Projects about QUIPPER
Final examination – June 2017

INTRODUCTION

Quipper programs contain three main stages during operation: compile time, circuit generation time and circuit execution time. It is possible to run the compile time phase and circuit generation time phase on a classical computer. The circuit execution time phase can only occur on a physical quantum computer. Quipper uses three data types: **Bit**, **Bool**, and **Qubit**. **Bit** and **Bool** are for classical data, and **Qubit** is for quantum data. Quipper is an indentation and case sensitive language. Use **Qubit** to build the quantum circuits.

A. Quntaum Circuit Design

We import the Quipper library to use quantum built-in functions. The **feynman** keyword refers to CNOT gate, which is a function that takes 2-qubits and returns the same number of qubits. The **Circ** keyword is a type operator that indicates the side effects during evaluation of the function. We declare the variables of each input after constructing the function type. The body of this program starts with **do** operator and ends with **return** operator. The **qnot at** and the controlled operators will apply CNOT gate to the qubits. Quipper has some predefined quantum gates which we may use including **gate V at**, **gate V inv at**, **hadamard at**, **swap qubit at** and so on. The program in Fig.2 will start running from the main function, which will call the feynman gate function to print the quantum circuit in PDF format of feynman gate shown in Fig. 1.

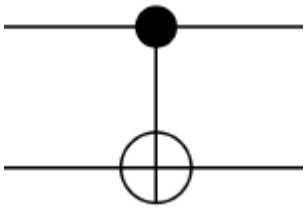


Fig. 1. Quantum circuit model of the Feynman (CNOT) gate.

```
import Quipper
feynman::(Qubit,Qubit) ->Circ(Qubit,Qubit)
feynman (q1, q2) = do
qnot at q2 ‘ controlled ‘ (q1)
return (q1,q2)
main = print simple PDF feynman
```

Fig. 2. Quantum circuit program for Feynman (CNOT) gate.

The Quipper library has another printing function to use instead of PDF namely **GateCount**, which prints the counts for each gate that has been used in the circuit as shown in Fig. 3.

```
    1: "not, arity 1", controls 1
Total gates: 1
Inputs: 2
Outputs: 2
Qubits in circuit: 2
```

Fig. 3. Circuit cost on the Feynman gate

B. Quipper Simulator

Quipper provides three different simulators, each one used differently, depending on the gates used in a circuit. Classical simulation is used to simulate classical circuits. Stabilizer simulation is used to simulate Clifford group circuits. Quantum simulation can simulate any circuit. All these simulators are generic, meaning that they take a circuit that produce a function and convert it into another function. Use run **generic function**, which will initialize the inputs of the quantum circuit to the corresponding given boolean arguments. The outputs of the quantum circuit are measured, and then the Boolean measurement results are returned. To use the circuit simulation functions, we need to import the **QuipperLib.Simulation**. The program starts in the main function:

```
import QuipperLib.Simulation
main = do
first <- run_generic_io d
feynman(False,False)
putStrLn ("Feynman first run =
"++show first)
second <- run_generic_io d feynman
(False,True)
putStrLn ( "Feynman second run =
"++ show second )
third <- run_generic_io d feynman feynman
(True , False )
putStrLn ( "Feynman third run =
" ++ show third )
fourth <- run_generic_io d feynman feynman
(True ,True)
putStrLn ( "Feynman fourth run =
" ++ show fourth )
where
d : : Double
d = undefined
```

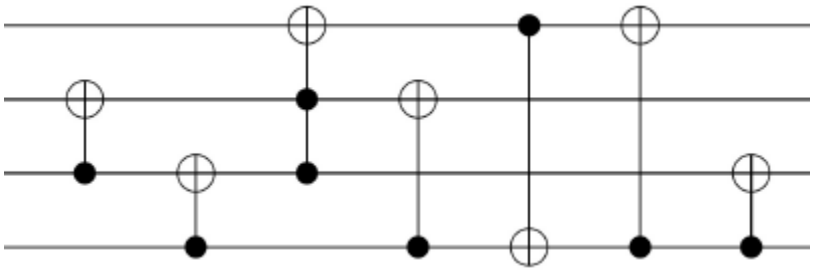
Each variable: **first**, **second**, **third** and **fourth**, is a constructor name that is declared to run the boolean arguments of the circuit function, which is the feynman gate function. The keyword **putStrLn** prints the boolean results from the keyword **show,** followed by the constructor name. The boolean results are shown as

```
Feynman first run = (False,False)
Feynman second run = (False,True)
Feynman third run = (True,True)
Feynman fourth run = (True,False)
```

Simulation results for the Feynman gate.

IMPLEMENTATION OF REVERSIBLE GATES

This section presents an example that implements the circuit design and run the Quipper circuit simulation of one reversible quantum gate, called **ALG**. The ALG gate is a 4-bit full adder gate:



Quantum circuit model of the ALG gate.

This gate has the least number of quantum cost which is eleven, and a total of seven gates required to build the circuit of the ALG gate. In this simulation, we will use *only four states out of the possible sixteen states* for testing purposes. The steps to design and simulate the ALG gate are given below:

```
import Quipper
alg::(Qubit,Qubit,Qubit,Qubit) ->
Circ (Qubit,Qubit,Qubit,Qubit)
alg (a,b,c,d) = do
qnot at b `controlled` (c)
qnot at c `controlled` (d)
qnot at a `controlled` (b,c)
qnot at b `controlled` (d)
qnot at d `controlled` (a)
qnot at a `controlled` (d)
qnot at c `controlled` (d)
return (a,b,c,d)
main = print simple PDF (alg)
```

Using the printing function **GateCount**, instead of PDF, will print the counts for each gate that has been used in the circuit:

```
    6: "not, arity 1", controls 1
    1: "not, arity 1", controls 2
Total gates: 7
Inputs: 4
Outputs: 4
Qubits in circuit: 4
```

Circuit cost on the ALG gate.

To use the circuit simulation functions, we need to import the **QuipperLib.Simulation**. The program starts in the main function:

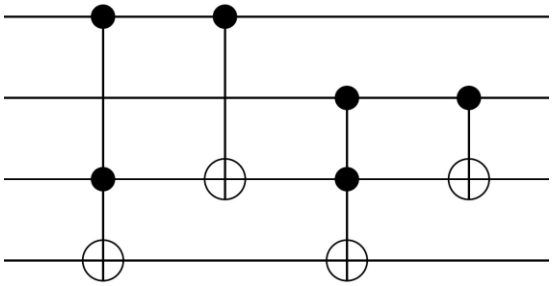
```
import QuipperLib.Simulation
main = do
first <- run_generic_io d alg
(False,False,False,False)
putStrLn ("ALG first run =
" ++ show first )
second <- run_generic_io d alg
(False,False,False,True)
putStrLn ("ALG second run =
" ++ show second )
third <- run_generic_io d alg
(False,False,True,False)
putStrLn ("ALG third run =
" ++ show third )
fourth <- run_generic_io d alg
(True,True,True,True)
putStrLn ("ALG fourth run =
" ++ show fourth )
where
d :: Double
d = undefined
```

```
ALG first run =(False,False,False,False)
ALG second run =(True,True,False,True)
ALG third run =(False,True,False,True)
ALG fourth run =(True,True,False,False)
```

Simulation results for the ALG gate.

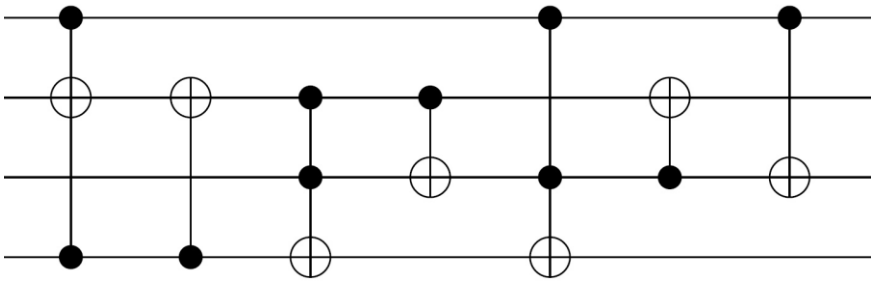
PROBLEMS TO SOLVE

1. The HNG gate is a 4-bit reversible gate, which is a full adder. A total of four gates are required to build the quantum circuit of the HNG gate:



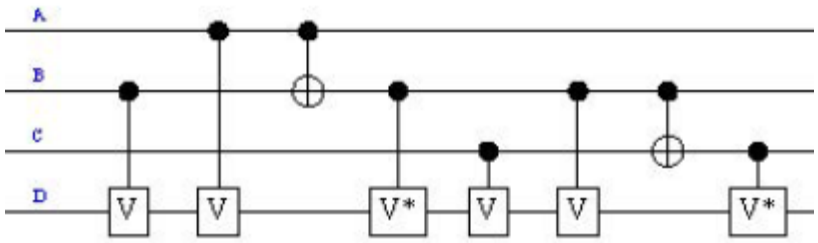
The quantum cost is twelve. In this simulation, use *only four states out of the possible sixteen states*. Write the Quipper code for implementing and simulating the HNG gate, and provide the results in the form of the solved example *on the ALG gate*.

2. The MKG gate is a 4-bit gate that is used to implement all boolean functions and design efficient adders. This gate can also work independently as a reversible full adder:



In this simulation, use *only four states out of the possible sixteen states*. Write the Quipper code for implementing and simulating the MKG gate, and provide the results in the form of the solved example *on the ALG gate*.

3. The Peres Full Adder Gate (PFAG) is designed by adding two 3-bit Peres gates. This gate needs only one clock cycle and has no more garbage outputs. The total number of gates required to build the quantum circuit of PFAG gate is eight gates:

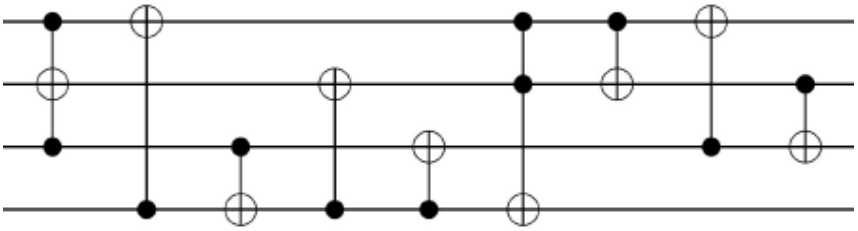


Use the command **label** to indicate the label of each input wire, as shown in figure above:

l a b e l (a , b , c , d) (”A” , ”B” , ”C” , ”D”)

Write the Quipper code for implementing and simulating the PFAG gate, and provide the results in the form of the solved example *on the ALG gate*.

4. The TSG gate is a 4-bit full adder gate. The total number of gates needed to implement the TSG full adder circuit is nine gates:



In this simulation, use *only four states out of the possible sixteen states*. The quantum cost is seventeen. Write the Quipper code for implementing and simulating the TSG gate, and provide the results in the form of the solved example *on the ALG gate*.