Erste Schritte im Quantencomputing mit R und dem Paket `QsimulatR`

Alm

23. Juni 2022

Inhaltsverzeichnis

1. 1-Qbits, X-Gate und Hadamard-Gate **2**

2. n-Qbits und CNOT-Gate **3**

3. Der Pipe-Operator aus dem Paket `tidyverse` **4**

4. Die Aufgabe für den Deutsch-Josza-Algorithmus **5**

5. Implementierung der Testfunktion **5**

6. Der Deutsch-Josza-Algorithmus **6**

7. Visualisierung und Export nach qiskit **7**

Anhang **8**

A. R-Code **8**

B. Beweis für den Deutsch-Josza-Algorithmus **10**

1. 1-Qbits, X-Gate und Hadamard-Gate

Ein 1-Qbit hat im allgemeinen die Form

$$|\,\mathtt{q}\!> = a\,|\,\mathtt{0}\!> + b\,|\,\mathtt{1}\!>, \tag{1}$$

wobei a und b komplexe Koeffizienten sind, deren Betragsquadrat gleich 1 ist, sprich

$$|a|^2 + |b|^2 = 1. \tag{2}$$

|0> und |1> bezeichnen zwei Basiszustände. Unser Quantencomputer ist in der Lage, diese Zustände herzustellen (zu präparieren) und auszulesen. Die Messung ist eine Zufallsvariable

$$M(|q\rangle) \tag{3}$$

mit

$$P\left(M(|q\rangle) = 0\right) = |a|^2 \tag{4}$$

und

$$P\left(M(|q\rangle) = 1\right) = |b|^2. \tag{5}$$

Durch die Messung wird das Qbit auf den gemessenen Wert zurückgesetzt. Liest man zum Beispiel den Messwert $M(|q\rangle) = 0$ aus, so gilt unmittelbar danach |q> = |0>.

In dem Paket QsimulatR von Johann Ostmeyer und Carsten Urbach erzeugt man ein Qbit mit

```
> qstate(nbits = 1)
   ( 1 ) * |0>
```

Die Ausgabe deutet an, dass das Qbit im Zustand |0> präpariert ist. Man liest das Qbit mit dem Befehl measure aus, wobei diesem Befehl die ausgelesene Stelle als Parameter mitgegeben werden muss (hier: 1) und der Messwert mit der Option $value ausgegeben wird:

```
> q <- qstate(nbits = 1)
> measure(q, 1)$value
[1] 0
```

Man kann die Koeffizienten a und b mit dem NOT-Operator vertauschen, den man auch X-Gate nennt. Gilt ursprünglich $a = 1$ und $b = 0$, hat man danach $a = 0$ und $b = 1$:

```
> q <- qstate(nbits = 1)
> q <- X(1) * q
> measure(q, 1)$value
[1] 1
```

Eine weitere Operation, die wir benötigen, ist das Hadamard-Gate H, das die Koeffizienten a und b wie folgt mischt:

$$(a, b) \mapsto \frac{1}{\sqrt{2}}\left(a + b, a - b\right). \tag{6}$$

```
> q <- qstate(nbits = 1)
> H(1) * q
   ( 0.7071068 ) * |0>
 + ( 0.7071068 ) * |1>
```

Eine Messung dieses Qbits mit measure(q, 1)$value liefert den Messwert 0 oder 1 jeweils mit Wahrscheinlichkeit $1/2$.

2. n-Qbits und CNOT-Gate

Man kann auch Qbits mit mehr als einem 1-Qbit erzeugen. Wir beschränken uns im Folgenden auf 2-Qbits.

```
> qstate(nbits = 2)
   ( 1 ) * |00>
```

Die beiden 1-Qbits des 2-Qbits zählen wir von rechts nach links, das erste ist also hintere, das zweite das links daneben usw. Wenden wir also das X-Gate auf das 2. Qbit an, hat das vordere danach den Wert 1:

```
> q <- qstate(nbits = 2)
> X(2) * q
   ( 1 ) * |10>
```

Es ist zulässig, d.h. mit einem Quantencomputer realisierbar, das X-Gate nur dann auf das zweite Qbit anzuwenden, wenn das erste den Wert 1 hat. Diese Operation nennt man „controlled NOT" oder CNOT-Gate. In dem folgenden Code setzen wir das erste Qbit zunächst auf 1 und wenden dann das CNOT-Gate an. Danach hat das zweite Qbit ebenfalls den Wert 1.

```
> q <- qstate(nbits = 2)
> q <- X(1) * q
> CNOT() * q
   ( 1 ) * |11>
```

Man kann festlegen, welches Qbit das Kontroll-Qbit und welches das Ziel-Qbit sein soll. Die Default-Belegung ist `CNOT(bits = c(1, 2))`.

3. Der Pipe-Operator aus dem Paket `tidyverse`

Die Logik eines Quantenalgorithmus ist, auf einem Qbit zu operieren und es schließlich auszulesen (zu messen). Es geht also zunächst darum, verschiedene Operationen hintereinander auszuführen. Um dies in R intuitiv schreiben zu können, benutzen wir den Pipe-Operator `%>%`, der von dem Paket `tidyverse` zur Verfügung gestellt wird.

Zum Beispiel schreibt sich der zuletzt angegebene Code zum CNOT-Gate in einer Zeile:

```
CNOT() * (X(1) * qstate(nbits = 2))
   ( 1 ) * |11>
```

Das ist also im Sinne der Klammerung „von innen nach außen" zu lesen. Wenn sich die Operationen häufen, wird dies unübersichtlich. Lieber wollen wir den Code so schreiben, dass die Leserichtung der Logik folgt. Wir verwenden den Pipe-Operator `%>%`, den man als „und dann" lesen kann und schreiben lieber

```
> qstate(nbits = 2) %>% Xp(1) %>% CNOTp()
   ( 1 ) * |11>
```

Damit das funktioniert, haben wir die Funktionen `X` und `CNOT` durch Pipe-fähige Funktionen `Xp` und `CNOTp` ersetzt, die wie folgt definiert sind:

`Xp(q, bit)` entspricht `X(bit) * q`,
`CNOTp(q, bits)` entspricht `CNOT(bits) * q`.

Die Pipe benötigt Funktionen, bei denen das eigentliche Funktionsargument an erster Stelle der Parameter kommt, danach die übrigen Parameter. Dann funktioniert sie so, dass sie das vorhandene Argument als Argument der nachfolgenden Funktion einsetzt, wobei der erste Parameter (sprich das Funktionsargument) in der Parameterliste dieser Funktion weggelassen wird. Statt

```
> Xp(qstate(nbits = 2), 1)
   ( 1 ) * |01>
```

kann man also schreiben

```
> qstate(nbits = 2) %>% Xp(1)
   ( 1 ) * |01>
```

so dass die intuitive „und dann"-Leseweise ermöglicht wird. Unsere Implementierung der Pipe-fähigen Funktionen `Xp`, `CNOTp` sowie `Hp` findet man im Anhang A im Skript `qbib.R`.

4. Die Aufgabe für den Deutsch-Josza-Algorithmus

Wir stellen uns die Aufgabe, eine gegebene Funktion, die den Argumenten 0 und 1 die Werte 0 und 1 zuordnet, einer von zwei Klassen zuzuordnen. Die erste Klasse sind die konstanten Funktionen. Wir bezeichnen sie in naheliegender Weise mit f_0 und f_1 und sie sind durch

$$f_0(0) = 0, \quad f_0(1) = 0,$$
$$f_1(0) = 1, \quad f_1(1) = 1$$

gegeben. Die zweite Klasse von Funktionen sind die „ausgeglichenen" Funktionen (englisch: balanced). Hier gibt es ebenfalls zwei Funktionen, die wir mit f_{id} und f_{x} bezeichnen und die wie folgt definiert sind:

$$f_{\mathrm{id}}(0) = 0, \quad f_{\mathrm{id}}(1) = 1,$$
$$f_{\mathrm{x}}(0) = 1, \quad f_{\mathrm{x}}(1) = 0.$$

Der Algorithmus soll nun entscheiden, ob eine Funktion konstant oder ausgeglichen ist. Offenbar kann man das entscheiden, wenn man die Funktion auf den Argumenten 0 und 1 auswertet. Der Witz ist, dass sich im Folgenden zeigen wird, dass wir die Funktion mit dem Quantencomputer nur *einmal* auswerten müssen, um entscheiden zu können, welche der beiden Klassen sie angehört.

5. Implementierung der Testfunktion

Wir implementieren die Testfunktion als Funktion auf einem 2-Qbit wie folgt: Die Funktion operiert auf dem ersten Qbit und addiert dort den Funktionswert mod 2, wobei das Argument der Funktion das zweite Qbit ist.

Im Fall $f = f_0$ wird das erste Qbit nicht geändert. Die anderen Qbits werden schon gar nicht geändert. Wenn wir den Input (ein n-Qbit) mit `x` bezeichnen, so ist der Output ebenfalls `x`.

Im Fall $f = f_1$ wird zu dem ersten Qbit 1 mod 2 addiert. Mit anderen Worten wird es negiert, d.h. wir wenden das X-Gate auf das erste Qbit an und der Rückgabewert ist `Xp(x, 1)`.

Im Fall $f = f_{\mathrm{id}}$ addieren wir 1 mod 2 zum ersten Qbit, wenn das zweite Qbit gleich 1 ist, sonst nicht. Das ist nichts anderes als das CNOT-Gate: Der Rückgabewert ist `CNOTp(x, c(2,1))`.

Im Fall $f = f_{\mathrm{x}}$ machen wir es umgekehrt und addieren 1 mod 2 dann, wenn das zweite Qbit gleich 0 ist. Dies realisieren wir, indem wir das zweite Qbit negieren, dann das CNOT-Gate anwenden und zuletzt das zweite Qbit erneut negieren, um es im Ausgangszustand zurückzugeben. Der Rückgabewert ist somit

```
Xp(x, 2) %>% CNOTp(c(2, 1)) %>% Xp(2)
```

Fazit: Wir können die Funktion mit den uns bekannten Operationen implementieren. Der Code ist in Anhang A dokumentiert (die Funktion `uf` im Skript `deutsch.R`).

6. Der Deutsch-Josza-Algorithmus

Wir beschreiben den Deutsch-Josza-Algorithmus für ein 2-Qbit. Den Code der betreffenden Funktion `deutsch2` findet man in Anhang A im Skript `deutsch.R`, den Beweis für seine Funktionstüchtigkeit in Anhang B.

Die Funktion `deutsch2` erhält als Argument einen Parameter `type`, der angibt, welche Variante der Testfunktion `uf` ausgewertet wird. Man kann die Parameter `"0"`, `"1"`, `"id"` und `"x"` eingeben, um die im vorigen Abschnitt beschriebenen 4 Fälle durchzuspielen. Die Parameter müssen mit Anführungszeichen als String gekennzeichnet werden.

Der Algorithmus beginnt damit, dass ein 2-Qbit initialisiert wird:

```
> qstate(nbits = 2)
   ( 1 ) * |00>
```

Bevor nun die Testfunktion angewendet wird, werden die beiden 1-Qbits in geeigneter Weise gemischt. Dazu negieren wir das erste Qbit mit dem X-Gate und wenden anschließend das Hadamard-Gate auf beide Qbits an. Das Ergebnis ist folgender Zustand:

```
> qstate(nbits=2) %>%
+     Xp(1) %>%
+     Hp(1) %>%
+     Hp(2)
   ( 0.5 ) * |00>
 + ( -0.5 ) * |01>
 + ( 0.5 ) * |10>
 + ( -0.5 ) * |11>
```

Wenn man das Qbit nun messen würde, erhielte man jeden der 4 möglichen Basiszustände mit Wahrscheinlichkeit $1/4$. Danach wäre das Qbit allerdings auf den gemessenen Wert zurückgesetzt, so dass wir von einer Messung absehen. Stattdessen wenden wir die Funktion `uf` mit dem gegebenen Parameter an und „entmischen" das Qbit anschließend, indem wir den Hadamard-Operator erneut auf das zweite Qbit anwenden. Im Beispiel `type = "x"` erhalten wir

```
> qstate(nbits=2) %>%
+     Xp(1) %>%
+     Hp(1) %>%
+     Hp(2) %>%
+     uf(type = "x") %>%
+     Hp(2)
   ( -0.7071068 ) * |10>
 + ( 0.7071068 ) * |11>
```

Der Typ der Funktion kann nun am zweiten Qbit ausgelesen werden. Wenn die Funktion ausgeglichen ist, wie es hier der Fall ist, ist das zweite Qbit immer gleich 1. Ist die Funktion konstant, ist das zweite Qbit immer gleich 0. Wir probieren es am Beispiel `type = "1"`:

```
> qstate(nbits=2) %>%
+     Xp(1) %>%
+     Hp(1) %>%
+     Hp(2) %>%
+     uf(type = "1") %>%
+     Hp(2)
   ( -0.7071068 ) * |00>
 + ( 0.7071068 ) * |01>
```

Der Algorithmus schafft es also tatsächlich, den Typ der Funktion mit nur einem Funktionsaufruf zu bestimmen.

7. Visualisierung und Export nach qiskit

Das Paket `QsimulatR` stellt eine Variante der `plot`-Funktion zur Verfügung, die einen Quantenalgorithmus visualisiert. Dazu ist `plot` die Messung eines Qbits mit dem Parameter `$psi` zu übergeben:

```
plot(measure(u, 2)$psi, qubitnames = c("|1>", "|2>"))
```

erzeugt. Hierbei ist `u` das im Algorithmus manipulierte 2-Qbit, dessen zweites Qbit gemessen wird, und die Beschriftungen `|1>` und `|2>` deuten im Schaubild das erste und zweite Qbit des 2-Qbits an. Sie haben also nichts mit den Basiszuständen `|0>` und `|1>` zu tun, die für das 2-Qbit ohnehin `|00>`, `|01>`, `|10>` und `|11>` lauten. Die Ausgabe sieht im Fall `type = "id"` wie folgt aus:

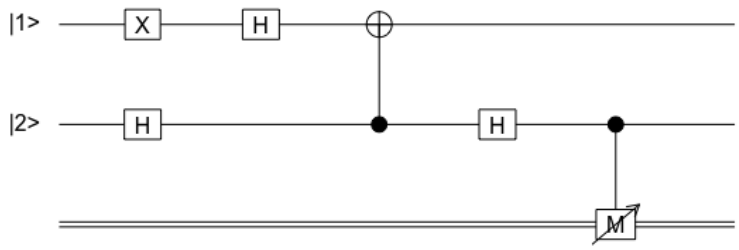

Abbildung 1: `plot(measure(u, 2)$psi, qubitnames = c("|1>", "|2>"))`

Schließlich ist es möglich, den R-Code nach qiskit zu exportieren. Dies ist das aktuelle Python-Paket zur Programmierung von Quantencomputern, das zum Beispiel von den IBM-Quantencomputern verstanden wird.

Dazu stellt `QsimulatR` die Funktion `export2qiskit` bereit, die im Arbeitsverzeichnis eine Datei `circuit.py` mit qiskit-Code erzeugt. Zum Beispiel:

```
> export2qiskit(qstate(nbits=2) %>%
+                   Xp(1) %>%
+                   Hp(1) %>%
+                   Hp(2) %>%
+                   uf(type = "id") %>%
+                   Hp(2))
```

In diesem Beispiel lautet der Inhalt der Datei `circuit.py`:

```
# automatically generated by qsimulatR
qc = QuantumCircuit(2)
qc.x(0)
qc.h(0)
qc.h(1)
qc.cx(1, 0)
qc.h(1)
```

Anhang

A. R-Code

qbib.R

```
library(qsimulatR)
library(tidyverse)

Hp <- function(x = qstate(nbits = 1), bit = 1) {
  # pipe version of qsimulatR function H, the Hadamard gate
  #
  # Args:
  #   x: qstate, to which the Hadamard gate is applied
  #   bit: number of the qbit to be shuffled. Counts beginning
  #        at the end of x
  #
  # Returns:
  #   qstate, where H is applied to bit-th qbit
  #   (counting from right to left)
  return(H(bit) * x)
}

Xp <- function(x = qstate(nbits = 1), bit = 1) {
  # pipe version of qsimulatR function X, the X gate
  # (the NOT operation)
  #
  # Args:
  #   x: qstate, to which the X gate is applied
  #   bit: number of the qbit to be changed.
  #        Counts beginning at the end of x
  #
  # Returns:
  #   qstate, where X is applied to bit-th qbit
  #   (counting from right to left)
  return(X(bit) * x)
}

CNOTp <- function(x = qstate(nbits = 2), bits = c(1,2)) {
  # pipe version of qsimulatR function CNOT, the CNOT gate
  # (the controlled NOT operation)
  #
  # Args:
  #   x: qstate, to which the CNOT gate is applied
```

```
  #   bits: integer vector of length two, the first bit being the
  #         control and the second the target bit.
  #         (counts beginning at the end of x)
  #
  # Returns:
  #   qstate, where CNOT is applied according to bits
  return(CNOT(bits) * x)
}
```

deutsch.R

```
source("qbib.R")

uf <- function(x = qstate(nbits = 2), type = "0") {
  # Testfunktion für den Deutsch-Josza-Algorithmus. Die Funktion
  # operiert auf dem ersten (rechten) qbit und addiert dort den
  # Funktionswert mod 2, wobei das Argument das zweite qbit ist
  #
  # Args:
  #   x: n-qbit mit mindestens 2 qbits
  #   type: Typ der Funktion. Es gibt vier Fälle:
  #         type = "0": f(0) = 0, f(1) = 0
  #         type = "1": f(0) = 1, f(1) = 1
  #         type = "id": f(0) = 0, f(1) = 1
  #         type = "x": f(0) = 1, f(1) = 0
  #
  # Returns:
  #   n-qbit
  if (type == "id") return(CNOTp(x, c(2, 1)))
  if (type == "1") return(Xp(x, 1))
  if (type == "x") return(Xp(x, 2) %>% CNOTp(c(2, 1)) %>% Xp(2))
  return(x)
}

deutsch2 <- function(type = "0") {
  # Deutsch-Josza Algorithmus für ein 2-qbit. Der Algorithmus
  # prüft, ob die Funktion uf(type) konstant oder ausgeglichen
  # (balanced) ist. Wir nennen sie ausgeglichen, wenn die
  # Funktionswerte 0 und 1 gleich häufig vorkommen, in
  # diesem Fall also jeweils 1-mal.
  #
  # Args:
  #   type: Typ der Funktion, die geprüft wird.
  #         Es gibt vier Fälle:
```

```
  #         type = "0": f(0) = 0, f(1) = 0 (konstant)
  #         type = "1": f(0) = 1, f(1) = 1 (konstant)
  #         type = "id": f(0) = 0, f(1) = 1 (ausgeglichen)
  #         type = "x": f(0) = 1, f(1) = 0  (ausgeglichen)
  #
  # Returns:
  #   0, wenn die Funktion konstant ist
  #   1, wenn die Funktion ausgeglichen ist
  u <- (qstate(nbits=2) %>%
          Xp(1) %>%
          Hp(1) %>%
          Hp(2) %>%
          uf(type = type) %>%
          Hp(2))
  plot(measure(u, 2)$psi, qubitnames = c("|1>", "|2>"))
  return(measure(u, 2)$value)
}
```

B. Beweis für den Deutsch-Josza-Algorithmus

Wir bezeichnen die Anwendung des X-Gates auf das n-te Qbit mit $\overset{X(n)}{\mapsto}$, für das Hadamard-Gate verwenden wir den Buchstaben H. Die ersten Operationen des Deutsch-Josza-Algorithmus schreiben sich dann wie folgt, wobei wir die Definition des Hadamard-Gates (6) verwenden:

$$\begin{aligned} |00\rangle \overset{X(1)}{\mapsto} |01\rangle \overset{H(1)}{\mapsto} \frac{1}{\sqrt{2}}(|00\rangle - |01\rangle) \\ \overset{H(2)}{\mapsto} \frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle) \end{aligned} \tag{7}$$

Der Beweis besteht nun im Folgenden darin, dass wir für alle 4 Fälle nachrechnen, dass das zweite Qbit gleich 1 ist, wenn die Testfunktion ausgeglichen ist, und gleich 0 ist, wenn die Testfunktion konstant ist – nachdem die Testfunktion auf das Qbit (7) angewendet wurde und anschließend der Hadamard-Operator $H(2)$. Wir beginnen mit den konstanten Funktionen.

1. Fall $f = f_0$. In diesem Fall bleibt das Qbit (7) zunächst ungeändert und wir erhalten

$$\begin{aligned} &\frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle) \overset{f_0}{\mapsto} \frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle) \\ &\quad \overset{H(2)}{\mapsto} \frac{1}{2\sqrt{2}}(|00\rangle + |10\rangle + |00\rangle - |10\rangle - |01\rangle - |11\rangle - |01\rangle + |11\rangle) \\ &\quad = \frac{1}{\sqrt{2}}(|00\rangle - |01\rangle) \end{aligned} \tag{8}$$

Das zweite Qbit ist also mit Sicherheit gleich 0, wie behauptet.

2. Fall $f = f_1$. In diesem Fall erhalten wir

$$\begin{aligned}
&\frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle) \overset{f_1}{\mapsto} \frac{1}{2}(|01\rangle + |11\rangle - |00\rangle - |10\rangle) \\
&\quad \overset{H(2)}{\mapsto} \frac{1}{2\sqrt{2}}(|01\rangle + |11\rangle + |01\rangle - |11\rangle - |00\rangle - |10\rangle - |00\rangle + |10\rangle) \\
&\quad = \frac{1}{\sqrt{2}}(|01\rangle - |00\rangle)
\end{aligned} \tag{9}$$

Wiederum ist das zweite Qbit gleich 0.

3. Fall $f = f_{\mathrm{id}}$.

$$\begin{aligned}
&\frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle) \overset{f_{\mathrm{id}}}{\mapsto} \frac{1}{2}(|00\rangle + |11\rangle - |01\rangle - |10\rangle) \\
&\quad \overset{H(2)}{\mapsto} \frac{1}{2\sqrt{2}}(|00\rangle + |10\rangle + |01\rangle - |11\rangle - |01\rangle - |11\rangle - |00\rangle + |10\rangle) \\
&\quad = \frac{1}{\sqrt{2}}(|10\rangle - |11\rangle)
\end{aligned} \tag{10}$$

Nun ist das zweite Qbit immer gleich 1, wie behauptet.

4. Fall $f = f_{\mathrm{x}}$.

$$\begin{aligned}
&\frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle) \overset{f_{\mathrm{id}}}{\mapsto} \frac{1}{2}(|01\rangle + |10\rangle - |00\rangle - |11\rangle) \\
&\quad \overset{H(2)}{\mapsto} \frac{1}{2\sqrt{2}}(|01\rangle + |11\rangle + |00\rangle - |10\rangle - |00\rangle - |10\rangle - |01\rangle + |11\rangle) \\
&\quad = \frac{1}{\sqrt{2}}(|11\rangle - |10\rangle)
\end{aligned} \tag{11}$$

Wiederum ist das zweite Qbit gleich 1. Daraus folgt die Behauptung.