Grafik unter Punkt 3.1 sind die Linien ebenfalls richtig aufgezeichnet! Die Benutzung der Funktion wird vereinfacht, da bei einer Benutzerdefinierten Nachbarschaft direkt die Nachbars-Elemente als Gewichte eingegeben werden können!
  - **neighb_type:** Input gibt den Typ der Nachbarschaft an. Input kann eine der Zahlen 1, 2, oder 3 sein, als auch dazugehörig ein String namens 1:"Moore", 2:"VonNeumann" oder 3:"Diamond" sein. (bzw. ähnliche Abänderungen des Worts, siehe Code.)
  - **width:** Beschreibt die maximale Entfernung der Pixel der Nachbarschaft von der Mitte.
  - **include.own:** Legt fest, ob der mittlere Pixel auch zur Nachbarschaft angehört.
  - **custom_neighb:** Es ist möglich, eine benutzderdefinierte Nachbarschaft mitzugeben. Dies kann entweder eine Matrix mit x und y Koordinaten vom Mittelpunkt als Spalten sein, oder eine quadratische Matrix mit ungeraden Zielen/Spalten Anzahl, welche die Positionen und Gewichte der Nachbarn angibt.
  - **FUN_advanced:** Wenn angegeben, wird diese Funktion anstatt von mat_apply() angewendet. Die kann z.B. sinnvoll sein, wenn selbst ein optimierter Algorithmus zur Spaltenweisen Auswertung programmiert wird. Wichtig ist, dass die Ausgabe einen Vektor erzeugen muss, welcher die selbe Länge wie Anzahl der Element der Input Matrix hat. Zum Beispiel, soll *effizient* ein zufälliger Nachbarspixel ausgewählt werden, funktioniert das mit der Funktion:

```
a <- matrix(1:25, ncol = 5)
f <- function(m) m[matrix(c(sample(1:8, ncol(m), rep = TRUE),
1:ncol(m)), ncol = 2)]
nhapply(a, FUN_advanced = f)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    6   12   17   25   20
## [2,]    6    3   18   12   18
## [3,]    7    3   19   17    3
## [4,]   23   10   20   15   18
## [5,]   25    9   20   14    4
```

- **transition:** Falls eine Nachbarschaft, die einen Rand erreicht, auf der anderen Seite weitergehen soll, ist transition=TRUE zu setzen.
- **force:** Wird der Parameter auf TRUE gestellt können schlimme Dinge passieren, da ein paar Tests nicht durchgeführt werden. Diese Funktion ist noch in der Testphase, das Ziel ist es, benutzerdefinierte Funktionen zuerst an kleinen Matrizen zu versuchen, und dann erst auf große Anzuwenden. Dies sollte helfen, Fehler frühzeitig zu finden und genauer zu lokalisieren! In der aktuellen Version ist diese Funktion noch nicht ausgereift!

- **Output:** Die Ausgabe der Funktion ist eine Matrix derselben Dimension wie die Input Matrix.
- **Beispiel:**

  Zum Beispiel wird die Funktionsweise der Masterfunktion nhapply vorgezeigt und getestet. Diese soll für verschiedene Inputs und Nachbarschaften richtige Ergebnisse liefern. Anbei sind alle durchgeführten Tests aufgelistet:

  1. Testen von Gewichten
  2. Testen von Nachbarschaften
  3. Testen verschiedener Arten von Inputs
  4. Testen der Restriktionen
  5. Testen der FUN_advanced

  Händisches Nachrechnen bestätigte die Richtigkeit aller Funktionen

```
set.seed(42)
mat <- matrix(sample(0:5, 45, rep = TRUE), ncol = 9)
weights <- c(rep(1, 4), rep(2, 4))

# Weights testen
nhapply(mat, sum)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   25   29   30   21   23   20   21   17   24
## [2,]   17   25   24   26   30   28   21   19   18
## [3,]   23   27   19   27   21   24   23   22   23
## [4,]   18   20   19   21   22   26   23   17   19
## [5,]   28   23   23   22   30   23   30   26   31

nhapply(mat, sum, weights = weights)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   41   41   47   31   33   30   32   23   39
## [2,]   25   36   39   36   45   43   28   24   29
## [3,]   34   40   27   39   35   36   33   34   38
## [4,]   28   32   26   34   34   37   32   27   29
## [5,]   43   31   35   37   45   34   46   39   45

nhapply(mat, sum, weights = weights, density = TRUE)

##          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]
## [1,] 3.416667 3.416667 3.916667 2.583333 2.750000 2.500000 2.666667
## [2,] 2.083333 3.000000 3.250000 3.000000 3.750000 3.583333 2.333333
## [3,] 2.833333 3.333333 2.250000 3.250000 2.916667 3.000000 2.750000
```

```
## [4,] 2.333333 2.666667 2.166667 2.833333 2.833333 3.083333 2.666667
## [5,] 3.583333 2.583333 2.916667 3.083333 3.750000 2.833333 3.833333
##          [,8]     [,9]
## [1,] 1.916667 3.250000
## [2,] 2.000000 2.416667
## [3,] 2.833333 3.166667
## [4,] 2.250000 2.416667
## [5,] 3.250000 3.750000

# Neighbourhoods testen
nhapply(mat, sum, width = 2) # check (recall: transition)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   65   69   74   70   70   71   66   69   68
## [2,]   65   68   72   70   75   72   66   73   68
## [3,]   69   72   71   75   70   69   68   72   70
## [4,]   66   69   75   73   70   72   66   68   65
## [5,]   67   68   74   72   75   69   70   70   68

nhapply(mat, sum, width = 2, transition = FALSE)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   24   36   47   43   46   42   35   25   17
## [2,]   32   45   60   56   65   61   56   45   31
## [3,]   45   61   71   75   70   69   68   54   38
## [4,]   32   43   55   55   51   51   48   37   23
## [5,]   20   24   36   39   41   37   44   31   20

nhapply(mat, sum, width = 2, transition = FALSE, include.own = TRUE)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   29   39   49   48   51   45   39   29   19
## [2,]   37   49   64   61   65   63   60   45   33
## [3,]   46   61   76   75   75   74   70   55   38
## [4,]   36   46   56   57   56   53   52   42   28
## [5,]   23   28   38   42   41   42   44   34   22

nhapply(mat, sum, neighb_type = "diamond", width = 2)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   36   39   46   29   39   36   34   31   36
## [2,]   28   39   32   37   43   40   27   34   33
## [3,]   37   34   29   40   33   33   32   35   29
## [4,]   34   34   34   36   32   38   41   27   30
## [5,]   39   32   35   36   37   36   38   35   37

nhapply(mat, sum, neighb_type = "VonNeumann", width = 2)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   24   25   30   23   24   32   24   23   25
## [2,]   23   26   24   20   30   24   14   26   22
## [3,]   23   20   15   30   22   20   23   20   15
## [4,]   28   23   27   24   19   31   27   23   22
## [5,]   26   20   22   23   25   18   24   20   19
```

```r
U <-  rbind(c(.1, 0, 0, 0, .1),
            c(.1, 0, 0, 0, .1),
            c(.2, 0, 0, 0, .2),
            c(.2, 0, 0, 0, .2),
            c(.3, .3, .3, .3, .3))
nhapply(mat, sum, custom_neighb = U)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]  5.2  5.8  7.5  8.3  8.6  8.0  6.9  7.1  5.3
## [2,]  8.5  7.1  8.0  7.6  8.0  8.1  8.6 10.2  9.4
## [3,]  7.6  7.0  8.1  7.4  6.8  6.3  6.9  7.8  6.9
## [4,]  8.0  8.2  9.5  9.3  8.6  9.5  8.5  9.5  8.6
## [5,]  7.9  9.1  9.5  8.5  7.3  7.1  5.7  8.3  7.6
```

```r
nhapply(mat, sum, custom_neighb = U, weights = 1)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   28   32   38   41   38   40   34   38   30
## [2,]   39   34   39   39   37   39   37   49   42
## [3,]   36   35   42   36   36   33   34   40   36
## [4,]   37   36   43   43   41   40   37   45   39
## [5,]   38   39   46   40   35   34   32   41   35
```

```r
nhapply(mat, sum, custom_neighb = generate_coords(U)$coords)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   28   32   38   41   38   40   34   38   30
## [2,]   39   34   39   39   37   39   37   49   42
## [3,]   36   35   42   36   36   33   34   40   36
## [4,]   37   36   43   43   41   40   37   45   39
## [5,]   38   39   46   40   35   34   32   41   35
```

```r
# Inputs testen
f <- "max"
nhapply(mat, f)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    5    5    5    5    5    5    5    4    5
## [2,]    5    5    5    5    5    5    5    4    5
## [3,]    5    5    5    5    5    5    5    5    5
## [4,]    5    5    5    5    5    5    5    5    5
## [5,]    5    5    5    5    5    5    5    5    5
```

```r
nhapply(mat, "max")
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    5    5    5    5    5    5    5    4    5
## [2,]    5    5    5    5    5    5    5    4    5
## [3,]    5    5    5    5    5    5    5    5    5
## [4,]    5    5    5    5    5    5    5    5    5
## [5,]    5    5    5    5    5    5    5    5    5
```

```r
nhapply(mat, max)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    5    5    5    5    5    5    5    4    5
## [2,]    5    5    5    5    5    5    5    4    5
## [3,]    5    5    5    5    5    5    5    5    5
## [4,]    5    5    5    5    5    5    5    5    5
## [5,]    5    5    5    5    5    5    5    5    5
```

```
nhapply(mat, function(x) max(x))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    5    5    5    5    5    5    5    4    5
## [2,]    5    5    5    5    5    5    5    4    5
## [3,]    5    5    5    5    5    5    5    5    5
## [4,]    5    5    5    5    5    5    5    5    5
## [5,]    5    5    5    5    5    5    5    5    5
```

```
# restriction testen
set.seed(123)
ref <- matrix(sample(0:1, 100, rep = TRUE), ncol = 10)
nhapply(mat, sum, restriction = "ref == 1")
```

```
## Error in 1:rows:  argument of length 0
```

```
nhapply(mat, sum, restriction = "mat == 5 & ref == 1")
```

```
## Error in mat == 5 & ref == 1:  non-conformable arrays
```

```
# Advanced testen
nhapply(mat, FUN_advanced = function(x) colSums(x))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   25   29   30   21   23   20   21   17   24
## [2,]   17   25   24   26   30   28   21   19   18
## [3,]   23   27   19   27   21   24   23   22   23
## [4,]   18   20   19   21   22   26   23   17   19
## [5,]   28   23   23   22   30   23   30   26   31
```

```
nhapply(mat, FUN_advanced = function(x) {
  out <- numeric(ncol(x))
  for (i in 1:nrow(x)) out <- out + x[i, ]
  return(out)
})
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   25   29   30   21   23   20   21   17   24
## [2,]   17   25   24   26   30   28   21   19   18
## [3,]   23   27   19   27   21   24   23   22   23
## [4,]   18   20   19   21   22   26   23   17   19
## [5,]   28   23   23   22   30   23   30   26   31
```

- **Error-Handling:**
  - **stop2:** Input ist ein Dataframe, wobei die Spalten unterschiedliche Datentypen haben.

- **stop3:** Input ist nicht 2-Dimensional.
- **stop14:** Es gibt negative Gewichte und/oder die Summe der Gewichte ist 0! Fehler, falls Gewichte normiert werden sollen!
- **warning2:** Es wurden sowohl eine FUN, als auch eine FUN_advanced mitgegeben. In diesem Fall hat die FUN_advanced Vorrang

5 Erfüllen der Problemstellung

In folgenden Absätzen lege ich dar, inwiefern die Problemstellung im Bezug auf Einfachheit, Flexibilität und Effizienz erfüllt ist. Dabei nehme ich an, wir arbeiten mit einer Matrix M, die sich z.B.: durch matrix(sample(0:3, 1000000, rep=TRUE), ncol=1000) generieren lässt.

5.1 Einfachheit

Das Ziel ist es, die Funktionen so anwendungsfreundlich zu machen, sodass kaum Hintergrundwissen über die Funktionen benötigt wird. Anhand von 3 kurzen Anwendungsbeispielen werde ich die Einfachheit von nhapply darlegen.

- Angenommen, wir wollen, dass jeder Pixel den Wert des häufigsten umliegenden Nachbarn annimmt (eine häufige Anwendung von zellulären Automaten) dann funktioniert das einfach mit nhapply(A, "maxcountvalue")
- In Bildverarbeitung werden oft gewichtete Summen der Nachbarn benötigt. Die Gewichte eines Gauß-Filters auf 3x3 schauen z.B. wie folgt aus: w=c(1,2,1, 2,4,2, 1,2,1). Durch nhapply(A, sum, weights=w).
- Ist man daran interessiert, wie viele unterschiedliche Objekte in einer 7x7, Diamant-förmigen Umgebung zu finden sind, erreicht man das durch nhapply(A, function(x) length(unique(x)), neighb_type="diamond", width=3)

Zwei weitere Paremeter erleichtern die Benutzung. Mit density können Gewichte-Vektoren automatisch auf 0-1 skaliert werden, sodass sie auf 1 aufsummiert werden. Mithilfe von transition geht der Bildschirmrand auf den anderen automatisch über. Drüber hinaus sind 20 Error- und Warnmeldungen implementiert die auf fehlerhafte Inputs oder Programmfehler hinweisen und konkret den Fehler aufzeigen.

Es ist also möglich, mithilfe der Funktion nhapply(), so einfach wie mit sapply() auf Spalten, eine Funktion auf Pixel-Nachbarschaften anzuwenden.

5.2 Flexibilität

Neben der Einfachheit der Anwendung, sollen die Funktionen dennoch vielfältige Anwendungsmöglichkeiten haben. Dies wird hauptsächlich durch volle Flexibilität bei der Definition der Nachbarschaft und dem Verarbeiten der Daten erreicht.

- Der User kann einerseits von vorgefertigten Templates von Nachbarschaften wählen, welches die Nutzung vereinfacht, aber gleichzeitig besteht die Möglichkeit, eine beliebige, benutzerdefinierte Nachbarschaft anzugeben, wodurch volle Flexibilität in der Nachbarschaft erreicht wird.
  Hier gibt es die Möglichkeit, die benutzderdefinierte Nachbarschaft als Koordinaten oder als Matrix anzugeben, was die Benutzung erleichtert und die Flexibilität erhöht.
- Einerseits kann man zwischen optimierten und Base Funktionen wählen, darüber hinaus können auch benutzerdefinierte Funktionen angegeben werden. In diesem Fall wird die benutzerdefinierte Funktion an apply() in mat_open() weitergegeben.
  Weitere Flexibilität bietet der Parameter FUN_advanced. Über diesen Parameter kann eine Funktion definiert werden, die auf eine geöffnete Matrix angewendet wird. Dadurch können benutzerdefinierte, effiziente Algorithmen angewendet werden.

- Mithilfe von dem Parameter restriction kann die eingegebene Matrix aufgrund von eigener Restriktion oder einer anderen Matrix gleicher Dimension restringiert werden. Soll zum Beispiel nur die Summe von Nachbarn herangezogen werden, welche in einer ähnlichen Matrix gleicher Dimension Werte über x besitzen, kann dies über diesen Parameter geregelt werden. Werte, die die Restriktion nicht erfüllen, werden NAs: mat[!restriction] = NA

5.3 Effizienz

Getestet wird die Funktion nhapply gegen ein simple_apply. Das simple_apply stellt eine einfache Lösung, die auf einer Schleife basiert, dar. Getestet werden Funktionen auf der Matrix der Grautöne eines 4123x3731 Bildes eines Hasens. Quelle: https://goo.gl/XB568h

Getestet wird mit einer 3x3 Moore-Nachbarschaft, den eigenen Punkt inkludiert und Übergang (transition) deaktiviert. Testfunktionen sind max und sum eines Nachbars. Nachfolgend sind die Berechnungszeiten in Sekunden. Die Ergebnisse beider Funktionen sind ident.

	nhapply	simple_apply
Summe	25.62	549.50
Maximum	35.93	536.76

6 Workflow: Erstellen eines Pakets

Informationen von Hadley Wickhams Book R Packages: https://goo.gl/DHVZZ9

1. **Erstellen eines Projektordners:** Bei RStudio (1.0.153) New Project, New Directory, R Package auswählen, dem Projekt einen Titel geben und einen validen Objekt-Pfad angeben. Dort erstellt RStudio automatisch einen Ordner mit den Datein NAMESPACE, DESCRIPTION, eine Projektdatei, sowie zwei Unterordner R (für die Codes) und man (für die Dokumentation). Der Code wird in den R-Files geschrieben und die Dokumentation sollte den Roxygen Standards entsprechen. Die DESCRIPTION Datei wird upgedated, ein Author wird angegeben sowie ev. weitere Informationen, sowie License, festgelegt.

2. **Updating des Pakets:** Mithilfe von devtools::document(pkg = "pathToRCode") werden die Files man und DESCRIPTION aktualisiert.

3. **R CMD check:** Bevor das Paket an CRAN submitted wird, sollte es zumindest auf zwei Systemen geprüft sein. Dies wird vereinfacht erreicht durch die Funktion devtools::check(). In der Ausgabe sollten vor allem ausgegeben werden.

4. **Überprüfen für Windows:** Um einen R CMD check von Mac auf Windows durchzuführen, empfiehlt Wickham, die Funktion devtools::build_win() zu verwenden, welche auf den Servern von CRAN ausgeführt werden und das Paket auf Windows Systeme testet.

5. **cran-comments.md:** Diese Datei hilft MitarbeiterInnen von CRAN, das Paket schneller zu überprüfen. Es beinhaltet, auf welchen Systemen das Programm getestet wurde, und welche Notes, Warnings und Errors unbehoben sind. Diese Datei sollte aber .Rbuildignore inkludiert werden. durch devtools::use_cran_comments() wird auch dieser Schritt erleichtert.

6. **NEWS und README:** Diese Dateien dienen dem Benutzer, um Informationen über das Package zu erhalten und über die aktuelle Version informierte zu bleiben.

7. **Finale Release:** Der Release erfolgte vereinfacht über devtools::release(). Es wird noch einmal nachgeprüft, das Form-Vorschriften eingehalten werden.

Diskussion: Man muss bei beiden Lösungsansätzen jedoch Vor- und Nachteile genauer betrachten.

Der Vorteil der simple_apply Funktion liegt hauptsächlich an der Möglichkeit eines iterativen Verfahrens, welche bei nhapply ausgeschlossen, bzw. aufwendig zu implementieren ist. Außerdem ist die Implementierung von Bildschirm-Übergängen, sowie benutzerdefinierte Nachbarschaften und Funktionen mit dem selben Aufwand bewältigbar, wie bei der Erstellung von nhapply nötig war. Auf der anderen Seite ist nhapply bei nicht iterativen Anwendungen ca. 25 Mal schneller.

7 Anwendungen

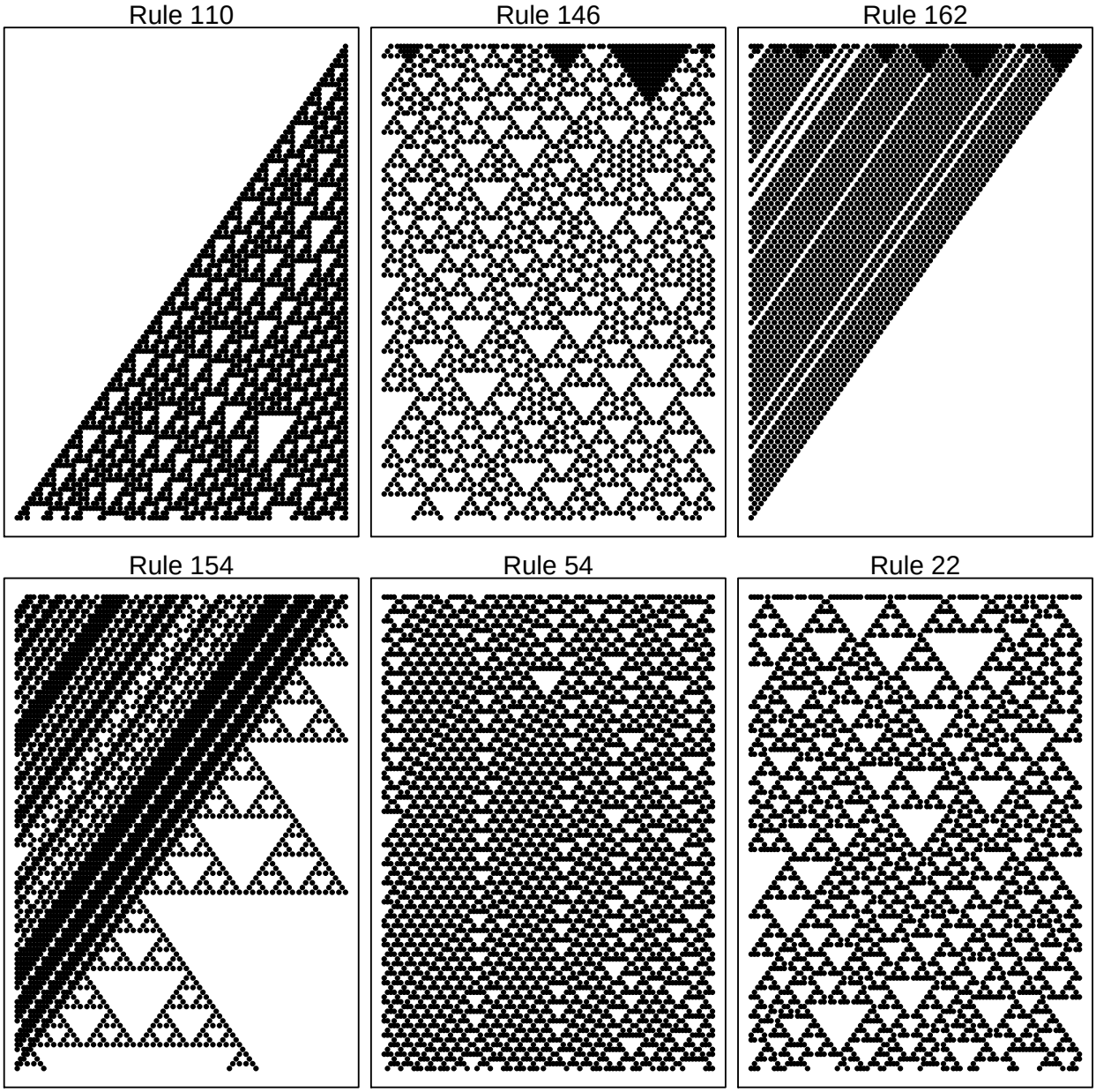
7.1 Elementare Zelluläre Automaten

https://en.wikipedia.org/wiki/Elementary_cellular_automatonQuelle Wikipedia

Mithilfe von nhapply ist es vereinfacht möglich, flexible zellulare Automaten zu bauen. Nicht nur kann man die im Wikipedia Artikel beschriebenen 256 Regeln leicht erstellen, aber auch flexibel auf mehrere, vorangegangene Zeilen und breitere Spalten zugreifen. Hier ist ein Beispiel einer Implementierung der 256 Regeln, welches jeweils die folgende Zeile abhängig von dem Pixel links, gerade und rechts über eines jeden Pixel macht. Die Regeln sind in dem oben angeführten Wikipedia Artikel genauer erläutert!

```
set.seed(42)

# Elementary cellular automaton Rules
# https://en.wikipedia.org/wiki/Elementary_cellular_automaton

rule <- function(rule_no, nbhood,
                 steps = 100, cols = 100, vector_input, probs = .4) {

  if (!missing(nbhood)) print("At this point only one-dimensional nbhoods
                              accepted! Inputs must be x-y coordinates!")

  if (missing(vector_input)) vec <- sample(0:1, cols, rep = TRUE,
                                           prob = c(probs, 1-probs))
  else vec <- vector_input

  # Get patterns from c(111, 110, 101, 100, 011, 010 001)
  choose <- rev(as.logical(intToBits(rule_no))[1:8])
  elements <- (7:0)[choose]  # select corresponding binary values
  weights <- c(4, 2, 1)  # value of triplet will be 4, 2, 1 respectively

  # Set neighbourhood (at this point only one-dimensional)
  if (missing(nbhood)) nbhood <- cbind(c(-1, 0, 1), rep(0, 3))
  else nbhood <- nbhood

  out <- matrix(0, ncol = length(vec), nrow = steps)
  out[1, ] <- vec

  for (i in 2:steps) {
    temp <- nhapply(out[i-1,, drop = FALSE], sum, density = FALSE,
                    transition = FALSE, custom_neighb = nbhood,
                    weights = c(4, 2, 1))
    out[i, temp %in% elements] <- 1
  }
  return(out)
}
```

```
r_plot <- function(results, pch = 19, cex = .5, col = results, main) {
  plot(col(results), rev(row(results)), col=col, pch = pch, cex = cex,
       xaxt="n", yaxt = "n", xlab="", ylab="")
  mtext(main)
}

par(mfrow=c(2, 3), mar = c(.5, .3, 1.5, .3))

r_plot(rule(110, probs = .1, vector_input = c(rep(0, 99), 1)), main="Rule 110")
r_plot(rule(146, probs = .2), main = "Rule 146")
r_plot(rule(162, probs = .2), main = "Rule 162")
r_plot(rule(154, probs = .5), main = "Rule 154")
r_plot(rule(54, probs = .4), main = "Rule 54")
r_plot(rule(22, probs = .3), main = "Rule 22")
```



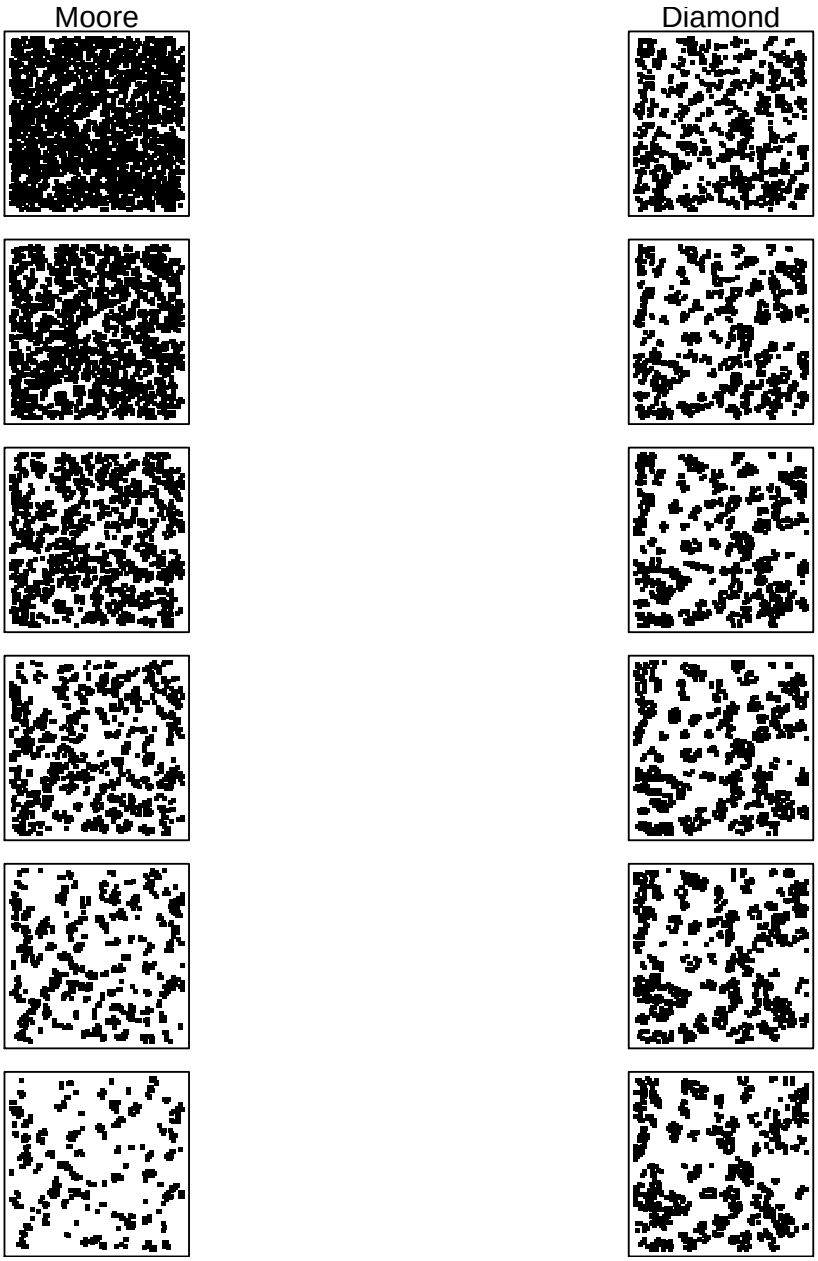
7.2 Conway's Game of Life

https://en.wikipedia.org/wiki/Conway%27s_Game_of_LifeQuelle Wikipedia

Auch der Klassiker Conway's Game of Life ist in nur einigen Zeilen programmierbar. Durch die schnelle und einfache Anpassung von nhapply ist es auch hier möglich, einfach und schnell mit Nachbarschaften, Regeln und Funktionen zu experimentieren. Bei dem Game of Life gibt gelten folgende Regeln: jeder Pixel hat 2 Zustände: lebend oder nicht (1-0 kodiert). Ist in einer Pixelnachbarschaft genau 2 Pixel, behält er den aktuellen Status bei, bei 3 Nachbarpixeln lebt der Pixel, bei weniger als 2 oder mehr als 3 lebenden Nachbarpixel stirbt er. Standardgemäß wird eine Moore Nachbarschaft der Breite eins ausgewählt.

In dieser Simulation wird die Entwicklung des Spieles bei einer Moore und einer Diamond Nachbarschaft gegenüber gestellt, wobei beide identisches Start-Bild haben. Man erkennt, dass die Diamond Nachbarschaft für stabilere Ergebnisse über die Zeit sorgt.

```r
set.seed(42)

par(mfrow=c(6, 2), mar=c(0, 0, 1, 0), pty = "s")

game_field1 <- matrix(sample(0:1, 100 * 100, rep = TRUE, prob = c(4, 3)), nrow = 100)
game_field2 <- game_field1

i <- 1
while (i < 7){
  # apply function to neighbours
  sum_neighbours1 <- nhapply(game_field1, sum)

  # game-variation:
  sum_neighbours2 <- nhapply(game_field2, sum,
  neighb_type = "diamond", width = 2)

  # process result
  game_field1[sum_neighbours1 < 2 || sum_neighbours1 > 3] <- 0
  game_field1[sum_neighbours1 == 3] <- 1
  game_field2[sum_neighbours2 < 2 || sum_neighbours2 > 3] <- 0
  game_field2[sum_neighbours2 == 3] <- 1

  # plotting
  plot(col(game_field1), rev(row(game_field1)), col = game_field1,
       pch = 15, yaxt="n", xaxt="n", ylab="", xlab="",
       main = "", cex = .5)
  if (i == 1) mtext("Moore", col = "black")

  plot(col(game_field2), rev(row(game_field2)), col = game_field2,
       pch = 15, yaxt="n", xaxt="n", ylab="", xlab="",
       main = "", cex = .5)
  if (i == 1) mtext("Diamond", col = "black")
  i <- i+1
}
```



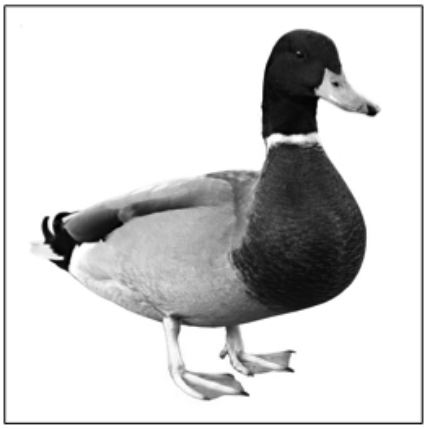
7.3 Gauss Filter

Quelle Wikipedia on Gauss: https://goo.gl/96xEX4

In diesem Beispiel wird ein Bild von einer Ente eingelesen und ein Gauss'scher Weichzeichner angewendet! Dazu benötigt man zuerst die Form einer 2-Dimensionalen Multivariaten Normalverteilung und darauf hin eine Funktion zum Generierung von Nachbarschaften mit Gewichten, die einer Gaussverteilung entsprechen. Es werden Pixel bis zu 3 Standardabweichungen vom Mittelpunkt berücksichtigt. Wie erwartet ist bereits bei diesem 640x600 großem Bild die Größe ein Problem: 640x600 (Pixel) x $11\hat{2}$ (Nachbarschaft) x 8 (Byte) entspricht ca. 370 MB Memory-Verbrauch. Plotting im Latex Dokument ist deaktiviert, um ständiges berechnen bei jedem rendern zu ersparen!

```
# library(jpeg)

# Density function Multivariate Normal Distributionl 2 Dimensional
```

```r
nv <- function(x, y, sd) 1/(2*pi*sd^2) * exp(-(x^2 + y^2)/(2 * sd^2))

# Gauss NB
g_neighb <- function(width = 1, sd) {
  # Bis maximal 3 SD von Mitte
  if (missing(sd)) sd = width/3
  width_total = (2*width + 1)
  mat <- matrix(0, ncol = width_total, nrow = width_total)
  out <- nv(row(mat)-width-1, col(mat)-width-1, sd)
  return(out/sum(out))
}

# Ergebnis ident mit https://en.wikipedia.org/wiki/Gaussian_blur
# bei sd = 0.84089642 und width = 3

# img <- readJPEG("duck.jpg")[,,1]

# apply functions
# neighbourhood <- g_neighb(5)
# img_blurred <- nhapply(img, sum, custom_neighb = neighbourhood)

# par(mfrow=c(1, 2))
# plot(1,1, yaxt="n", xaxt="n", ylab = "", xlab="")
# rasterImage(img, .6,.6,1.4,1.4)

# plot(1,1, yaxt="n", xaxt="n", ylab = "", xlab="")
# rasterImage(img_blurred, .6,.6,1.4,1.4)
```



8 Weiteres

8.1 Probleme mit großen Matrizen

Eine in dieser Aufgabe ungelöste Problemstellung ist der Umgang mit großen Matrizen und großen Nachbarschaften. Das Package hat keine "Absicherung" gegen Speicherplatz Probleme im Memory.

Problemstellung: Wird eine Funktion auf ein Bild mit 1680x1050 Pixel, also 1.764.000 Pixel insgesamt, angewendet und darauf ein Gauß-Filter mit einer Moore-Nachbarschaft der Breite 5, dieser besitzt eine Nachbarschaft (mit mittlerem Pixel) mit 121 Pixel, so ergibt sich eine offene Matrix mit 213.444.000 Pixeln.

Angenommen ein Pixel hat Größe 8 Byte, entspricht die offene Matrix hier bereits 1.7 GB, bei einer Breite von 6, ist ihre Größe bereits 2.6 GB! Die Rechnung ist bestätigt mithilfe vom Package pryr, welches die Speichergröße von Objekten in R misst.

Mit so großen Matrizen zu arbeiten ist sehr ineffizient, sobald nicht mehrIin-Memory gerechnet wird. Wie aus obigem Beispiel ersichtlich ist, tritt dieses Problem bereits bei relativ kleinen Bildern (iPhone 8 erzeugt Bilder von ca. 4000x3000 Pixeln) und kleinen Nachbarschaften auf! Außerdem werden beim Berechnen von Funktionen teils weitere Vektoren/Matrizen erstellt, wodurch die berrechnete Größe zumindest nocheinmal verdoppelt werden muss.

Dementsprechend gilt abhängig vom frei verfügbarem Memory folgende Ungleichung für effiziente Nutzung der Funktionen:

g ... Verfügbarer Speicherplatz im Memory
a/b ... Anzahl Zeilen/Spalten
k ... Breite der Nachbarschaft
m = (k*2+1)^2...$Moore$
$m = \frac{(k*2+1)^2}{2} + 1$...$Diamond$
$m = 4 * k + 1$...$VonNeumann$

$$a * b * m * 8(Byte) * 2(Buffer)/10^9 < g \tag{1}$$

Potentieller Lösungsansatz: Mithilfe oben-stehender Ungleichung wird die zu verarbeitende Matrix in Chunks aufgeteilt. Jeder Chunk entspricht einer, für den verfügbaren Memory, optimale Größe, welche mithilfe oben-stehender Formel gewählt werden kann. Nun werden die Chunks einzeln ausgewertet und letztendlich wieder zusammengefügt. Die hierbei entstehende Herausforderung besteht hauptsächlich aus den Rändern der Chunks, da diese enthalten sein müssen, aber nicht eigens ausgewertet werden dürfen.

Diese Lösung zu implementieren und zu testen, sodass sie für jegliche Betriebssysteme und Aufgaben Problemlos implementiert ist, würde leider den Rahmen dieser Aufgabe sprengen.

8.2 Herausforderungen mit Expressions

Da die Übergabe von Funktionen, Names und Strings als Funktion für nhapply einen großen Teil des Arbeitsaufwandes in Anspruch nahm, werde ich kurz auf meinen Lösungsansatz eingehen!

Folgende Problembeschreibung war zu bewältigen: Der User hat vier Möglichkeiten, Funktionen weiterzugeben: als Character ("max"), als Funktionsname (max), als benutzderdefinierte Funktion (function(x) max(x)) oder als assigned name (f = "max", f). Dabei gibt es nun zwei Herausforderungen:

1. muss dieser Input sowohl von nhapply and mat_apply weitergegeben, als auch von mat_apply direkt als Input erkannt werden. Bei der weitergabe mithilfe von match.call() (notwendig, da sonst die primitive von max genommen wird) verändert sich aber der mode von Funktion auf call, und name auf name, obwohl ein name, dem ein Character unterliegt, zuerst evaluiert werden muss.

2. soll auch bei der Funkion max erkannt werden, dass es eine optimierte Funktion davon gibt.

Lösungsansatz: Bei der Weitergabe zwischen den Funktioniert wird zuerst der Fall abgefangen, dass es sich bei einem name ein Character unterliegt. Bei mat_apply wird zuerst ein character richtig aufgefangen, nachher ein name, dann falls match.call() ein name ausgibt. Diese Reihenfolge ist wichtig und funktioniert bei Änderung nicht, da auch ein character im match.call() einen name darstellen kann!!

Übersicht zur Problemstellung und Überprüfung der Lösung

```
a <- function(fun) {
  print("a -----------")
  print(is.name(fun))
  print(mode(fun))
  print(mode(match.call()$fun))
  b(match.call()$fun)
```

```
}

b <- function(fun) {
  print("b -----------")
  print(mode(fun))
  print(mode(match.call()$fun))
}
f <- "max"
a(f)

## [1] "a -----------"
## [1] FALSE
## [1] "character"
## [1] "name"
## [1] "b -----------"
## [1] "name"
## [1] "call"

a("max")

## [1] "a -----------"
## [1] FALSE
## [1] "character"
## [1] "character"
## [1] "b -----------"
## [1] "character"
## [1] "call"

a(max)

## [1] "a -----------"
## [1] FALSE
## [1] "function"
## [1] "name"
## [1] "b -----------"
## [1] "name"
## [1] "call"

a(function(x) max(x))

## [1] "a -----------"
## [1] FALSE
## [1] "function"
## [1] "call"
## [1] "b -----------"
## [1] "call"
## [1] "call"

o_mat <- rbind(1:6, 6:1, rep(4:5, 3), rep(5:6, 3))
f <- "max"
mat_apply(o_mat, f)

## [1] 6 6 5 6 5 6

mat_apply(o_mat, "max")
```

```
## [1] 6 6 5 6 5 6

mat_apply(o_mat, max) # funktioniert für min/max

## [1] 6 6 5 6 5 6

mat_apply(o_mat, function(x) max(x))

## [1] 6 6 5 6 5 6

nhapply(mat, f)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    5    5    5    5    5    5    5    4    5
## [2,]    5    5    5    5    5    5    5    4    5
## [3,]    5    5    5    5    5    5    5    5    5
## [4,]    5    5    5    5    5    5    5    5    5
## [5,]    5    5    5    5    5    5    5    5    5

nhapply(mat, "max")

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    5    5    5    5    5    5    5    4    5
## [2,]    5    5    5    5    5    5    5    4    5
## [3,]    5    5    5    5    5    5    5    5    5
## [4,]    5    5    5    5    5    5    5    5    5
## [5,]    5    5    5    5    5    5    5    5    5

nhapply(mat, max)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    5    5    5    5    5    5    5    4    5
## [2,]    5    5    5    5    5    5    5    4    5
## [3,]    5    5    5    5    5    5    5    5    5
## [4,]    5    5    5    5    5    5    5    5    5
## [5,]    5    5    5    5    5    5    5    5    5

nhapply(mat, function(x) max(x))

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    5    5    5    5    5    5    5    4    5
## [2,]    5    5    5    5    5    5    5    4    5
## [3,]    5    5    5    5    5    5    5    5    5
## [4,]    5    5    5    5    5    5    5    5    5
## [5,]    5    5    5    5    5    5    5    5    5
```

8.3 Optimierung von Optionen

1. **count_**: Bei dieser Funktion wird entweder der Häufigste Nachbar oder wie oft der Häufigste Nachbar vorkommt, zurückgegeben. Diese Funktion erlangt durch die Vektorwertig implementierte Lösung eine immense Performenzsteigerung. **Lösungsansatz**: Zuerst wird eine Liste mit einzigartigen (unique) Werten angelegt. Diese werden nun iteriert und jeweils festgestellt, wie oft ein Wert pro Spalte in der Open-Matrix vorkommen. Mit pmax() werden nach jeder Iteration jener Wert in den Lösungsvektor übernommen, welcher häufiger vorkommt. Diese Lösung benötigt k (... Anzahl einzigartiger Werte) Iteration, wobei bei jeder Iteration lediglich ein effizienter Vergleichsoperator, eine Spaltensummen-

Funktion und eine pmax Funktion angewendet wird.

Die einfachste Alternative ist folgende Funktion mithilfe von apply(): function(x) max(table(x)) welches unter anderem einen aufwändigen Sortier-Algorithmus auf n (... Anzahl Pixel) Spalten anwenden muss.

2. **extreme**: Auch wenn diese Funktion bereits sehr effizient sind, müssen dennoch n Iterationen über alle Spalten durchlaufen werden. Deshalb wandelt die Funktion extreme ein min/max statement in eine Iteration von pmax/pmin um, um nur noch k Iterationen durchlaufen zu müssen.

3. **which.min/which.max**: Auch hier macht der Algorithmus von den effizienten Funktionen max.col() Gebrauch um die Anzahl an Iterationen von n auf k zu reduzieren!