Mingshi Yang

Collaborator: Nuocheng Pan

Quantum Computing Results

# Dirac Notation:

|  |
| --- |
| myState2**=**[  (numpy.sqrt(0.1)**\***1.j, '101'),  (numpy.sqrt(0.5), '000') ,  (**-**numpy.sqrt(0.4), '010' ) ] PrettyPrintBinary(myState2) PrettyPrintInteger(myState2)  Paste the result of running your code on the above output:  ( 0.31622776601683794j |101> + 0.7071067811865476 |000> + -0.6324555320336759 |010> )  ( 0.31622776601683794j |5> + 0.7071067811865476 |0> + -0.6324555320336759 |2> ) |

|  |
| --- |
| print(StateToVec(myState2)) print(VecToState(StateToVec(myState2)))  Paste the result of running your code on the above output:  [ 0.70710678+0.j 0.+0.j -0.63245553+0.j 0.+0.j 0.+0.j 0.+0.31622777j 0.+0.j 0.+0.j ]  [((0.7071067811865476+0j), '000'), ((-0.6324555320336759+0j), '010'), (0.31622776601683794j, '101')] |

# Quantum Simulator I(abc)

|  |
| --- |
| Paste the result from **rand.circuit** for Simulator Ia  [((0.14565796004368076+0.1134829433332308j), '00000'), ((0.043097550946832805-0.12184596055817096j), '00001'), ((-0.0001782615750045796+0.08005459383129261j), '00010'), ((0.05035629868818657-0.00728248979944325j), '00011'), ((0.07232672559216348-0.022619270303906328j), '00100'), ((-0.03444784817538657-0.04222821092998378j), '00101'), ((0.053947952411923565+0.008954381028198075j), '00110'), ((0.0017934458673293612-0.026754015134267232j), '00111'), ((0.13464628848368856-0.12560319656579141j), '01000'), ((-0.10409972085638461-0.06762381241864739j), '01001'), ((0.010652323173449774+0.07094564243091295j), '01010'), ((0.031536789559932485-0.03537779772012157j), '01011'), ((0.007920474449196614+0.2978985122057254j), '01100'), ((0.19608126814268645-0.05835344971980598j), '01101'), ((-0.1130653010103075-0.022843082802562493j), '01110'), ((0.02567065343635605+0.07563944999428007j), '01111'), ((0.0014746900019902839-0.03645903212646821j), '10000'), ((-0.012673925486559911+0.0055143710488224865j), '10001'), ((-0.08205100536822754+0.01551399017949564j), '10010'), ((0.016011739671690343+0.022594139337382406j), '10011'), ((-0.17165615534648052+0.366922113805503j), '10100'), ((0.2774713578139311+0.042329407228946035j), '10101'), ((-0.14693154505266537-0.043614610906351804j), '10110'), ((0.007401893528164777+0.1078858351455145j), '10111'), ((-0.44267963957731876-0.08917051669782344j), '11000'), ((0.03432507377519413+0.31797145079525097j), '11001'), ((-0.05478857583103481-0.21473166615430497j), '11010'), ((-0.12098548283141325+0.06544840304773135j), '11011'), ((-0.18144336972286906-0.07802773562153227j), '11100'), ((-0.005850147569994806+0.12703336730376014j), '11101'), ((-0.057810652163285264+0.01012335649380969j), '11110'), ((-0.011813081237055226+0.02004417137333429j), '11111')] |

|  |
| --- |
| Paste the histogram from doing **measure.circuit** for simulator Ia |

|  |
| --- |
| Paste the output from **input.circuit** for Simulator Ia  [((0.1540991990041256-0.0390231428734757j), '00000'), ((0.04895105577891024-0.030018515495801427j), '00001'), ((-0.021319813075355014+0.08774787414090225j), '00010'), ((-0.11160918928703438-0.23332812293116634j), '00011'), ((-0.03390125834814464-0.009674003342578265j), '00100'), ((-0.016746063710111822-0.13026586952891356j), '00101'), ((-0.19592098529914306+0.009885177232733806j), '00110'), ((-0.0979609729023842+0.01347599683384764j), '00111'), ((-0.18082511268453247+0.014381045617727517j), '01000'), ((0.05382740960549372-0.06863464649457768j), '01001'), ((0.0010329113421169364-0.003455337432564678j), '01010'), ((-0.021540979121069115+0.0174461353591827j), '01011'), ((-0.007195884290384205+0.15958103150789765j), '01100'), ((-0.2230964998748638+0.07462898841821956j), '01101'), ((-0.03789377980145892+0.005859520326027656j), '01110'), ((0.0739034039925964-0.05074229743090883j), '01111'), ((-0.14804641063958532+0.08745635006745289j), '10000'), ((-0.06328890454391886+0.2566203542330267j), '10001'), ((0.11691184507936052+0.013808709741378385j), '10010'), ((-0.0266764615139493-0.004062838923814446j), '10011'), ((0.21923132553221591-0.21304182634262625j), '10100'), ((-0.04894199617823482-0.11635508158684049j), '10101'), ((-0.14793915437213984-0.10481145122901087j), '10110'), ((0.22316757432809134+0.05083463296709695j), '10111'), ((-0.1396394771733931-0.2372012994854915j), '11000'), ((0.18475071104180588-0.005148336320294419j), '11001'), ((-0.2632220901419421-0.06560050732512156j), '11010'), ((0.1751967845487979-0.02010633083183443j), '11011'), ((0.11220121663454644+0.0012498444406552761j), '11100'), ((0.024190828060211958-0.12445058311966195j), '11101'), ((-0.20830028473401135+0.07702445377195205j), '11110'), ((0.0547698054843489+0.3061304395854816j), '11111')] |

|  |
| --- |
| * Check that simulator Ib gives the same results as Ia for the three tests: YES * Check that simulator Ic gives the same results as Ia for the three tests: YES |

|  |
| --- |
| Paste the output of your time and RAM tests for simulators Ia,Ib,Ic (and II and III if you do them):    RAM Tests:  1a and 1b: 4700MB (94% allocated include other apps) until Memory Error with 12 wires.  1c: 3000MB  2: 400MB |

# Quantum Simulator II

|  |
| --- |
| My simulator II results for the three circuit tests (should agree with previous results):  They agree with previous results  RAND:  [((0.14565796004368065+0.11348294333323074j), '00000'), ((0.043097550946832736-0.12184596055817097j), '00001'), ((-0.00017826157500451523+0.08005459383129264j), '00010'), ((0.05035629868818656-0.007282489799443279j), '00011'), …  INPUT:  [((0.15409919900412572-0.039023142873475775j), '00000'), ((0.04895105577891031-0.03001851549580141j), '00001'), ((-0.02131981307535495+0.08774787414090221j), '00010'), ((-0.1116091892870344-0.23332812293116612j), '00011'),…  MEASURE: |

# Quantum Simulator III (extra credit)

|  |
| --- |
| My simulator III results for the three circuit tests (should agree with previous results): |

# 

# Non-Atomic Gates

|  |
| --- |
| **Circuit Description for Not:**  NOT w =  H w  P w np.pi  **H w**  **Result of Not on |1>:**  **|0>** |

|  |
| --- |
| **Circuit Description for Rz:**  Rz w X =  P w X/2  H w  P w np.pi  H w  P w -X/2  H w  P w np.pi  H w |

|  |
| --- |
| **Circuit Description for (short-range) Control-Rz:**  CRz cw ow X =  P ow X/2  CNOT cw ow  P ow -X/2  CNOT cw ow |

|  |
| --- |
| **Circuit Description for (short-range) Control-Phase:**  CPHASE cw ow X =  P cw X/2  P ow X/2  CNOT cw ow  P ow -X/2  CNOT cw ow |

|  |
| --- |
| **Circuit Description for Swap(2,5) (using short-range gates):**  SWAP 2 5 =  CNOT 2 3  CNOT 3 2  CNOT 2 3  CNOT 4 5  CNOT 5 4  CNOT 4 5  CNOT 3 4  CNOT 4 3  CNOT 3 4  CNOT 2 3  CNOT 3 2  CNOT 2 3  CNOT 4 5  CNOT 5 4  CNOT 4 5 |

|  |
| --- |
| 9 H 0 CPHASE 0 5 0.3 P 1 0.3 CNOT 4 7 SWAP 2 8  **Result of running your circuit (after precompilation) on the above input:**  **[((0.7071067811865475+0j), '000000000'), ((0.7071067811865474+1.3877787807814457e-17j), '100000000')]** |

# Phase Estimation

|  |
| --- |
| Run your simulator for 100 evenly chosen values of ϕ between 0 and 2pi and make the following graph: On the x-axis put ϕ/(2π) and on the y-axis put the θj maximally predicted by your circuit |

|  |
| --- |
| As a separate graph, let *ϕ*/(2*π*)=0.1432394487827058 and graph a histogram of the probability your circuit gives back the result *θj* (as a function of *θj*). Paste your histogram and mark on your histogram 0.1432: |

|  |
| --- |
| Produce the maximally predicted θj plot and measured θj histogram for the circuit with **2 wires** on top: |

|  |
| --- |
| Produce the maximally predicted θj plot and measured θj histogram for the circuit with **6 wires** on top. Also paste the circuit description for this phase estimate circuit. |

|  |
| --- |
| Using *ϕ*=0.5 and the given initial state, run the phase estimate circuit with 6 wires on top. Make a graph which histograms how often you get all 2^6 outputs for the top wires. |

|  |
| --- |
| Paste a circuit description for the *ϕ*=0.5, 6 top wire phase estimation circuit that uses fewer gates to represent the Quantum Fourier Transform:  CPHASE 5 6 0.5  CPHASE 4 6 1  CPHASE 3 6 2  CPHASE 2 6 4  CPHASE 1 6 8  CPHASE 0 6 16 |

|  |
| --- |
| Come up with your own circuit description for phase estimation with a U on the bottom wire that is made of NOTs and P gates, rather than just a single phase gate as we have been doing. Run your phase estimation circuit with this U gate and generate a histogram of the possible outputs for the top wires.  **4**  **INITSTATE BASIS |0001>**  **H 0**  **H 1**  **CNOT 1 2**  **CPHASE 1 2 0.3**  **CNOT 1 3**  **CNOT 0 2**  **CPHASE 0 2 0.3**  **CNOT 0 3**  **CNOT 0 2**  **CPHASE 0 2 0.3**  **CNOT 0 3**  **H 0**  **CPHASE 0 1 (-np.pi/2)**  **H 1**  **SWAP 0 1**  **MEASURE** |

# Quantum Fourier Transform

|  |
| --- |
| Paste the circuit description for the three-qubit Quantum Fourier Transform (QFT). Demonstrate that it produces the correct output when run through your simulator:  3  SWAP 0 2  H 2  CPHASE 1 2 -1.5707963267948966  H 1  CPHASE 0 2 -0.7853981633974483  CPHASE 0 1 -1.5707963267948966  H 0 |

|  |
| --- |
| Paste the circuit description for the five-qubit QFT. Demonstrate that it produces the correct output when run through your simulator. Show the output of the five-qubit QFT when run with the myInputState input file:  SWAP 0 4  SWAP 1 3  H 4  CPHASE 3 4 -1.5707963267948966  H 3  CPHASE 2 4 -0.7853981633974483  CPHASE 2 3 -1.5707963267948966  H 2  CPHASE 1 4 -0.39269908169872414  CPHASE 1 3 -0.7853981633974483  CPHASE 1 2 -1.5707963267948966  H 1  CPHASE 0 4 -0.19634954084936207  CPHASE 0 3 -0.39269908169872414  CPHASE 0 2 -0.7853981633974483  CPHASE 0 1 -1.5707963267948966  H 0  Output:  [((-0.04028017284397903-0.22092218580979783j), '00000'), ((-0.17228309183862875+0.0020356731158599067j), '00001'), ((0.07630073206041763-0.11869394467289085j), '00010'), ((-0.07697244593325293-0.12275736706850485j), '00011'), ((0.12578799246104255-0.02452186818470411j), '00100'), ((-0.21405643426925483+0.0843345595381216j), '00101'), ((-0.02928952581624665+0.012532684274186433j), '00110'), ((-0.10529823498224497+0.03863507247432338j), '00111'), ((-0.11810635653448795+0.016239738609156066j), '01000'), ((0.17684418571117022-0.05058456739571497j), '01001'), ((-0.14934362602958876-0.042653384487615006j), '01010'), ((-0.12503365026646246-0.0073475223815336055j), '01011'), ((0.24414416577410566+0.03885389987237137j), '01100'), ((-0.2533515917977188-0.2308748369894301j), '01101'), ((-0.028812459116675987-0.0853136628941907j), '01110'), ((0.02799599698216213-0.0006560027147068659j), '01111'), ((-0.03374953559586266+0.0794227539691016j), '10000'), ((-0.15946600095304178+0.146682252820024j), '10001'), ((-0.19912455571863316-0.1132357069504829j), '10010'), ((-0.045213012026622604-0.11330274327482465j), '10011'), ((-0.0014405690537448862+0.10863479965142833j), '10100'), ((-0.0770460657309485-0.16739174591450331j), '10101'), ((-0.23523338562565788-0.0009765323146217796j), '10110'), ((0.10723693789509367+0.0013057823514517966j), '10111'), ((0.21663304951566456+0.020112971083311847j), '11000'), ((0.12275573414814124-0.20487331415853322j), '11001'), ((0.024436563546173073+0.007291647371941334j), '11010'), ((0.21186219852653587-0.13549377193560447j), '11011'), ((0.15665937326444315-0.08463583856503613j), '11100'), ((0.00028527132556341317-0.022246985487152454j), '11101'), ((0.18477309891527138+0.06390445657675539j), '11110'), ((0.1411024261125871+0.16650808261166422j), '11111')] |

# 

# Understanding the QFT (extra credit)

|  |
| --- |
| Work through the three approaches for building the QFT. |

# 

# Classical Shors

|  |
| --- |
| Show that your code successfully factors numbers up to 10 bits:  63 [7, 9]  33 [11, 3]  63 [7, 9]  87 [3, 29]  95 [19, 5]  93 [31, 3]  55 [11, 5]  87 [3, 29]  69 [3, 23]  95 [19, 5]  573 [3, 191]  493 [29, 17]  565 [5, 113]  295 [59, 5]  879 [3, 293]  985 [197, 5]  285 [57, 5]  713 [23, 31]  453 [151, 3]  995 [199, 5]  861 [21, 41]  1001 [143, 7]  Factor the number 33 and give the x and r you find:  33 [11, 3]  x=5  r=10  Put a list of ten N, x, r where N is less than 5 bits and x and r are not trivial:  13 2 12  14 3 6  15 2 4  16 3 4  19 3 18  20 3 4  21 2 6  25 3 20  26 5 4  28 5 6 |

# How fast is classical Shor’s? (extra credit)

|  |
| --- |
| Plot the execution time versus k: |

|  |
| --- |
| Plot the frequency of the two failure modes as a function of k and show that they do not scale linearly with k: |

# 

# Period Finding Unitary Matrix

|  |
| --- |
| Write code to produce a period finding unitary matrix for a given co-prime (x,N). Give an example of an output unitary matrix:  x=2,N=15  [[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]  [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]  [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]  [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]  [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]  [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]  [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]  [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]] |

|  |
| --- |
| For a few different examples of x, N, generate the matrix U and find its eigenvalues e. Also, compute the period r using your Classical Shor’s algorithm. For each (x,N), paste the vector of eigenvalues e, the period r, and the vector e\*r, which should be integers:  x=2,N=15  r=4  e: [0.5 , 0.25, 0.25, 0. , 0.5 , 0.25, 0.25, 0. , 0. , 0.5 , 0.5 , 0.25, 0.25, 0. , 0. , 0. ]  e \* r: [2., 1., 1., 0., 2., 1., 1., 0., 0., 2., 2., 1., 1., 0., 0., 0.]  x=2,N=21  r=6  e: [0.5 , 0.333, 0.333, 0.167, 0.167, 0. , 0.333, 0.333, 0. ,0.333, 0.333, 0. , 0. , 0.5 , 0.5 , 0.333, 0.333, 0.167,0.167, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,0. , 0. , 0. , 0. , 0. ]  e\*r [3., 2., 2., 1., 1., 0., 2., 2., 0., 2., 2., 0., 0., 3., 3., 2., 2.,  1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.] |

|  |
| --- |
| Show that you can find the period r given a random eigenvalue of the matrix U for a particular (x,N).  X=7, N=99, phase =0.36666666666666686, r=30  X=3, N=65, phase=0.16666666666666674, r=6 |

|  |
| --- |
| Using the above algorithm for factoring using only the eigenvalues of the U matrix (without the help of the Classical Shor’s algorithm), factor some numbers. Paste the output here:  45 [3, 15]  35 [7, 5]  93 [31, 3] |

# 

# Adding classical gates to your simulator

|  |
| --- |
| Paste in your circuit descriptions that use the xyModN and control-xyModN gates and show the input and output that verifies that they work:  7  INITSTATE BASIS |0100010>  FUNC 0 4 xyModN 3 10  OUT: |0010010>  Same as  VecToState(  np.kron(np.kron(np.kron(xNUnitary(3,10),np.identity(2)),np.identity(2)),np.identity(2))  @StateToVec([[1,'0100010']])  )  Same as running  7  INITSTATE BASIS |0100010>  CFUNC 5 0 4 xyModN 3 10 |

# 

# 

# Shor’s Algorithm

|  |
| --- |
| Show that your quantum computing simulator running the Quantum Shor’s circuit can successfully factor numbers. Try to factor 21.  Input: 21  Output: [3, 7] |

|  |
| --- |
| Show that your simulator runs faster with the speed-up trick.  Time for 100 factorizations of 21:  Old: 5.900653600692749 s  New: 4.515197992324829 s  Time for 100 factorizations of 15:  Old: 1.5817365646362305 s  New: 1.384293794631958 s  Time for 20 factorizations of 91:  Old: 50.66567897796631 s  New: 42.38434076309204 s |