QuantumOps

Salonik Resch

January 2019

1 Introduction

QuantumOps is designed to be a convenient tool when practicing linear algebra for quantum computation. It allows for the quick set up of kets, bras, and gate matrices to enable the user to quickly try different inputs and get a feel for the operations. Multi-qubit kets, bras, and gates can be built easily with a provided tensor product operation. It has built in all standard gates and has functionality for users to create their own gates. Additionally, a user can specify a general function and a unitary gate matrix will be generated implementing the same operation. Measurements can be performs on kets in probabilistic fashion. Also contains the QFT and examples of Shor's factoring algorithm.

1.1 Installation

QuantumOps is available on the CRAN repository. To bring up the QuantumOps environment

```
install.packages("QuantumOps")
library("QuantumOps")
```

2 Kets and Bras

Kets are represented by column vectors and Bras by row vectors. The *ket* and *bra* functions can be used to create them. The arguments to these functions are a list of amplitudes. There can be any number of amplitudes specified, but typically the number of amplitudes should be a power of 2. Each function will automatically normalize the ket or bra so that the sum of the squares of the amplitudes equals 1. The *dirac* function can be used to display kets in dirac notation.

2.1 Single Qubit Kets

Single qubit kets have two amplitudes, one for the state $|0\rangle$ and one for the state $|1\rangle$. To produce the state $|0\rangle$, use the line of code

```
ket(1,0)
```

Which produces the output

```
     [,1]
[1,] 1+0i
[2,] 0+0i
```

Which is a 2-element column vector. The first element is 1, indicating that ket is fully in the $|0\rangle$ state.

To produce the state $\sqrt{2}|0\rangle + \sqrt{2}|1\rangle$, use the line of code

```
ket(1,1)
```

Which produces the output

```
                [,1]
[1,] 0.7071068+0i
[2,] 0.7071068+0i
```

Note that amplitudes in the ket are not the same as the arguments specified. This is due to the auto normalization. The only requirement is that the two inputs be the same value. The *probs* function reports the probability of measuring each state.

```
k <- ket(1,1)
print(probs(k))
```

Output:

```
     [,1]
[1,]  0.5
[2,]  0.5
```

Which is equally likely for each state.

The *ket* function also removes global phase. Imaginary components determine the phase between the $|0\rangle$ and $|1\rangle$ components. Both components can be multiplied by a global phase, which leaves the physical meaning of ket unchanged. For example, $i|0\rangle$ is equivalent to $|0\rangle$. The following example demonstrates.

```
k <- ket(1,0)            #Specify state 1|0> + 0|1>
print("K is")
print(k)
i <- complex(1,0,1)      #Set variable i to mathematical i
k <- ket(i,0)            #Specify state i|0> + 0|1>
print("K is")
print(k)
```

Output:

```
[1] "K is"
     [,1]
[1,] 1+0i
[2,] 0+0i
[1] "K is"
     [,1]
[1,] 1+0i
[2,] 0+0i
```

Note that the meaning is only unchanged if *both* amplitudes are multiplied by the phase. For example
$i(|0\rangle + |1\rangle) = i|0\rangle + i|1\rangle = |0\rangle + |1\rangle$
But phase differences between amplitudes do matter
$|0\rangle + |1\rangle \neq |0\rangle + i|1\rangle$
It is convention to have the phase in front of the $|1\rangle$. For example, the following states are equivalent.
$i|0\rangle + |1\rangle = i(i|0\rangle + |1\rangle) = -|0\rangle + i|1\rangle = -1(-|0\rangle + i|1\rangle) = |0\rangle - i|1\rangle$
Which when normalized is
$\frac{1}{\sqrt{2}}|0\rangle - \frac{i}{\sqrt{2}}|1\rangle$

So input to the *ket* function will follow this format

```
ket(i,1)
```

Output:

```
                          [,1]
[1,] 0.7071068+0.0000000i
[2,] 0.0000000-0.7071068i
```

2.2 Multi-qubit Kets

Multi-qubit kets can be built in two ways. The first is to use the *ket* function but with more inputs. To produce the 2-qubit state $0|00\rangle + 1|01\rangle + 0|10\rangle + 0|11\rangle = |01\rangle$ use the line of code

```
ket(0,1,0,0)
```

Output:

```
     [,1]
[1,] 0+0i
[2,] 1+0i
[3,] 0+0i
[4,] 0+0i
```

Which indicates the 2-qubit system is in the $|01\rangle$ state. Note that R begins indexing at 1, rather than 0. So the amplitude corresponding to 1 is at index 2.

The second method is to use the *tensor* function. This is convenient for combining kets into larger ones. If qubit 1 is in state $|1\rangle$ and qubit 2 is in state $|0\rangle$, the combined system is in state $|10\rangle$. *tensor* performs the tensor (outer) product on the arguments from left to right, creating a larger, combined system.

```
qubit1 <- ket(0,1)
qubit2 <- ket(1,0)
combined <- tensor(qubit1,qubit2)
print(combined)
```

Output:

```
     [,1]
[1,] 0+0i
[2,] 0+0i
[3,] 1+0i
[4,] 0+0i
```

tensor can take any number of inputs and can be use to construct arbitrarily large kets.
A convenient way to define basis kets is in encoded integer form
$|2\rangle = |10\rangle$
The function *intket* can be used to create these. The first argument specifies the integer value and the second specifies how many qubits should be in the ket

```
dirac(intket(0,4))
dirac(intket(3,4))
```

Output:

```
[1] "1|0000>"
[1] "1|0011>"
```

2.3 Summary

Algebraic Representation	Input	Output

$\|0\rangle$	`ket(0,1)`
	or
	`intket(0,1)`
		`[,1]`
		`[1,] 1+0i`
		`[2,] 0+0i`

$\|1\rangle$	`ket(0,1)`
	or
	`intket(1,1)`
		`[,1]`
		`[1,] 0+0i`
		`[2,] 1+0i`

$\|0\rangle + \|1\rangle$	`ket(1,1)`	`[,1]`
		`[1,] 0.7071068+0i`
		`[2,] 0.7071068+0i`

$\|00\rangle$	`ket(1,0,0,0)`
	or
	`intket(0,2)`
		`[,1]`
		`[1,] 1+0i`
		`[2,] 0+0i`
		`[3,] 0+0i`
		`[4,] 0+0i`

$\|01\rangle + \|10\rangle$	`ket(0,1,1,0)`	`[,1]`
		`[1,] 0.0000000+0i`
		`[2,] 0.7071068+0i`
		`[3,] 0.7071068+0i`
		`[4,] 0.0000000+0i`

3 Gates

Quantum gates are represented by matrices. Matrix-vector multiplication performs the action of the gate on the vector (ket) representing the state of the system. All gates are available from a corresponding function, which will either return the matrix of the gate or perform the action of the gate on a given ket. For example, the X gate is available from the function *X()*. If *X()* is called without arguments, the matrix for X will be returned. If it is called with a ket argument, *X(ket(1,0))*, an X gate will be performed on that ket and the result will be returned.

```
#All segments of code have the same effect
#Option 1
v <- ket(1,0)   #v is qubit in state |0>
g <- X()        #Assign X gate matrix to variable g
v <- g %*% v    #multiply ket v by X matrix (in variable g) and store result in v
#Option 2
v <- ket(1,0)   #v is qubit in state |0>
v <- X() %*% v  #multiply ket v by X matrix and store result in v
#Option 3
v <- ket(1,0)   #v is qubit in state |0>
v <- X(v)       #perform X gate on v and store in v
```

QuantumOps has the following primitive gates

Gate	Function	Description	Input
I	I()	Identity	1 qubit
X	X()	Rotate around x-axis (NOT)	1 qubit
Y	Y()	Rotate around y-axis	1 qubit
Z	Z()	Rotate around z-axis (phase flip)	1 qubit
H	H()	Hadamard	1 qubit
R	R()	Phase Shift	Angle of rotation and 1 qubit
S	S()	$\pi/2$ Phase Shift	1 qubit
T	T()	$\pi/4$ Phase Shift	1 qubit
CX	CX()	Controlled X (CNOT)	2 qubits
CY	CY()	Controlled Y	2 qubits
CZ	CZ()	Controlled Z	2 qubits
BELL	BELL()	Bell gate (EPR pair creation)	2 qubits
TOFFOLI	TOFFOLI()	Toffoli gate (CCNOT)	3 qubits

3.1 Arbitrary Input Gates

QuantumOps allows for computing with an arbitrary number of qubits. Hence, gates for many qubits are required. There are a few different methods of creating higher order gates. The *tensor* function works with gates (matrices) as well as kets (vector). It can be used to build a matrix which can then mulitply a higher order ket. For example, if one wants to perform an X gate on the 2nd qubit of 2-qubit system, while leaving the first qubit alone, a 2-qubit matrix can be constructed from the I (identity) and X gates.

```
k <- ket(1,0,0,0)                #qubit in state |00>
print("Ket before gate")
print(k)
print(dirac(k))
g <- tensor(I(),X())             #construct 2-qubit gate (X gate on 2nd qubit)
k <- g %*% k                     #perform operation
print("Ket after gate")
print(k)
print(dirac(k))
```

Output:

```
[1] "Ket before gate"
     [,1]
[1,] 1+0i
[2,] 0+0i
[3,] 0+0i
[4,] 0+0i
[1] "1|00>"
[1] "Ket after gate"
     [,1]
[1,] 0+0i
[2,] 1+0i
[3,] 0+0i
[4,] 0+0i
[1] "1|01>"
```

The operation flipped the 2nd qubit, going from state $|00\rangle$ to $|01\rangle$.
Controlled gates can be created with the *controlled* function, which takes a gate matrix as input and an integer which specifies how many control qubits there should be.

```
controlled(X(),1)
```

is equivalent to

4 U function

The *U* function is a quick way to construct gates for arbitary numbers of qubits and, optionally, to apply them to kets. The same functionality as the previous example, applying an X gate to the 2nd qubit of a 2-qubit system, can be performed as follows.

```
k <- ket(1,0,0,0)                #qubit in state |00>
print("Ket before gate")
print(k)
print(dirac(k))
k <- U(I(),X(),k)        #construct and apply 2-qubit gate, and apply to k, store in k
print("Ket after gate")
print(k)
print(dirac(k))
```

Output:

```
[1] "Ket before gate"
     [,1]
[1,] 1+0i
[2,] 0+0i
[3,] 0+0i
[4,] 0+0i
[1] "1|00>"
[1] "Ket after gate"
     [,1]
[1,] 0+0i
[2,] 1+0i
[3,] 0+0i
[4,] 0+0i
[1] "1|01>"
```

5 Quantum Oracles

Quantum oracles can be formed by converting functions to quantum gates. This can be done with the *Uf* function. R allows the passing of functions as arguments. *Uf* takes as input a function followed by 2 integers. The first integer specifies the number of qubits in the input register and the second specifies the number of qubits in the target register. The following example shows a function that performs addition.

```
v <- intket(0,4)    #Create |0000>
f <- function(x){ (x+2) %% 4 }  #Define function that returns input + 2 mod 4
#Create quantum oracle for f, with 2 input register qubits
m <- Uf(f,2,2)  #and 2 target register qubits
print("Ket before")
print(dirac(v))
v <- U(m,v) #Apply oracle
print("Ket after")
print(dirac(v))
```

Output:

```
[1] "Ket before"
[1] "1|0000>"
[1] "Ket after"
```

```
[1] "1|0010>"
```

Uf is used to build the quantum oracle used in Shor's factoring algorithm

6 Quantum Fouier Transform

The QFT can be representated as a unitary matrix. The function *QFT* acts nearly identically to other gates. If the argument supplied is an integer, it will return the matrix of the QFT which would act on a ket with as many qubits as the specified integer. If a ket is supplied, it will apply the appropriately sized QFT matrix and return the modified ket.

7 Measurement

Measurement is a probabilistic process. QuantumOps simulates measurement by taking a random sample. There are two functions that perform measurement, *measure* and *reduceMeasure*. When measuring an entire ket both functions perform the same action. One of the states is chosen according to the appropriate probabilities and a list is returned. The first item in the list is the ket post measurement. The second item is the integer representation of the state that was measured.

```
v <- ket(1,1,1,1)       #initiate 2-qubit ket with each state equally probable
print("Pre-measurment")
print(dirac(v))
L <- measure(v)
v <- L[[1]]             #ket is first in returned list
s <- L[[2]]             #State is 2nd in returned list
print("Post-measurement")
print(dirac(v))
print(paste("Measured state",s))
```

Output:

```
[1] "Pre-measurment"
[1] "0.5|00> + 0.5|01> + 0.5|10> + 0.5|11>"
[1] "Post-measurement"
[1] "1|11>"
[1] "Measured state 3"
```

The measurement functions can also measure a subset of the qubits. This will happen if a list of integers is supplied after the ket in the argument list. The qubits are indexed from 0 starting from the right
$|x_3x_2x_1x_0\rangle$
This is so that the left-most qubit is the most significant. If a subset of the qubits are measured, the *measure* function will set the amplitudes to the states that are no longer possible to 0, but leave the ket with the same dimension. *reduceMeasure* will remove the states that are no longer possible, reducing the dimension of the ket.

```
v <- ket(1,1,1,1)
w <- ket(1,1,1,1)
Lv <- measure(v,0)       #Measure qubit 0 of v
Lw <- reduceMeasure(w,0) #Measure qubit 0 of w
v <- Lv[[1]]             #get measured kets
w <- Lw[[1]]
print("v is now")
print(v)
print(dirac(v))
print("w is now")
print(w)
print(dirac(w))
```

Output:

```
[1] "v is now"
             [,1]
[1,] 0.7071068+0i
[2,] 0.0000000+0i
[3,] 0.7071068+0i
[4,] 0.0000000+0i
[1] "0.707|00> + 0.707|10>"
[1] "w is now"
             [,1]
[1,] 0.7071068+0i
[2,] 0.7071068+0i
[1] "0.707|0> + 0.707|1>"
```

8 Miscellaneous Functions

addmod2 bitwise adds the two bitstrings of two integers. Finds use in Simon's algorithm
adjoint finds the adjoint of a ket, bra, or gate matrix. Is the same as the dagger operation. Will convert kets to bras and bras to ket
CFA continued fractions algorithm. Finds numerator and denominator within a specified error of input value. Used in Shor's algorithm
colv returns a column vector. Similar to ket but is not normalized
dist find the distance between two vectors *dotmod2* finds the dot product of the bitstrings of two integers and returns the value modulus 2
exponentialMod returns a function that raises a number to a power modulus a specified integer. Used to avoid overflow errors with large numbers for the creation of the oracle in Shor's factoring algorithm
hermitian returns a boolean indicating whether a gate matrix is hermitian
inner finds the inner product between two vectors
mm short for make-matrix. Performs same function as R function *matrix* but assumes it is square
norm finds the norm of a vector. Used for normalizing kets
pp short for print(paste()) and performs the same functionality
probs reports the probabilities of measuring each value of a ket by taking the absolute value square of each amplitude
Shor performs Shor's factoring algorithm by making the quantum oracle, using the QFT, and performing a measurement. Then uses CFA to find the period and the euclidean algorithm to find the prime factors
teleport gives an example of quantum teleportation of a specified qubit
unitary returns a boolean indicating whether a gate matrix is unitary