Documentation
Overview

There are 3 classes:

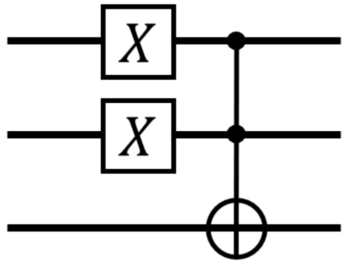
1. QuantumAlgorithms - higher-level functions with quantum computations which are built using pennylane library
- U_n()
- C_U_n()
- ADDER
- ADDER_MOD()
- Controlled_MULT_MOD()
- MODULAR_EXPONENTIATION()
- QFT()
- Phase_Estimation()

2. QuantumGates - simpler functions with quantum computations which are built using pennylane library
- T_dagger()
- Toffoli()
- Controlled_Toffoli()
- SWAP()
- Controlled_SWAP()
- Controlled_Controlled_SWAP()
- Controlled_register_SWAP()
- Controlled_reset_zero_register_to_N()
- CARRY()
- SUM()
- Controlled_U_block()
- CR_k()
- C_U()
- gray_code_C_X()
- Two_level_U()
- controlled_Two_level_U()

3. ClassicalOperations - auxiliary functions without quantum computations
- states_vector()
- int_to_binary_array()
- get_non_trivial_indices()
- Gray_code()
- gcd()
- diophantine_equation_auxiliary()
- diophantine_equation()
- modular_multiplicative_inverse()
- matrix_power()
- matrix_natural_power()
- ZY_decomposition_angles()
- U_given_ZY_angles()
- Two_level_unitary_decomposition()
- matrix_natural_power()
- ZY_decomposition_angles()

Example

Functions from classes QuantumAlgorithms and QuantumGates can be used in the same way as elementary gates inside pennylane's QNode structure. The script below implements the following 3-qubit quantum circuit with 2 standard PauliX gates from pennylane library and 1 Toffoli gate from QuantumGates class:



In [1]:
```
import pennylane as q
from QuantumOperations import QuantumGates
```

In [2]:
```
qg = QuantumGates()
```

In [3]:
```
# wires
wires = ['q0','q1','q2']
# device
dev = q.device('default.qubit', wires=wires, shots=1000000, analytic=None)

# circuit
def func():

    # standard X gates
    q.PauliX(wires=wires[0])
    q.PauliX(wires=wires[1])
    # gate from QuantumGates class
    qg.Toffoli(wires=[wires[0],wires[1],wires[2]])

    return q.probs(wires)

# QNode
circuit = q.QNode(func,dev)
```

In [4]:
```
# given rntry |000>, the circuit always gives |111>
circuit()
```

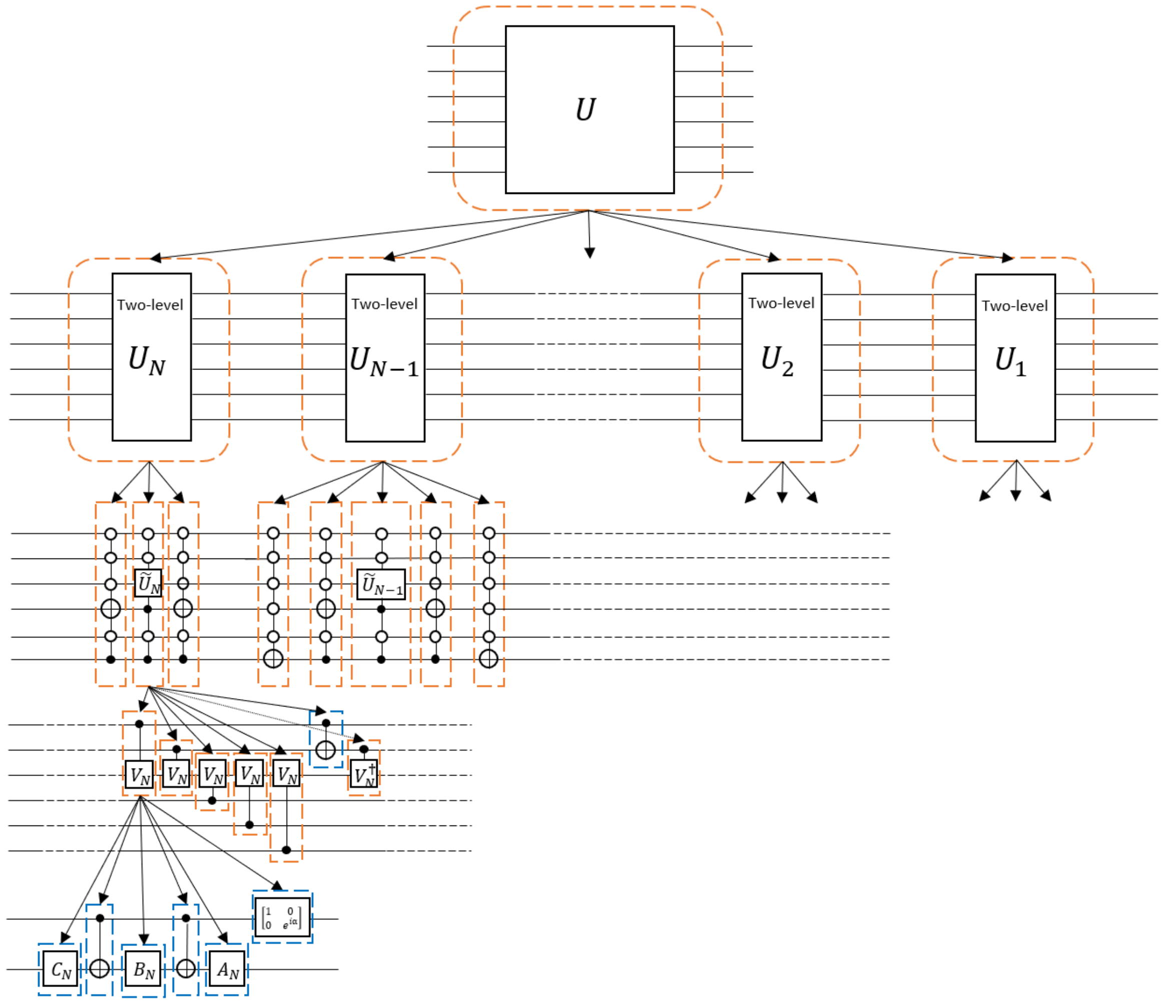
Out[4]:
```
tensor([0., 0., 0., 0., 0., 0., 0., 1.], requires_grad=False)
```

1 Functions from QuantumAlgorithms class

1.1 QuantumAlgorithms.U_n(U, wires)

U_n() is a realization of arbitrary unitary gate specified by the arbitrary matrix U (arbitrary amount of qubits up to 6).

Workings of U_n() function are motivated by Nielsen, Chuang, chapter 4.5, and can be illustrated by the following scheme:



On this scheme,

- orange dotted frame denotes a gate type which is implemented via one of the functions of the class from the list "Functions **with quantum computations**"above

- blue dotted frame denotes a gate type which is implemented via standard **pennylane library**

As one can see, arbitrary unitary gate U is realized by elementary single-qubit rotation operations **RZ, RY, Phase Shift** and two-qubit **CNOT** operation

Current **maximal number of qubits** for which operations from the class can be implemented is **6 qubits**. Note that no work qubits are used in any of the functions from the class. It essentially means that, for instance, 6-qubit operation can be performed using only these 6 qubits and no additional work qubits.

1. **Parameters**

- U - unitary matrix to act on qubits in wires. Current maximal number of qubits is 6.

2. **Results**:

- system after U acts on it

1.2 QuantumAlforithms.C_U_n(U,control_wire,operation_wires)

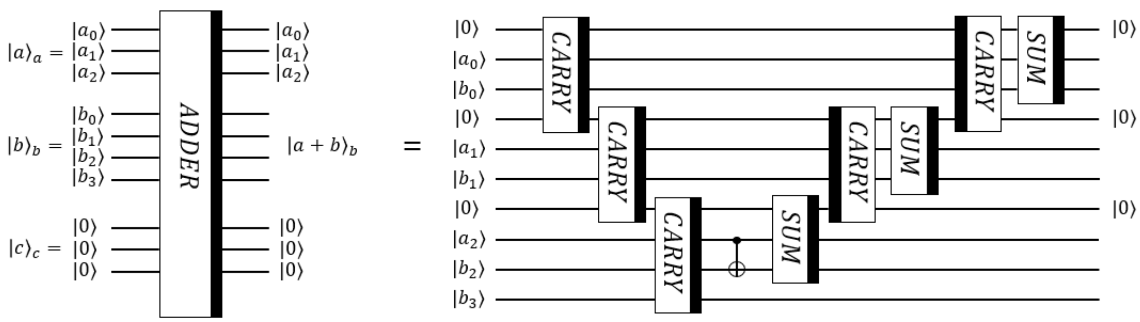
Controlled version of QuantumAlgorithms.U_n(), up to n = 5 wires in operation_wires

1. **Parameters**
- U - $(2^n) \times (2^n)$ arbitrary unitary matrix U, where n = len(operation_wires)
- control_wire
- operation_wires
2. **Results**:
- transformed control_wire
- transformed operation_wires

1.3 QuantumAlgorithms.ADDER(wires_a,wires_b,wires_c,inverse=False)

ADDER is a circuit which implements addition for classical inputs (i.e. 0s or 1s in a qubit registers). Algorithm works for arbitrary amount of qubits, but scheme is provided for 3 qubits.

Workings of ADDER function are motivated by 'Quantum Networks for Elementary Arithmetic Operations' – Vedral, Barenco, Ekert, 1995, and can be illustrated by the following scheme:



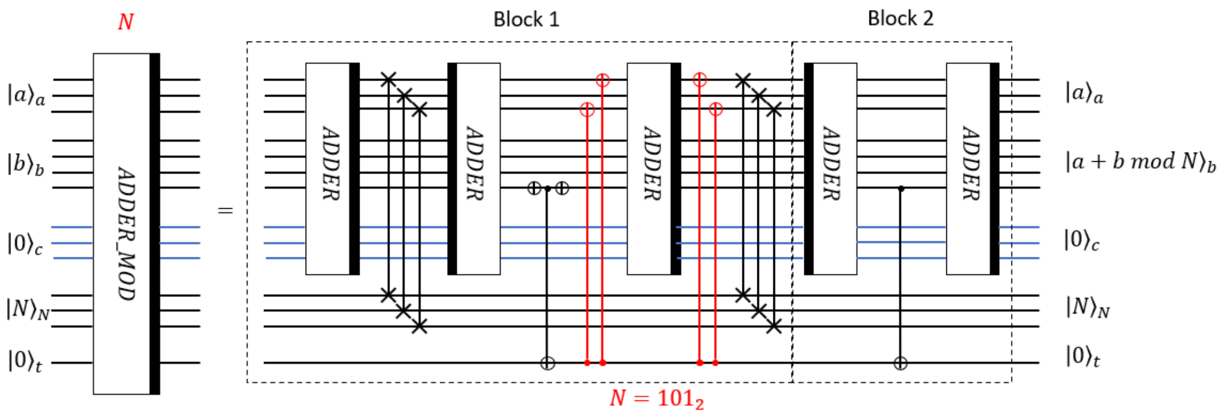
More elaborate description: https://quantumcomputing.stackexchange.com/questions/6842/is-there-a-simple-formulaic-way-to-construct-a-modular-exponentiation-circuit/1477314773

1. **Parameters**
- wires_a - names for wires of register of "number"a to be added
- wires_b - names for wires of register of "number"b to be added
- wires_c - names for wires of register of zeros
- inverse - if passed False, then circuit from the scheme is implemented; if passed True, then circuit with inverse order of all elementary operations with respect to the circuit from the scheme is implemented
2. **Results**:
- "number"a in register wires_a
- "number"a+b in register wires_b
- zeros in register wires_c

1.4 QuantumAlgorithms.ADDER_MOD(wires_a,wires_b,wires_c, wires_N,wires_t,N,inverse=False)

ADDER_MOD is a circuit which implements modular addition for classical inputs (i.e. 0s or 1s in a qubit registers). Circuit works as addition of a and b modulo N only if $0 <= a, b < N$. Algorithm works for arbitrary amount of qubits, but scheme is provided for 3 qubits.

Workings of ADDER_MOD function are motivated by 'Quantum Networks for Elementary Arithmetic Operations' – Vedral, Barenco, Ekert, 1995, and can be illustrated by the following scheme:



More elaborate description: https://quantumcomputing.stackexchange.com/questions/6842/is-there-a-simple-formulaic-way-to-construct-a-modular-exponentiation-circuit/1477314773

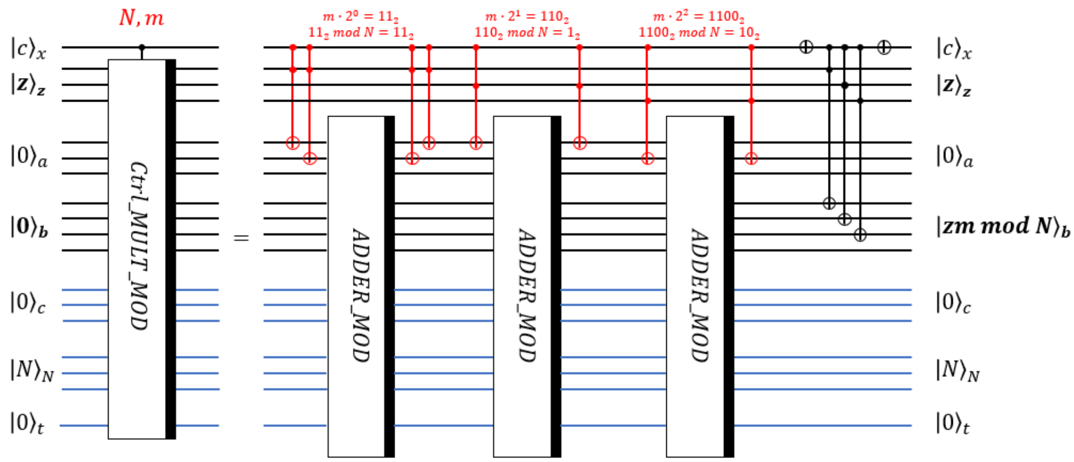
1. **Parameters**
- wires_a - names for wires of register of "number"a to be added modulo N
- wires_b - names for wires of register of "number"b to be added modulo N
- wires_c - names for wires of register c initialized with zeros
- wires_N - names for wires of register of "number"N
- wires_t - name for auxiliary control wire of register t

2. **Results**:
- "number"a in register wires_a
- "number"a+b mod N in register wires_b
- zeros in register wires_c
- "number"N in register wires_N
- zero in register wires_t

1.5 QuantumAlgorithms.Controlled_MULT_MOD(control_wire,wires_z, wires_a,wires_b,wires_c,wires_N,wires_t,N,m,inverse=False)

Controlled_MULT_MOD is a circuit which implements controlled modular multiplication for classical inputs (i.e. 0s or 1s in a qubit registers). Algorithm works for arbitrary amount of qubits, but scheme is provided for 3 qubits.

Workings of Controlled_MULT_MOD function are motivated by 'Quantum Networks for Elementary Arithmetic Operations' – Vedral, Barenco, Ekert, 1995, and can be illustrated by the following scheme:



More elaborate description: https://quantumcomputing.stackexchange.com/questions/6842/is-there-a-simple-formulaic-way-to-construct-a-modular-exponentiation-circuit/1477314773

1. **Parameters**

- control_wire - name for control wire
- wires_z - names for wires of register of "number"z to be multiplied by m modulo N
- wires_a - names for auxiliary wires of register a initialized with zeros
- wires_b - names for auxiliary wires of register of b initialized with zeros
- wires_c - names for auxiliary wires of register c initialized with zeros
- wires_N - names for wires of register of "number"N
- wires_t - name for auxiliary control wire of register t
- N - number "modulo"
- m - multiplier
- inverse - if passed False, then circuit from the scheme is implemented; if passed True, then circuit with inverse order of all elementary operations with respect to the circuit from the scheme is implemented

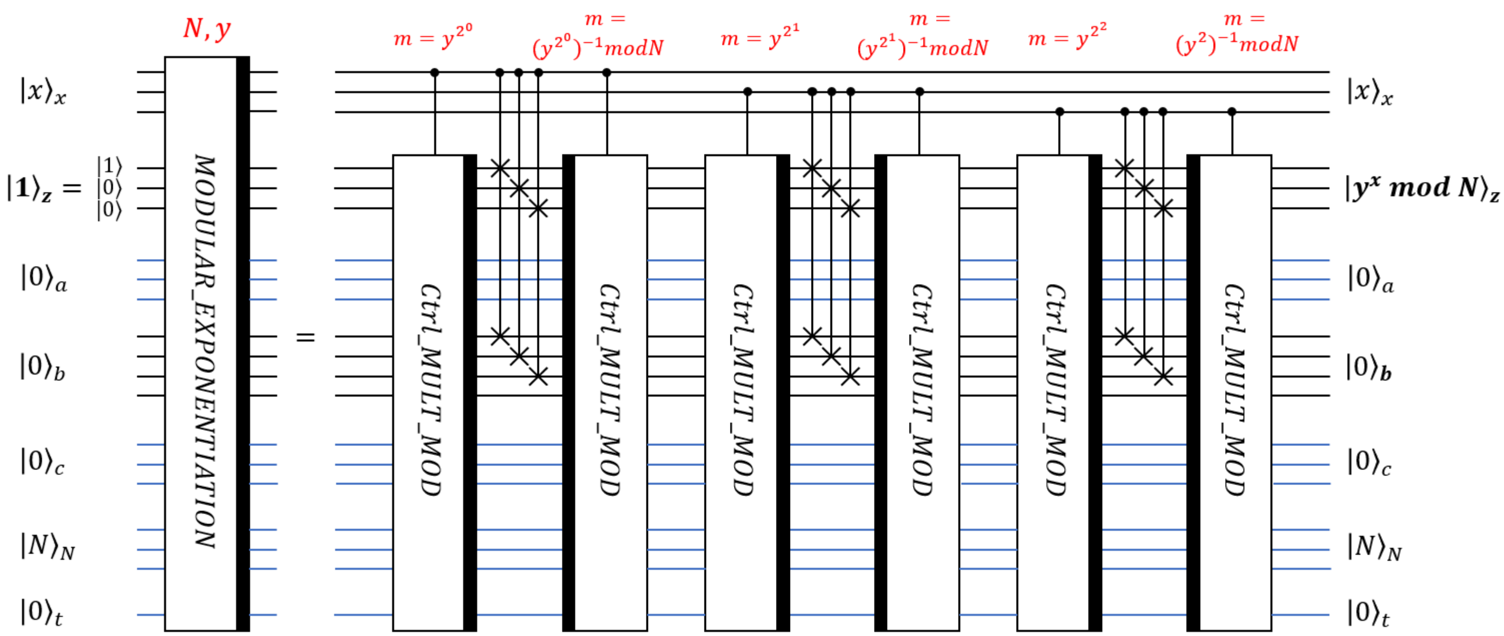
2. **Results**:

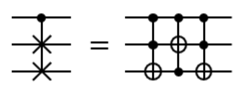
- control_wire - initially passed control
- wires_z - initially passed z
- wires_a - initially passed zeros
- wires_b - z*m mod N
- wires_c - initially passed zeros
- wires_N - initially passed N
- wires_t - initially passed auxiliary control

1.6 QuantumAlgorithms.MODULAR_EXPONENTIATION(wires_x,wires_z, wires_a,wires_b,wires_c,wires_N,wires_t,N,y,inverse=False)

MODULAR_EXPONENTIATION is a circuit which implements effective $O(n^3)$ modular exponentiation for classical inputs (i.e. 0s or 1s in a qubit registers). Algorithm works for arbitrary amount of qubits, but scheme is provided for 3 qubits.

Workings of MODULAR_EXPONENTIATION function are motivated by 'Quantum Networks for Elementary Arithmetic Operations' – Vedral, Barenco, Ekert, 1995, and can be illustrated by the following scheme:





More elaborate description: https://quantumcomputing.stackexchange.com/questions/6842/is-there-a-simple-formulaic-way-to-construct-a-modular-exponentiation-circuit/1477314773

1. **Parameters**

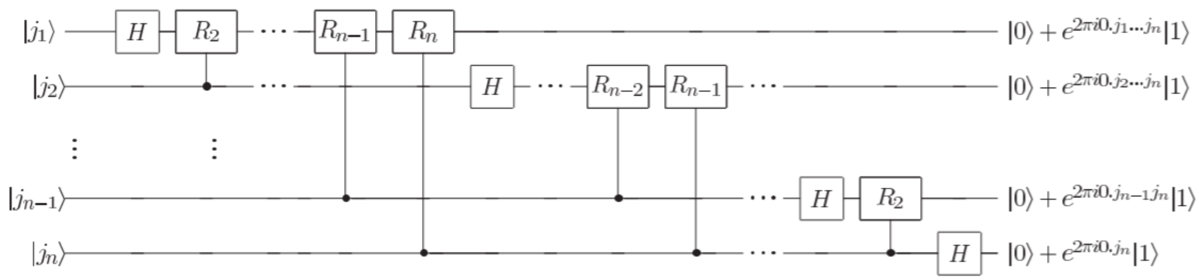
- wires_x - name for wires with "number"x to be "number"y's power
- wires_z - names for wires of register initialized with "number"1
- wires_a - names for auxiliary wires of register a initialized with zeros
- wires_b - names for auxiliary wires of register of b initialized with zeros
- wires_c - names for auxiliary wires of register c initialized with zeros
- wires_N - names for wires of register of "number"N
- wires_t - name for auxiliary control wire of register t
- N - number "modulo"
- y - number to be exponentiated to the power x
- inverse - if passed False, then circuit from the scheme is implemented; if passed True, then circuit with inverse order of all elementary operations with respect to the circuit from the scheme is implemented

2. **Results**:

- control_x - initially passed x
- wires_z - y$^x modN$
- wires_a - initially passed zeros
- wires_b - initially passed zeros
- wires_c - initially passed zeros
- wires_N - initially passed N
- wires_t - initially passed auxiliary control

1.7 QuantumAlgorithms.QFT(wires,inverse=False)

QFT is a circuit which implements quantum Fourier transform - see Nielsen Chuang p.219

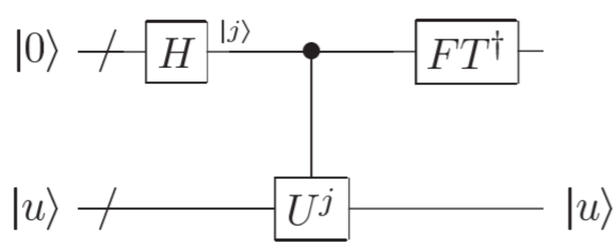


1. **Parameters**
- wires
- inverse - if passed False, then circuit from the scheme is implemented; if passed True, then circuit with R_k_dagger instead of R_k is implemented, i.e. inverse quantum Fourier transform
2. **Results**:
- qubits, tensor product of which is a Fourier transform of j encoded as 0.j1j2j3..jn

1.8 QuantumAlgorithms.Phase_Estimation(U,t,wires)

Phase_Estimation is a circuit which implements general Phase estimation procedure using arbitrary (up to 5-qubits) unitary operation U - see Nielsen Chuang p.221

This realization of Phase estimation is not efficient (exponential) since it uses elements with controlled arbitrary unitary transformation

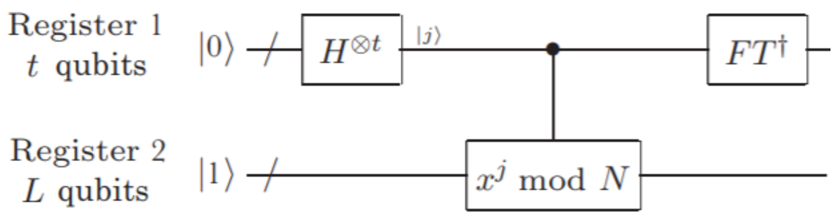


1. **Parameters**
- U - unitary matrix
- t number such that the output of the measurement is an approximation to accurate to t round_up(log(2+1/2*e bits with probability of success at least 1 epsilon.
- wires
2. **Results**:
- transformed wires (not measured)

1.9 QuantumAlgorithms.Order_Finding(wires,wires_x,wires_z, wires_a,wires_b,wires_c,wires_N,wires_t,t,N,y)

Order_Finding is a circuit which uses phase estimation structure to find order r of y modulo N, i.e. such r that $y^r mod N = 1$.

The circuit is $O(n^3)$ efficient.

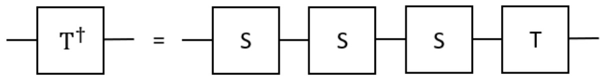


1. **Parameters**
- wires - list of all wires
- wires_x - list of wires of register x
- wires_z - list of wires of register z
- wires_a - list of wires of register a
- wires_b - list of wires of register b
- wires_c - list of wires of register c
- wires_N - list of wires of register N
- wires_t - name or index of a wire of register t
- t - integer, amount of qubits in the register x
- N - integer such that $y^r mod N = 1$
- y - integer such that $y^r mod N = 1$
2. **Results**:
- transformed wires (not measured)

2 Functions from QuantumGates class

2.1 QuantumGates.T_dagger(wires)

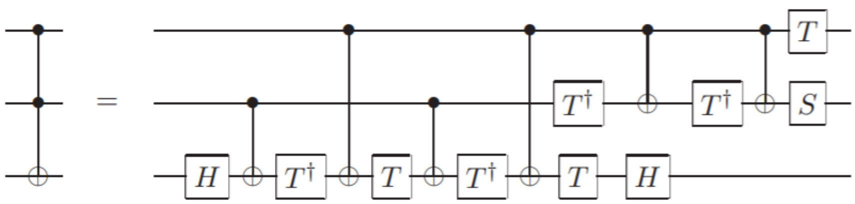
Implements T_dagger (conjugate transform of T)



1. **Parameters**
- wires
2. **Results**:
- transformed wires

2.2 QuantumGates.Toffoli(wires)

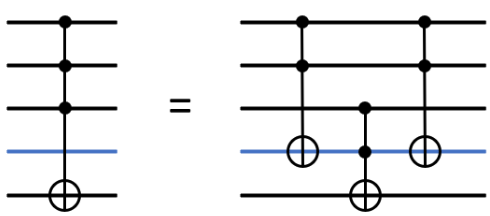
Implements Toffoli gate



1. **Parameters**
- wires
2. **Results**:
- transformed wires

2.3 QuantumGates.Controlled_Toffoli(control_wires,operation_wire,work_wire)

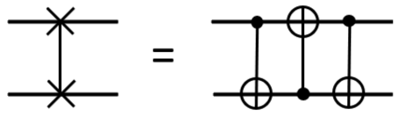
Implements modified Toffoli gate with 3 controls instead of 2. Requires 1 additional work qubit



1. **Parameters**
- control_wires
- operation_wire
- work_wire with zero
2. **Results**:
- transformed control_wires
- transformed operation_wire
- transformed work_wire with zero

2.4 QuantumGates.SWAP(wires)

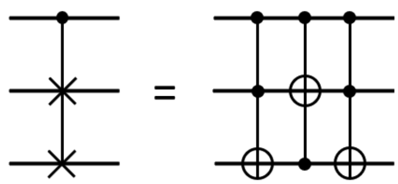
Implements standard 2-wires SWAP-gate



1. **Parameters**
- wires
2. **Results**:
- transformed wires

2.5 QuantumGates.Controlled_SWAP(control_wire,swap_wires)

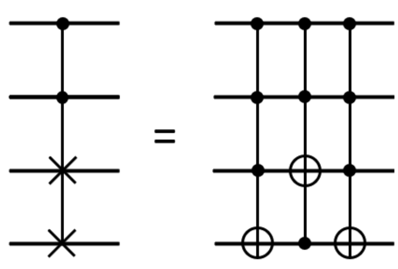
Implements 2-wires SWAP conditional on 1-wire control



1. **Parameters**
- control_wire
- swap_wires
2. **Results**:
- transformed control_wire
- transformed swap_wires

2.6 QuantumGates.Controlled_Controlled_SWAP(control_wires,swap_wires, work_wire)

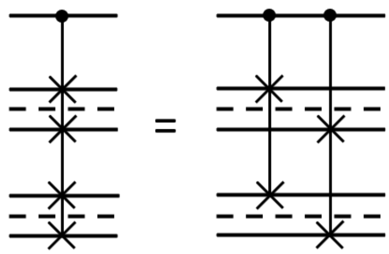
Implements 2-wires SWAP conditional on 2-wires controls. Requires 1 work qubit, because it uses Controlled_Toffoli



1. **Parameters**
- control_wires
- swap_wires
- work_wire
2. **Results**:
- transformed control_wires
- transformed swap_wires
- transformed work_wire

2.7 QuantumGates.Controlled_register_SWAP(control_wire,wires_register_1, wires_register_2)

Implements SWAP for 2 n-wires register conditional on 1-wire control



1. **Parameters**
- control_wire
- wires_register_1
- wires_register_2
2. **Results**:
- transformed control_wire
- transformed wires_register_1
- transformed wires_register_2

2.8 QuantumGates.Controlled_reset_zero_register_to_N(control_wire, wires_zero_register,N)

Implements resetting register with zeros to binary representation of classicaly known number N conditional on 1-wire control. If control == 1, then resulting values in the wires_zero_register are $[N_0, N_1, ..., N_{(n-1)}]$, where $N = N_{n-1} * 2^{n-1} + ... + N_1 * 2^1 + N_0 * 2^0$

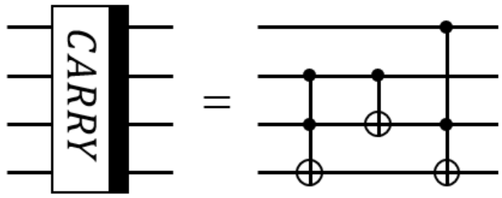
1. **Parameters**
- control_wire
- wires_zero_register with zero
- N - number to be put into wires_zero_register
2. **Results**:
- transformed control_wire
- transformed wires_zero_register with binary representation of N

2.9 QuantumGates.CARRY(wires,inverse=False)

Implements 4-wires carry operation used for ADDER

Setup: $wires[0] = c_i, wires[1] = a_i, wires[2] = b_i, wires[3] = c_{i+1} = |0\rangle$. Operation carries $|1\rangle$ in $wires[3] = c_{i+1}$ if $c_i + a_i + b_i > 1$

Based on Vedral, Barenco, Ekert - "Quantum Networks for Elementary Arithmetic Operations 1996



1. **Parameters**

- wires

- inverse - if passed False, then circuit from the scheme is implemented; if passed True, then circuit with inverse order of all elementary operations with respect to the circuit from the scheme is implemented

2. **Results**:

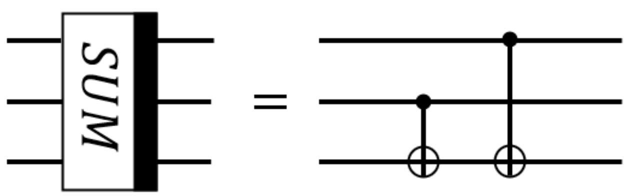
- transformed wires

2.10 QuantumGates.SUM(wires,inverse=False)

Implements 3-wires sum operation used for ADDER

Setup: $wires[0] = a, wires[1] = b, wires[2] = |0\rangle$. Operation makes $wires[2] = a + b \mod 2$

Based on Vedral, Barenco, Ekert - "Quantum Networks for Elementary Arithmetic Operations 1996



1. **Parameters**

- wires

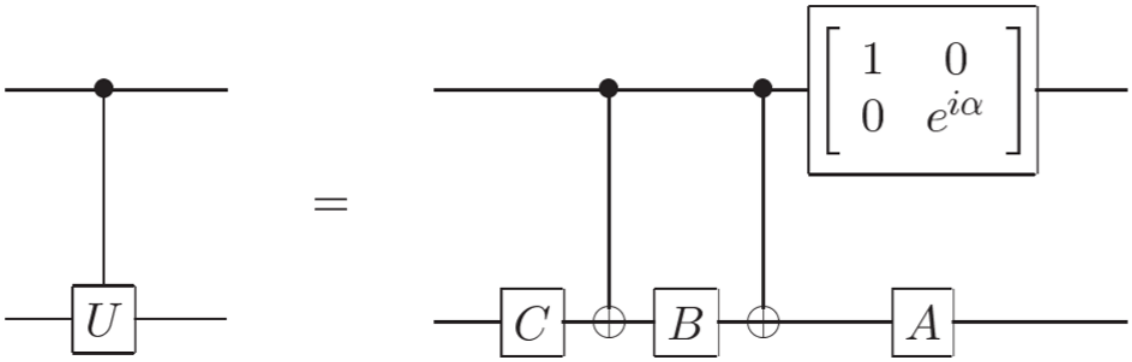
- inverse - if passed False, then circuit from the scheme is implemented; if passed True, then circuit with inverse order of all elementary operations with respect to the circuit from the scheme is implemented

2. **Results**:

- transformed wires

2.11 QuantumGates.Controlled_U_block(alpha,beta,gamma,delta, delta_plus_beta,delta_minus_beta,wires)

Implements controlled-U with 1 control and 1 operation wire, given angles (alpha, beta, gamma, delta) from ZY-decomposition of 2×2 unitary U



$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta) = e^{i\alpha} AXBXC \qquad ABC = I$$

1. **Parameters**
- alpha - angle from ZY-decomposition
- beta - angle from ZY-decomposition
- gamma - angle from ZY-decomposition
- delta - angle from ZY-decomposition
- delta_plus_beta = delta + beta
- delta_minus_beta = delta - beta
- wires
2. **Results**:
- transformed wires

2.12 QuantumGates.CR_k(control_wire,operation_wire,inverse=False)

Implements 2-qubit controlled R_k - phase shift gate which is used in QFT.

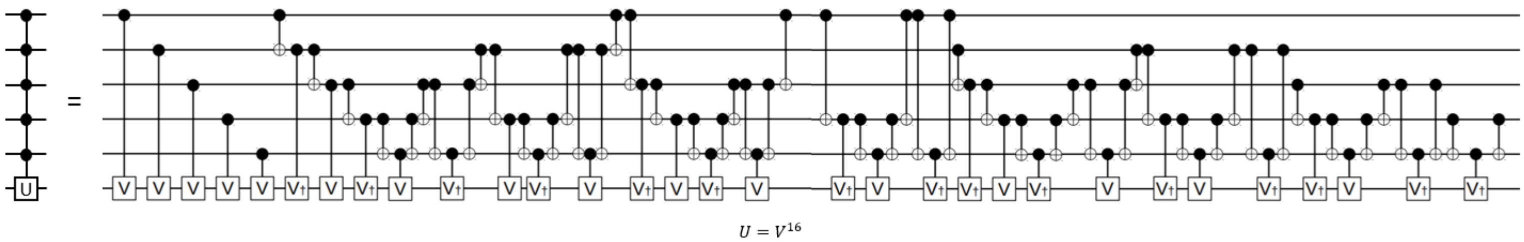
R_k has matrix form $\begin{pmatrix} 1 & 0 \\ 0 & \mathrm{e}^{2 \cdot \pi \cdot i / 2^k} \end{pmatrix}$

If inverse == True, then the function implements 2-qubit controlled R_k_dagger - phase shift gate which is used in inverse QFT R_k_dagger has matrix form $\begin{pmatrix} 1 & 0 \\ 0 & \mathrm{e}^{-2 \cdot \pi \cdot i / 2^k} \end{pmatrix}$

1. **Parameters**
- control_wire
- operation_wire
- inverse - if passed False, then CR_k is implemented; if passed True, conjugate transpose of CR_k is implemented
2. **Results**:
- transformed control_wire
- transformed operation_wire

2.13 QuantumGates.C_U(U,control_wires,operation_wire)

Implements C_n_U given arbitrary 2×2 U and arbitrary amount of control wires (up to 5 controls). Scheme is provided for 5 controls

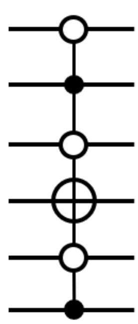


1. **Parameters**
- U - unitary 2×2 matrix
- control_wires
- operation_wire
2. **Results**:
- transformed control_wires
- transformed operation_wire

2.14 QuantumGates.gray_code_C_X(gray_code_element,changing_bit,wires)

Implements circuit block which corresponds to a binary string gray_code_element with (integer) index changing_bit of a changing bit - see Nielsen Chuang p.192

Scheme is the example of gray_code_C_X circuit for gray_code_element = '010001' and changing_bit = 3



1. **Parameters**
- gray_code_element - binary string
- changing_bit - int (index of wire to which controlled X operation is implemented)
- wires
2. **Results**:
- transformed wires

2.15 QuantumGates.Two_level_U(U,non_trivial_indices,wires)

Implements $(2^n) \times (2^n)$ two-level unitary matrix U circuit, where n = len(wires). This is a building block for arbitrary unitary U operation circuit QuantumAlgorithms.U_n().

1. **Parameters**
- U - $(2^n) \times (2^n)$ two-level unitary matrix U, where n = len(wires)
- non_trivial_indices - pair of integer indices which denote non-trivial entries in U
- wires
2. **Results**:
- transformed wires

2.16 QuantumGates.controlled_Two_level_U(U,non_trivial_indices, control_wire,operation_wires)

Controlled version of QuantumGates.Two_level_U(), with 1 control

1. **Parameters**
- U - $(2^n) \times (2^n)$ two-level unitary matrix U, where n = len(wires)
- non_trivial_indices - pair of integer indices which denote non-trivial entries in U
- control_wire
- operation_wires
2. **Results**:
- transformed control_wire
- transformed operation_wires

3 Functions from ClassicalOperations class

3.1 ClassicalOperations.states_vector(wires)

Prints out list of states in a computational basis given wires list

1. **Parameters**
- wires - list of wires in a circuit
2. **Returns**:
- states_vector - list of strings in a form '|001..01>' corresponding to all possible computational basis states

3.2 ClassicalOperations.int_to_binary_array(n,reverse=False,bits=False)

Returns np.array with bitwise binary representation of a given number n

1. **Parameters**
- n - integer number
- reverse - boolean; if option reverse=True reverses array so that the first bit is for 2^0 and the last is for 2^{k-1}
- bits - boolean or integer; if option bits is set to a number, then the length of resulting array is bits
2. **Returns**:
- array of integer zeros and ones corresponding to a binary representation of a given number n

3.3 ClassicalOperations.get_non_trivial_indices(Two_level_U_list)

Returns list of pairs of indices of non-trivial entries in two-level matrices. List of two-level matrices should be passed to the function as an argument. Two-level matrices are from decomposition of unitary matrix into product of two-level matrices. See Nielsen Chuang p.189

1. **Parameters**
- Two_level_U_list - list of two-level matrices
2. **Returns**:
- list of pairs with indices of non-trivial entries in the matrices from Two_level_U_list

3.4 ClassicalOperations.Gray_code(a,b,n)

Creates list with Gray code and changing bits for every step given binary a and b, and numer n where n is a length of a binary string.

1. **Parameters**
- a - str - start binary string
- b - str - end binary string
- n - integer - number of bits in a binary representation
2. **Returns**:
- list [gray_code, changing_bit], where gray_code is a list of str binary steps of gray code and changing_bit is a list of changing bits for every respective step

3.5 ClassicalOperations.gcd(a,b)

Euclid's algorithm - finds greater common divider for integers a and b.

Note: doesn't work correctly for too big numbers (because a%b and int(a/b) do not work).

1. **Parameters**
- a - integer
- b - integer
2. **Returns**:
- integer - greater common divider of a and b

3.6 ClassicalOperations.diophantine_equation_auxiliary(k_list,alpha,beta,i)

Auxiliary recursive function for finding alpha and beta in r_n = alpha * a + beta * b, where n: r_n = gcd(a,b)

Algorithm should be initialized with alpha = 1, beta = -k_(n-2), i = n-2, where n: r_n = gcd(a,b). before algorithms' execution, array of k should be defined. i denotes level in Euclid's algorithm

Note: doesn't work correctly for too big numbers (because a%b and int(a/b) do not work)

1. **Parameters**

- k_list - list of integers, where k_list[i] is such that r_i = k_list[i] * r_(i+1) + r_(i+2), i.e. k_list is a result of forward part of solving diophantine_equation
- alpha - integer from recursive representation of r_n = alpha*a + beta*b
- beta - integer from recursive representation of r_n = alpha*a + beta*b
- i - integer - level in Euclid's algorithm

2. **Returns**:

- list of two integers - [alpha,beta], which are for the next step of recursive algorithm

3.7 ClassicalOperations.diophantine_equation(a,b)

Solves diophantine equation, i.e. given a,b returns x,y such that ax + by = gcd(a,b)

During the function's execution, in the forward part Euclid's algorithm produces set of values (k_i,r_i), where r_i = k_i * r_(i+1) + r_(i+2), i goes from 0 to n and then in the backward part algorithm uses recursive ClassicalOperations.diophantine_equation_auxiliary()

Note: doesn't work correctly for too big numbers (because a%b and int(a/b) do not work)

1. **Parameters**

- a - integer from ax + by = gcd(a,b)
- b - integer from ax + by = gcd(a,b)

2. **Returns**:

- list of two integers - [x,y] from ax + by = gcd(a,b)

3.8 ClassicalOperations.modular_multiplicative_inverse(a,N)

Finds modular multiplicative inverse of a modulo N using diophantine_equation

1. **Parameters**

- a - integer
- N - integer

2. **Returns**:

- integer a^{-1} such that $a^{-1} * a = 1 mod N$

3.9 ClassicalOperations.matrix_power(U,power=1/2)

2×2 matrix to the custom power such that |power| < 1 using eigenvectors and eigenvalues

1. **Parameters**

- U - 2×2 matrix
- power - float such that |power| < 1

2. **Returns**:

- matrix U^{power}

3.10 ClassicalOperations.matrix_natural_power(U,power)

Arbitrary matrix to the natural power

1. **Parameters**
- U - 2×2 matrix
- power - integer - natural number
2. **Returns**:
- matrix U^{power}

3.11 ClassicalOperations.ZY_decomposition_angles(U)

Given 2×2 unitary matrix U, function returns angles alpha, beta, delta and gamma of ZY decomposition U = np.exp(1j * alpha) * RZ_beta.dot(RY_gamma).dot(RZ_delta)

1. **Parameters**
- U - 2×2 unitary matrix
2. **Returns**:
- dictionary 'alpha':alpha, 'beta':beta, 'delta':delta, 'gamma':gamma

3.12 ClassicalOperations.U_given_ZY_angles(alpha,beta,gamma,delta)

Given angles of ZY decomposition, function returns corresponding 2x2 matrix U = np.exp(1j * alpha) * RZ_beta.dot(RY_gamma).dot(RZ_delta)

1. **Parameters**
- alpha - float -
- beta
- delta
- gamma
2. **Returns**:
- U = np.exp(1j * alpha) * RZ_beta.dot(RY_gamma).dot(RZ_delta)

3.13 ClassicalOperations.Two_level_unitary_decomposition(U)

Given U, function returns its two-level unitary decomposition. See Nielsen Chuang p.189

Note that U = decomposition_list[0] * ... * decomposition_list[n-1]

1. **Parameters**
- U - unitary matrix
2. **Returns**:
- decomposition_list - list of matrices such that U = decomposition_list[0] * ... * decomposition_list[n-1]