Abstract

Reversible computing has gained increasing attention as a prospective solution for energy dissipation, particularly in quantum computing. As the first practical reversible logic gate using adiabatic superconducting devices, the reversible quantum-flux-parametron (RQFP) has been experimentally demonstrated in logical and physical reversibility. However, the circuit design of RQFP logic poses enormous challenges due to its distinctive logic function and structure. Furthermore, the circuit scale severely restricts the applicability of the existing exact logic synthesis method for RQFP logic. Therefore, this paper proposes RCGP, an automatic synthesis framework based on efficient Cartesian genetic programming, to generate large RQFP logic circuits. RCGP considers the characteristics of RQFP logic circuits to minimize the number of gates and garbage outputs. Meanwhile, RCGP combines circuit simulation with formal verification to assess the functional equivalence between the parent and its offspring. Experimental results on reversible logic benchmarks demonstrate the effectiveness of RCGP.

1 Introduction

The requirement for reversible computing arises from the fundamental limitations of traditional irreversible logic systems in terms of energy dissipation. In conventional logic systems, such as complementary metal-oxide-semiconductor (CMOS) logic, irreversible operations result in information loss and energy dissipation in the form of heat. According to the Landauer principle in 1961 [1], every bit loss of information is involved in a loss of $k_B T \ln 2$ J of energy, where $k_B$ is the Boltzmann constant and $T$ is the temperature of the system. This heat dissipation becomes significant as the rate of the processor frequency increases. Therefore, reversible computing has gained significant attention as a promising approach to minimizing energy dissipation in logic operations. It can theoretically produce nearly energy-free computation systems by preserving the loss of information [2]. This approach also holds particular significance in quantum computing due to the inherent reversibility of quantum operations [3].

However, the practical implementation of reversible computing presents significant challenges. It requires both logical and physical reversibility, as well as the development of ultra-low power logic devices [4]. Takeuchi et al. proposed the reversible quantum-flux-parametron (RQFP) [5], the first practical reversible logic gate using adiabatic superconducting devices. The logical and physical reversibility of RQFP has also been validated through experimental demonstrations [5, 6]. Moreover, an RQFP logic full adder was designed and fabricated by Yamae et al. [7], further revealing the reversibility and feasibility of RQFP logic circuits. Therefore, RQFP logic gains increasing attention from researchers [8, 9].

However, the design of RQFP logic circuits is complicated due to the lack of automated design tools [8]. The RQFP logic gate is implemented by adiabatic quantum-flux-parametron (AQFP) [10], an energy-efficient superconductor logic element based on the quantum flux. Fig. 1(a) shows the structure of a normal RQFP logic gate composed of three AQFP splitter gates and three AQFP majority gates. So, the RQFP logic gate has three functional outputs, $I, O, S$. A novel extension of RQFP logic gates is AQFP majority, where symbols employed include $\cdot$ for logical conjunction, $+$ for logical disjunction, and $-$ for logical negation.

![Functional schematics of RQFP logic gate](image)

**Figure 1:** Functional schematics of (a) RQFP logic gate, (b) MCT gate, and (c) MCF gate, where symbols employed include $\cdot$ for logical conjunction, $+$ for logical disjunction, and $-$ for logical negation.
As multi-controlled NOT and multi-controlled SWAP gates, respectively. It is clear that their output functions mainly focus on the last one or two output ports and realize the XOR-sum-of-products. Furthermore, RQFP logic must also satisfy specific constraints, including the same fan-out limitation and path-balancing requirement as AQFP logic [13–15], since RQFP logic is realized through AQFP logic. Consequently, the logic synthesis methods of conventional reversible logic do not apply to RQFP logic due to these distinctions in logic function and structure. In addition, the energy dissipation of RQFP logic circuits is significantly affected by the number of gates and garbage output. Exact logic synthesis [15] is the only existing method for generating RQFP logic circuits, but it is severely limited by the circuit scale, which has been demonstrated in the experiment. Therefore, the study of practical logic synthesis methods of RQFP logic circuits is vital.

Considering these above challenges, this paper proposes RCGP, an efficient Cartesian genetic programming (CGP)-based automatic synthesis framework for RQFP logic circuits, which aims to optimize the number of RQFP logic gates and garbage outputs. In summary, this paper makes the following contributions:

- This paper proposes a complete logic synthesis framework for RQFP logic circuit generation from RTL descriptions.
- The proposed algorithm employs a dynamic CGP encoding to present an RQFP logic circuit and integrates circuit simulation with formal verification to evaluate the functional equivalence between the parent circuit and its offspring.
- Furthermore, three mutations are proposed to effectively and efficiently generate a legal RQFP logic circuit.
- Experimental results of RCGP on large RevLib circuits [16] and reversible reciprocal circuits [17] show that the number of RQFP logic gates and the number of garbage outputs are significantly reduced by 32.38% and 59.13% on average, respectively.

### 2 Preliminaries

#### 2.1 Reversible Quantum-Flux-Parametron Logic

As a superconductor logic gate, the RQFP logic gate exhibits both logical and physical reversibility [6]. Fig. 1(a) illustrates its structure, composed of three three-output AQFP splitter gates (S₁, S₂, S₃) and three three-input AQFP majority gates (M₁, M₂, M₃). Similar to AQFP logic, excitation currents Iₓ₁ and Iₓ₂ are required to drive these splitter gates and majority gates, respectively. The outputs of the RQFP logic gate source from the three three-input AQFP majority gates, making RQFP logic possess a compact logic representation. In a normal RQFP logic gate, an inverter is configured before the fixed-position input of each AQFP majority. This produces a one-to-one correspondence between the inputs and outputs of the RQFP gate to make the RQFP logic gate logically reversible [5]. Since inverters can be freely integrated into any input of each AQFP majority gate, the functionality of RQFP logic gates can be extended, which means that the function of each RQFP logic gate can be configured by the inverter setting.

Moreover, since the RQFP logic gate is realized by AQFP logic gates, it also inherits the inherent characteristics [13–15] of AQFP logic. In AQFP logic, each output of each AQFP logic gate can only drive one successor, i.e., the single fan-out limitation. The insertion of AQFP splitter gates can eliminate this limitation. Similarly, the RQFP splitter can be constructed by the introduction of constant inputs to satisfy this limitation [15]. For example, a 1-to-3 splitter can be achieved by \( R(1, a, 0) = \{ M(1, a, 0), M(1, a, 0), M(1, a, 0) \} = \{ a, a, a \} \). Additionally, all inputs to each gate in AQFP logic must possess the same clock phases, i.e., the path-balancing requirement. This requirement can be satisfied by the insertion of AQFP buffers. Similarly, two cascaded AQFP buffers can be used to construct an RQFP buffer [15]. Meanwhile, the RQFP inverter can also be constructed by inserting an inverter in the RQFP buffer. Consequently, RQFP logic circuits must perform RQFP buffer and splitter insertion to meet these requirements, thereby ensuring proper circuit operation.

### 3 RCGP

RQFP logic exhibits unique characteristics that present significant challenges in the design of RQFP logic circuits. RQFP logic gate has three output ports, each capable of independently functioning as a majority function. Additionally, RQFP logic circuits require buffer and splitter insertion to meet fan-out and path-balancing requirements. These distinct features of RQFP logic contribute to the complexity of the RQFP logic circuit design. To end this, this paper proposes an end-to-end circuit generation framework for RQFP logic, as shown in Fig. 2. Unlike the existing exact synthesis method [15] that takes truth tables as input, the proposed RCGP divides the entire process into several phases to gradually optimize the RQFP logic.
To support the RTL description inputs with multiple standard formats, such as the Verilog format, .aig format, and .blif format, RCGP first automatically integrates current popular open-source logic synthesis tools to process them, including ABC [25] and mockturtle [26]. Both ABC and mockturtle can optimize the AND-inverter graph (AIG)-based network, and mockturtle can also optimize the majority-inverter graph (MIG)-based network. The selection of both these tools and corresponding methods is configurable. After logic synthesis using these tools, AIG or MIG-based optimized networks are directly converted to the RQFP logic netlist. Next, RQFP splitters are inserted into the initial RQFP logic network to meet the fan-out limitation. In this way, an initial RQFP logic network is obtained. The RQFP-oriented CGP is proposed to optimize this initial RQFP logic network and minimize the number of RQFP logic gates and garbage outputs. Finally, after RQFP buffer insertion, the final RQFP logic circuit can be obtained.

### 3.1 Initialization

To support the RTL design descriptions with multiple standard formats, such as the Verilog format, .aig format, and .blif format, RCGP first automatically integrates current popular open-source logic synthesis tools to process them, including ABC [25] and mockturtle [26]. Both ABC and mockturtle can optimize the AND-inverter graph (AIG)-based network, and mockturtle can also optimize the majority-inverter graph (MIG)-based network. The selection of both these tools and corresponding methods is configurable. After logic synthesis using these tools, AIG or MIG-based optimized networks can be obtained. Since the introduction of constant inputs can make the RQFP logic gate realize AND, OR, NOT, and majority functions easily, the AIG or MIG-based network can directly be converted into the RQFP logic netlist. Take the AND function as an example. Introducing one constant \( 1 \) can realize it, i.e., \( R(a, b, 1) = \{M(\overline{a} \cdot b, 1), M(a, \overline{b}, 1), M(a, b, 0)\} = \{\overline{a} \cdot b + a \cdot \overline{b}, a + \overline{b}, ab\} \), where the last output achieves the AND function.

After generating the initial RQFP logic netlist, a lot of multiple fan-outs may exist. To mitigate the complexity of RQFP logic optimization, RQFP splitters are inserted into the initial RQFP logic netlist ahead of time to meet the fan-out limitation. In this way, the initialization process of RQFP logic circuits is completed.

### 3.2 CGP-based Optimization

#### 3.2.1 CGP Encoding and Evaluation

To optimize the RQFP logic circuit, CGP encoding is first introduced to represent the RQFP logic circuit. CGP usually encodes a candidate solution using an integer array consisting of \( n_C \cdot n_R \) programmable nodes, where \( n_C \) and \( n_R \) determine the number of columns and rows, respectively. As for the circuit optimization, a linear form of CGP is usually preferred, i.e., \( n_R = 1 \). Each programmable node has a fixed number of inputs \( n_I \) and outputs \( n_O \) and can implement one of \( n_F \) predefined primitive functions. So, since each node represents an RQFP logic gate in RCGP, both \( n_I \) and \( n_O \) are set to 3. Besides, since the function of each RQFP logic gate depends on its inverter configuration, there are \( 2^9 = 512 \) kinds of functions, i.e., \( n_F = 512 \). Any input to each node can be connected to the output of a node placed in the previous \( n_I \) columns or to one of \( n_{pi} \) primary inputs. Due to the complicated evaluation, the cycle is not allowed in the standard version of CGP, which means there is no feedback to the node from its successors. Therefore, the candidate solution is encoded as a CGP chromosome.
with \( n_L = n_C \times n_R \times (n_1 + 1) + n_{po} \) integers, where \( n_{po} \) is the number of primary outputs of the circuits.

Since the positions of input ports in each RQFP logic gate are fixed and are directly related to the gate function output, the CGP encoding needs to adjust to apply to RQFP logic. Take a 2-to-4 decoder as an example. Fig. (3(a)) illustrates its CGP encoding, where \( n_C = 4 \) and \( n_R = 1 \). The 2-to-4 decoder has \( n_{po} = 2 \) primary inputs \( x_0 \) and \( x_1 \) and \( n_{po} = 4 \) primary outputs \( y_0, y_1, y_2 \) and \( y_3 \). Considering the requirement of RQFP splitters for constant inputs, constant \( 1 \) is introduced and indexed to 0. The primary inputs are indexed from 1 to \( n_{pi} \). The integer substring within each pair of parentheses in the long green string at the bottom is the CGP encoding of the corresponding RQFP node. For example, \((5, 4, 0, 101-000-000)\) is the CGP encoding of the second RQFP logic gate, where \(5, 4, 0\) denotes the interconnection of its input ports with the output ports indexed 5, 4, 0 (the constant input), respectively, and \(101-000-000\) denotes the inverter configuration in front of its three AQFP majority gates. The inverter configuration is represented by an integer with 9 bits, each of which indicates whether an inverter exists in the corresponding input port. For instance, \(100\) indicates whether an inverter exists before the second input port of each AQFP majority gate within the second RQFP logic gate. Besides, the last item \((6, 13, 14)\) represents the indexes of output ports connected to primary outputs, where \(10\) represents that the primary output \( y_1 \) is connected to the second output port of the third RQFP logic gate.

After encoding a candidate solution, its evaluation is also required. The fitness value evaluation of the CGP chromosome in RCGP contains two phases. The first is the function evaluation, which calculates the success rate of the simulation-based equivalence checking. The second is the performance evaluation, which calculates the number of RQFP logic gates and garbage outputs. The detail is defined as follows:

1. Only when the success rate reaches 100%, the performance evaluation will be performed, thereby ensuring the function legitimacy of the solution.
2. Then, the performance evaluation gives priority to ensuring the optimal number of RQFP logic gates.
3. In addition, while ensuring the optimal number of RQFP logic gates and garbage outputs, RCGP will also consider reducing the number of RQFP buffers that need to be inserted for the path balancing requirement.

3.2.2 Mutation. After constructing the CGP chromosome, CGP mutation must occur to produce its offspring. Point mutation is typically preferred due to its high efficiency. Point mutation randomly modifies up to \( m \) genes (integers) of a parent genotype to create an offspring, where \( m \) depends on the mutation rate \( \mu, \mu \in [0, 1] \). To ensure that each gene has a chance to be modified randomly within each mutation, RCGP sets the maximum of \( m \) to \( \mu \times n_L \), where \( n_L \) is the length of the chromosome.

Considering the CGP encoding, a single mutated gene causes either reconnection of a node, reconnection of a primary output, or change in the inverter configuration of a node. Due to the fan-out limitation of RQFP logic, there are two situations for the node reconnection. Notably, the initialization process has ensured the single fan-out limitation of the parent genotype.

Algorithm 1: The flow of CGP-based optimization.

\[
\text{Input: Initial chromosome } c, \ \text{total number } N \ \text{of generations, mutation rate } \mu, \ \text{number } \lambda \ \text{of offspring.}
\]

\[
\text{Output: Optimized chromosome.}
\]

1. calculate the functional fitness \( f \) of \( c \).
2. \( f_n = f, c_n = c, n_r = \infty, n_g = \infty, n_b = \infty \).
3. if \( f = 1 \) then
   4. remove useless nodes to shrink \( c \) and update \( c \) to \( c_n \).
   5. calculate the number \( (n_r, n_g, n_b) \) of gates, garbage outputs, and buffers in \( c_n \).
6. for \( i = 1 \rightarrow N \) do
   7. \( c = c_n \).
   8. for \( j = 1 \rightarrow \lambda \) do
   9. perform the mutation on \( c \) to generate a offspring \( c' \).
   10. calculate the functional fitness \( f' \) of \( c' \).
   11. if \( f' = 1 \) then
   12. remove useless nodes to shrink \( c' \) and update \( c' \).
   13. calculate the number \( (n'_r, n'_g, n'_b) \) of gates, garbage outputs, and buffers in \( c' \).
   14. if \( (n_r, n_g, n_b) > (n'_r, n'_g, n'_b) \) then
   15. \( (f_n, c_n, n_r, n_g, n_b) = (f'_n, c'_n, n'_r, n'_g, n'_b) \).
   16. else if \( f' > f_n \) then
   17. \( (f_n, c_n) = (f'_n, c'_n) \).
18. return \( c_n \).

(1) When the output port corresponding to the mutated value of a gene has been connected to another gene, RCGP will swap the values of these two genes. As shown in Fig. (3(a)), assume that ‘9’ in the last second item ‘\((9, 8, 3, 000-110-111)\)’ mutates to ‘8’, and then the output port corresponding to ‘8’ is already connected to the second input port of the last node. Therefore, the swap operation is executed, i.e., ‘\((9, 9, 3, 000-110-111)\)’.

(2) When the output port corresponding to the mutated value of a gene is the constant input \( 1 \) or has no output, this value is directly assigned to this gene. After mutating to ‘\((9, 9, 3, 000-110-111)\)’, assume that ‘9’ continues to mutate to ‘0’, the genotype of the last node becomes ‘\((8, 0, 3, 000-110-111)\)’. This means that the second input port of the last node is connected to the constant input 1, as shown in Fig. (3(b)).

For the reconnection of a primary output, its gene is directly updated to the mutated value, such as y1 in Fig. (3(b)) changed from 10 to 7. Furthermore, for the change of an inverter configuration \( f \), the mutation will produce an integer \( \beta \in [0, 9] \) to update \( f \) to \( f' = f \oplus (1 \ll \beta) \), which means that the \((1 \ll \beta)\)th inverter is inserted or removed. As shown in Fig. (3(a)), initial inverter configuration of the second node is ‘\(101-100-000\)’ \((352)\) and then is updated to ‘\(101-011-000\)’ \((344 = 352 \oplus ((1 \ll 3) + (1 \ll 4) + (1 \ll 5))\)’ through three mutations, as shown in Fig. (3(b)). In this way, the output function of the second majority in the second node is updated.

3.2.3 Shrink. After CGP mutation, some useless nodes may exist, such as the third node ‘\((0, 0, 7, 001-101-101)\)’ in Fig. (3(b)). This directly indicates a primary advantage of CGP encoding that even if the size of the chromosome is fixed, the size of the phenotype is variable since some nodes need not be used. Since RCGP aims to minimize
3.3 RQFP Buffer Insertion

Following the CGP-based optimization, it is possible that certain RQFP logic gates within the generated RQFP logic circuit may not satisfy the path balancing requirement. Consequently, the insertion of RQFP buffers becomes necessary to ensure uniform clock phases for all inputs to each gate. That is, a corresponding number of RQFP buffers must be inserted into each edge according to the clock phase difference between its connected RQFP logic gates. Fig. 3(d) shows the buffer insertion result of the RQFP logic circuit shown in Fig. 3(c) generated for the 2-4 decoder.

4 Experimental Results

The proposed automatic RQFP logic synthesis framework RCGP was implemented in C++ and Python. The experiments used the RevLib benchmark circuits [16] and the reversible reciprocal circuits [17] to evaluate RCGP. The logic synthesis phase of RCGP first used the "resyn2" command in the ABC [25] to obtain an optimized AIG network and then employed the "aqfp_resynthesis" command in the mockturtle to resynthesize it into an AQFP-oriented MIG network. The "aqfp_resynthesis" command implements a state-of-the-art MIG-based logic optimization method [27] of AQFP logic. Besides, the number of generations and the mutation rate in RCGP was 50,000,000 and 1, respectively. The number of RQFP logic gates in the generated initial RQFP logic netlist. \( n_c \) kept the same as the number of RQFP logic gates. The experiments were executed on the machine with Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz and 256.0 GB memory running CentOS 7.

There were two baselines to compare with RCGP. The first was designed by a heuristic method. As shown in Fig. 2, after the initialization and RQFP splitter insertion, the baseline directly carries out the RQFP buffer insertion to meet the path-balancing requirement, thereby generating RQFP logic circuits. The second was the existing exact logic synthesis method [15] using Z3 solver. According to the input truth table, it can directly generate the RQFP logic circuit with a given number of RQFP logic gates and garbage outputs. To ensure the consistency of each experimental result in handling the path-balancing requirement for primary inputs and primary outputs of the generated circuit, both primary inputs and primary outputs require RQFP buffer insertion so that they are in the same clock stage, respectively.

Table 1 shows the experimental results on small circuits from the RevLib benchmark. The "Original" part shows the attributes of input circuits, including the number of primary inputs, the number of primary outputs, and the lower bound \( g_{lb} = \max(0, n_{pi} - n_{po}) \) of garbage outputs. The "Initialization" part shows the results of the first baseline method. \( n_r \) represents the number of RQFP logic gates, and \( n_g \) represents the number of RQFP buffers inserted for the path-balancing requirement. "JJs" represents the number of Josephson junctions (JJs). Since JJs are a critical component of AQFP circuits and are used to create the fluxons used for computation, their number can estimate the complexity and energy efficiency of a given AQFP circuit. So, the number of JJs can also be used as a cost metric of RQFP logic circuits realized by AQFP logic. Since both a buffer and a splitter have 2 JJs, and a 3-input MAJ has 6 JJs in current AQFP logic circuits, the numbers of JJs for each RQFP logic gate and RQFP buffer are 2 and 4, respectively. \( n_d \) and \( n_g \) represent the circuit depth and the number of garbage outputs in generated RQFP logic circuits, respectively.

The "Exact logic synthesis" part shows the results of the exact logic synthesis method [15] for RQFP logic, where "T (s)" represents the runtime of the method and its unit is the second. The "RCGP" part shows the results of the proposed algorithm RCGP. Compared to the first baseline, the proposed RCGP significantly reduced the number of RQFP logic gates, JJs, and garbage outputs, specifically by 50.80%, 43.53%, and 71.55%, respectively. Meanwhile, although the exact logic synthesis can generate the RQFP logic circuit with the optimal number of RQFP logic gates and garbage outputs, the results of RCGP can be near even up to its results while using less runtime. Besides, the exact logic synthesis could not also find the solution for circuits "decoder_3_8", "graycode4", and "mux4" under sufficiently given 240,000 seconds. Therefore, these experimental results show that the proposed RCGP can generate RQFP logic circuits with a near-optimal number of RQFP logic gates and garbage outputs on small benchmark circuits.
Table 2: Experimental results on large circuits from the RevLib benchmark [16] and the reversible reciprocal circuits [17].

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Original</th>
<th>Initialization</th>
<th>Exact logic synthesis</th>
<th>RCGP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(n_{pi})</td>
<td>(n_{po})</td>
<td>(g_{lib})</td>
<td>(n_r)</td>
</tr>
<tr>
<td>4_graycode6_7</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>graycode6_11</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>mod5adder_66</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>105</td>
</tr>
<tr>
<td>hwb8_64</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>127</td>
</tr>
<tr>
<td>intdiv4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>intdiv5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>intdiv6</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>78</td>
</tr>
<tr>
<td>intdiv7</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>128</td>
</tr>
<tr>
<td>intdiv8</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>326</td>
</tr>
<tr>
<td>intdiv9</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>483</td>
</tr>
<tr>
<td>intdiv10</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>833</td>
</tr>
</tbody>
</table>

\(\because\) represents that the exact logic synthesis method cannot find a feasible solution within 240,000 seconds.

Table 2 shows the experimental results on large circuits from the RevLib benchmark and reversible reciprocal circuits. It is evident that exact logic synthesis could not find the solution of any test case under a given sufficient time, further enhancing the essentiality of RCGP. In addition, compared to the first baseline, the proposed RCGP has a significant reduction in the number of RQFP logic gates and the number of garbage outputs, specifically by 32.38% and 59.13%, respectively. Therefore, these experimental results demonstrate that the proposed RCGP can effectively and efficiently optimize large RQFP logic circuits regarding the number of gates and garbage outputs.

5 Conclusion

This paper proposed RCGP, a Cartesian genetic programming-based automatic synthesis framework for RQFP logic. RCGP employs CGP encoding to present the RQFP logic circuit and combines circuit simulation with formal verification to assess the functional equivalence between the parent and offspring. After the RQFP-oriented mutation, the RQFP logic circuit can be optimized for the number of gates and garbage outputs. The experimental results on the RevLib benchmark circuits and reversible reciprocal circuits show the effectiveness of RCGP on the generation of RQFP logic gates. Furthermore, the experimental results demonstrated the applicable limitation of exact logic synthesis on larger RQFP logic circuits, further revealing the significance of the proposed RCGP.

Acknowledgments

The research work described in this paper was conducted in the JC STEM Lab of Intelligent Design Automation funded by The Hong Kong Jockey Club Charities Trust and was supported in part by The Research Grants Council of Hong Kong SAR (No. CUHK14207523); in part by the European Union’s Horizon 2020 research and innovation programme (DA QC, No. 101001318); and in part of the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the HighTech Agenda Bayern Plus.

References