JCTC Journal of Chemical Theory and Computation

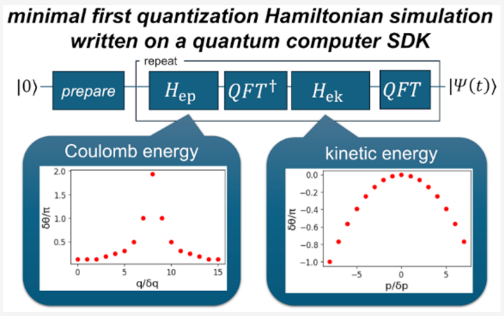
pubs.acs.org/JCTC Article

Chemical Reaction Simulator on Quantum Computers by First Quantization (II)—Basic Treatment: Implementation

Hideo Takahashi,* Tatsuya Tomaru, Toshiyuki Hirano, Saisei Tahara, and Fumitoshi Sato*

Cite This: J. Chem. Theory Comput. 2024, 20, 9290−9320 Read Online

ACCESS | Metrics & More | Article Recommendations



ABSTRACT: Chemical simulation is a key application area that can leverage the power of quantum computers. A chemical simulator that implements a grid-based first quantization method has promising characteristics, but an implementation fully in quantum circuits seems to have not been published. Here, we present "crsQ" (chemical reaction simulator Q), which is a quantum circuit generator that generates such a chemical simulator. The generated simulator is capable of antisymmetrization of the initial wave function and time-evolution of the wave function based on the Suzuki−Trotter decomposition. The potential energy term of the Hamiltonian is implemented using arithmetic gates, such as adders, subtractors, multipliers, dividers, and square roots. Circuit diagrams and output samples are shown. The number of qubits in the circuits scales on the order of $O(\eta \log \eta)$, where η is the number of electrons. Each component of the generated circuit was verified in unit tests. Along with this development, we designed frameworks to ease the development of large-scale circuits, namely, a temporary qubit allocation framework and an abstract syntax tree framework for arithmetic formulas. These frameworks are expected to be useful in large-scale quantum circuit generators.

1. INTRODUCTION

This study is the second part of a multipart report on the development of a quantum-computer-based simulator for chemical reactions. While the first part describes the simulator circuit design theoretically,[1] this second part describes the implementation of circuits on a quantum computer software development kit (SDK). The simulator implements a grid-based first-quantization method.[1−4] Such a simulator consists of a circuit for preparing the initial state of the wave function, a circuit for evolving the wave function in time, and a circuit for measurement. The time evolution circuit performs the Hamiltonian simulation by using arithmetic operations implemented as quantum circuits that operate on the values in a superposition stored in the qubits. Despite the numerous theoretical studies in the literature, a software implementation that includes all of these elements does not seem to have been published yet. In particular, the preceding studies on chemical simulators running on a quantum computer used simple circuits that lacked the elements needed for handling chemical reactions[2,5] or used a quantum computer simulator equipped with a "bespoke" (artificial) quantum gate that calculated a significant portion of the simulation by using classical computer code as if such a function were available as a built-in gate.[6] Such a simplification or optimization was chosen in order to make the circuits runnable on quantum computer simulators. As a result, the resulting chemical reaction simulator is not fully implemented in terms of the (actual) quantum gates.

Researchers could benefit from a complete implementation of quantum circuits in multiple ways: it could be used as packaged application software, a learning resource, a basis for implementing improvements, or as a basis for comparing improved versions of the circuits. The goal of this study was thus to combine the essential elements of the previous studies and describe and implement the overall circuit of a minimum essential chemical reaction simulator. The elements comprising the simulator are described in a separate paper, which is the first part of this report.[1] In this second part, we describe the implementation aspects of the circuits on Qiskit, a quantum computer SDK from IBM.[7] The goal is to implement the entire simulator on Qiskit and to verify the logical correctness of its circuits by testing the components individually in unit tests. In particular, this report describes the implementation of the state preparation circuit and the time evolution circuit.

Since the hardware requirements for the generated chemical simulator circuit cannot be met with what is available today, the simulator as a whole cannot be run on existing quantum computers or quantum computer simulators. What we have

Received: June 2, 2024
Revised: August 28, 2024
Accepted: September 20, 2024
Published: October 25, 2024

implemented so far is a circuit generator program written in python[8] that constructs a quantum circuit of the said chemical simulator. The program uses the SDK functions to construct the circuit on a quantum computer or a quantum computer simulator. The generated simulator comprises the initial state preparation with antisymmetrization, Coulomb potential energy calculation in 1D, 2D, or 3D coordinates using arithmetic gates, and the time evolution of the wave function through the Suzuki-Trotter decomposition. The measurement circuit is left for future work. We verified the components of the circuits by inspecting the simulated state vector after running the circuits with a test input.

As a far goal, we are interested in the simulation of large molecules such as proteins. However, at this point, we were able to generate simulator circuits for a model size up to simple amino acids. In this study, we show the generated circuits for simple models in the range of one atom in one or two dimensions (Section 3.1). For the amino acids, we present the generated circuit size (Section 3.2). Based on the qubit count of the generated circuits, we formulate the qubit count (Section 6.1) and estimate the resource requirements for some proteins (Section 6.2).

Throughout this work, we encountered several programming-level issues regarding the quantum circuit generator code and devised techniques to address them. One issue was the need to avoid interface mismatches when constructing a large circuit, which involves the decomposition of it into smaller subcircuits hierarchically. The other issue relates to the provision of a higher-level programming interface for implementing basic arithmetic formulas from low-level quantum computer instructions. Both issues become more serious as the circuit becomes larger and more complex. To deal with the first issue, we devised a design pattern[9] (a rule for structuring programs) for subcircuit invocation that features automated allocation of temporary qubits for local variables. To deal with the second issue, we applied the idea of an abstract syntax tree (AST), which is a widely known structure for compiler programs of classical computers.[10] Here, a source code expression, such as "r = sqrt($dx \times dx + dy \times dy$)", would be converted into a tree structure reflecting the order of the operations. Then, quantum instructions are generated by using the tree. The AST utilizes the said temporary qubit allocation framework whenever ancilla qubits are required by the arithmetic gates. We also implemented decimal point adjustment as a feature of the AST. The AST was used to implement the Coulomb potential energy term of the Hamiltonian, and we confirmed that it simplified the coding task. The details of fixed-point decimal point tracking are totally hidden from the Hamiltonian calculation code by the AST, and thanks to that, its formulas can be modified easily without the need for the programmer to keep track of those details manually.

We have tested all of the described software components in bottom-up unit tests. These tests were run on the Qiskit Aer simulator,[7] and the test programs were organized as an automated test suite based on the Python pytest framework.[11] The Qiskit Aer simulator allowed us to inspect the simulated state vector, and the amplitudes of all state components of the system can be obtained. Besides the test suite, we plotted the circuit output for the selected input data to demonstrate the calculations performed by the components. We used only a single computer to run the simulator and have not pursued any simulations on clusters. Doubling the memory size for a quantum computer simulator adds only one simulated qubit. Using a cluster for a quantum computer simulator would add several simulated qubits compared with those of a single computer, but currently, we do not have a strong need for those few extra qubits considering the significant cost increase.

The remainder of this study is structured as follows: Section 2 outlines the formulation of the simulator. The representation of wave functions and how they are processed are described. Section 3 describes selected samples of the simulator circuits. It starts by describing the overall circuit before detailing the subcircuits. A large portion of the subcircuits consists of arithmetic gates, such as adders and multipliers. Those gates are described in Appendix A. Section 4 discusses implementation issues. The overall simulator is composed of four components, and each component is described. Section 5 explains how the testing was performed, while Section 6 discusses the measurement of the circuit size of the current implementation and potential areas of improvement. Section 7 summarizes this work.

2. FORMULATION OF THE CHEMICAL REACTION SIMULATOR

In this section, we briefly describe the model and algorithm that we implemented. A detailed description can be found in the preceding study.[1]

2.1. Representation of the Wave Function.

The wave function is stored in quantum registers that represent the discretized coordinates of the particles. For each electron or nucleus, n qubits are used to store the coordinates of all dimensions. The coordinates of each dimension are stored in a quantum register consisting of n_1 qubits. We will describe this formulation for a 3-D model, but we will also describe simplified 2-D or 1-D models; in all cases, n represents the total number of qubits for all dimensions. Therefore, in the 2-D and 3-D models, n is $2n_1$ and $3n_1$, respectively. The integer values in these registers are transformed from discretized positional coordinates into momentum coordinates (or vice versa) by using an inverse Fourier (or Fourier) transform. In the positional representation, or "q-space", we define the simulated space to be a cube of size $L \times L \times L$. This space is discretized into 2^{n_1} segments for each dimension with a unit of $\delta q = L/2^{n_1}$. The coordinates of an electron identified by an index i are denoted as $\boldsymbol{q}_i$.

The discretized signed integer coordinates (x_i, y_i, z_i) are defined as

$$(x_i, y_i, z_i) = \frac{\boldsymbol{q}_i}{\delta q} \tag{1}$$

For atom nuclei, we will use capital letters such as X_i and $\boldsymbol{Q}_i$. Following the conventions in the chemical literature, the variable Z_i with a capital Z will be used to denote the electric charge of nucleus i, which should be easily distinguishable from the z coordinate of a nucleus by context.

The values of the positional coordinates correspond to the values of the qubits. The wave function for a system with η electrons and L_n nuclei is stored in the index register in the tensor product form:

$$|\Psi(t)\rangle = \sum_{q_1,\ldots,q_\eta,Q_1,\ldots,Q_{L_n}} \Psi(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_\eta; \boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_{L_n}; t) |x_1, y_1, z_1, \ldots, x_\eta, y_\eta z_\eta; X_1, Y_1, Z_1, \ldots, X_{L_n}, Y_{L_n}, Z_{L_n}\rangle \tag{2}$$

For a wave function in the momentum representation, the contents of the coordinate registers are transformed into the momentum of the particles. The momentum of electron i, denoted by $\boldsymbol{p}_i$, is mapped to the same coordinate qubits that were used for the spatial coordinates such that

$$(x_i, y_i, z_i) = \frac{\boldsymbol{p}_i}{\delta p} \tag{3}$$

where $\delta p = 2\pi\hbar/\delta q 2^{n_1}$ is the unit value for the discretization of momentum values.

2.2. Preparation of the Initial Wave Function. To initialize a set of qubits of a coordinate register so that the amplitude of the qubits will follow a desired distribution defined by the wave function ϕ_i provided as data, one must build a circuit to embed amplitude values to match the data.

$$|\phi_{\text{ini}}(\sigma_0)\rangle = \prod_{i=0}^{\eta-1} |\phi_i(i)\rangle = \prod_{i=0}^{\eta-1} \sum_{q_i=0}^{2^{n_1}-1} \phi_i(q_i)|q_i\rangle \tag{4}$$

Here, $|\phi_{\text{ini}}(\sigma_0)\rangle$ is the tensor product of the orbital states $|\phi_i\rangle$ and σ_0 represents the sequence of electron indices 0,1, ..., $\eta-1$, which means that orbital ϕ_i is implemented to electron i.

As the input for the preparation circuit, all coordinate registers are first set to $|0\rangle$, which means that the amplitude of $|0\cdots0\rangle$ is 1 and 0 for all other states. By applying the $R_y(\theta)$ gate, the amplitude of $|0\rangle$ can be distributed to $|0\rangle$ and $|1\rangle$ in a desired ratio of $\cos(\theta/2)$: $\sin(\theta/2)$. To decide the value for θ, let s_1 be the sum of squares of the amplitudes for the first half of the qubit states, i.e., $|00\cdots0\rangle$ to $|01\cdots1\rangle$, let s_2 be the same for the second half, i.e., $|10\cdots0\rangle$ to $|11\cdots1\rangle$. Then, $\theta = 2\tan^{-1}\sqrt{s_2/s_1}$. The $R_y(\theta)$ gate is applied to the most significant bit (MSB). This process distributes the amplitude into two states $|00\cdots0\rangle$ and $|10\cdots0\rangle$. The next step is to further distribute the amplitude of each state into still other states; the amplitude of $|00\cdots0\rangle$ is distributed into $|000\cdots0\rangle$ and $|010\cdots0\rangle$, and the amplitude of $|10\cdots0\rangle$ is distributed to $|100\cdots0\rangle$ and $|110\cdots0\rangle$. This can be done with a controlled R_y gate applied to the second MSB. By repeating this step for all qubits, we prepared the amplitude of each state. For implementing the phases, a separate process of phase embedding, which mirrors the structure of amplitude embedding, follows the amplitude embedding process. This second process applies R_z gates instead of R_y gates. The corresponding circuit is shown in Figure 28 of ref [1].

2.3. Preparation of Antisymmetric States. After the initial wave function data are stored in the coordinate registers, the amplitudes in the registers are systematically shuffled to form the Slater determinant. We used a set of ancilla qubit registers, each labeled as ak, to assist this process. The index k is in the range $1\cdots\eta-1$. These ancillae are collectively called register set a. Each value of the register set is related to a member of the permutation set S_η, which is the set of all possible permutations of the sequence of numbers in the range $0\cdots\eta-1$. We show two versions of the shuffling algorithm based on the coding scheme for registers ak. For the unary coding scheme, N bits are used to represent values from 0 to $N-1$. Here, the $(k-1)$-th bit is set to 1 to represent the number $k-1$. In the binary coding version, $\lceil \log_2 N \rceil$ bits are used, just as in ordinary binary numbers.

The shuffling consists of two steps: (1) preparation of the ancillae to the k-sequence state, or the k_Σ state, and (2) shuffling.

In the first step, the registers ak are initialized to a superposition of values which we call the k-sequence state k_Σ:

$$|k_\Sigma\rangle_{\text{a}(k-1)} \equiv \frac{1}{\sqrt{k}} \sum_{i=0}^{k-1} |i\rangle \tag{5}$$

Once all registers ak have been prepared in this way, the tensor product of all registers ak will represent the factorial state of η:

$$|\eta!\rangle_{\text{a}} = \prod_{k=2}^{\eta} |k_\Sigma\rangle_{\text{a}(k-1)} \tag{6}$$

The second step, shuffling, is the permutation of the coordinate qubits based on the content of register set a. The content of each register is an integer ranging from 0 to $k-1$. Because each $|k_\Sigma\rangle_{\text{a}(k-1)}$ in eq 6 is set individually, values for index i in eq 5 for each $|k_\Sigma\rangle_{\text{a}(k-1)}$ can appear repeatedly, such as the 0 in $\{i_{\text{a}3}, i_{\text{a}2}, \text{and } i_{\text{a}1}\} = \{1,0,0\}$. We call the sequence in the register set a σ'. For example, $\{1,0,0\}$ is a σ' for $\eta = 4$. This sequence can be mapped to one of the sequences in S_η, where the elements of each sequence in S_η are all different. To obtain the map, we define permutation τ to be a function of σ'. This is done recursively in eqs7 and8, where (i, j) is a permutation operator that swaps the ith and jth element of σ' and $\eta \geq 2$.

$$\tau(|k\rangle_{\text{a}1}) = (k, 1) \tag{7}$$

$$\tau(|k\rangle_{\text{a}(\eta-1)}|k'\rangle_{\text{a}(\eta-2)\ldots\text{a}1}) = (k, \eta-1)\tau(|k'\rangle_{\text{a}(\eta-2)\ldots\text{a}1}) \tag{8}$$

Here, "1" in eq 7 comes from $\eta - 1$ in eq8 where $\eta = 2$ in this case. τ operates on σ_0 such that $\sigma = \tau(|i_{\text{a}(\eta-1)}\rangle|i_{\text{a}(\eta-2)}\rangle\cdots|i_{\text{a}1}\rangle)\sigma_0$. For example, when $\sigma' = \{4,3,2,1\}$, i.e., registers a4 through a1 have the values 4,3,2 and 1, the resulting permutation would be the identity: $\tau(|4\rangle_{\text{a}4}|3\rangle_{\text{a}3}|2\rangle_{\text{a}2}|1\rangle_{\text{a}1}) = (4,4)(3,3)(2,2)(1,1) = I$, and for that case $\sigma = I\sigma_0$. On the basis of this relation, we can prepare the antisymmetrized wave function such as

$$|\Phi_{\text{ini}}\rangle \equiv |\phi_{\text{ini}}(\sigma_0)\rangle|\eta!\rangle_a \mapsto |\Psi_{\text{anti}}\rangle \equiv \frac{1}{\sqrt{\eta!}} \sum_{\sigma\in S_\eta} \text{sgn}(\sigma)|\phi_{\text{ini}}(\sigma)\rangle|\sigma'\rangle_a$$

2.4. Preparation of a Mixed State of Energy Configurations. To enable a simulation with a target temperature, several antisymmetrized wave functions with different energy levels can be combined to form a mixed state. An explanation of the circuit can be found in the first part of the report.[1]

2.5. Time Evolution. The time evolution is calculated on the basis of the Schrödinger equation by using the Suzuki−Trotter decomposition. The time-independent Hamiltonian of the system is a sum of Hamiltonians for electrons and nuclei; these are further decomposed into kinetic energy terms and potential energy terms:

$$H = (H_{\text{ek}} + H_{\text{ep}}) + H_{\text{en}} + (H_{\text{nk}} + H_{\text{np}}) \tag{10}$$

$$H_{\text{ek}} = \sum_i \frac{\boldsymbol{p}_i^2}{2m_{\text{e}}},\ H_{\text{ep}} = \sum_{i>j} \frac{e^2}{|\boldsymbol{q}_i - \boldsymbol{q}_j|}$$

$$H_{\text{en}} = -\sum_{i,j} \frac{e^2 Z_j}{|\boldsymbol{q}_i - \boldsymbol{Q}_j|}$$

$$H_{\text{nk}} = \sum_i \frac{\boldsymbol{P}_i^2}{2M_i},\ H_{\text{np}} = \sum_{i>j} \frac{e^2 Z_i Z_j}{|\boldsymbol{Q}_i - \boldsymbol{Q}_j|}$$

Here, H_{ek} and H_{ep} are the kinetic and Coulomb energy terms of electrons, respectively, where H_{en} is the Coulomb energy between electrons and nuclei. H_{nk} and H_{np} are the kinetic and Coulomb energies of nuclei, respectively. $m_e$and M_i are the mass of the electron and nucleus i.e., is the elementary charge, and Z_i is the atomic number of the nucleus i.

According to the Suzuki−Trotter formula, the time evolution for a period of t can be approximated as

$$\begin{aligned}|\Psi(t)\rangle &= e^{-iHt/\hbar}|\Psi_0\rangle \\ &\simeq (e^{-iH_{\mathrm{ek}}t/\hbar n} e^{-iH_{\mathrm{ep}}t/\hbar n} e^{-iH_{\mathrm{en}}t/\hbar n} e^{-iH_{\mathrm{nk}}t/\hbar n} e^{-iH_{\mathrm{np}}t/\hbar n})^n |\Psi_0\rangle\end{aligned} \tag{11}$$

Paying attention to the fact that nuclei are much heavier than electrons, we can approximate eq11 as

$$\begin{aligned}|\Psi(t)\rangle \simeq [&(e^{-iH_{\mathrm{ek}}t/\hbar nm} e^{-iH_{\mathrm{ep}}t/\hbar nm} e^{-iH_{\mathrm{en}}t/\hbar nm})^m \\ &e^{-iH_{\mathrm{nk}}t/\hbar n} e^{-iH_{\mathrm{np}}t/\hbar n}]^n |\Psi_0\rangle\end{aligned} \tag{12}$$

We will use the Fourier transform and its inverse to convert the wave function from the positional representation to the momentum representation and vice versa. Note that the time evolution of the kinetic energy term becomes much simpler in momentum space.

3. DESIGNED CIRCUITS

The main outcome of this study is a Python program that generates a time evolution simulator as a quantum circuit. The circuit takes the initial state of the wave function as precalculated input data. Then, it repeatedly calculates the time-evolved wave function. This will cause changes in the spin−orbitals. After the time-evolution process has been repeated a specified number of times, the resulting wave function is measured. So far, we have completed our implementation up to the time-evolution loop and left the measurement part for future work. Note that the circuit parameters shown in Table 1 can be adjusted.

Table 1. Configuration Parameters of the Circuit Generator

parameter	definition
d	dimension of coordinates [1,3]
n_1	number of bits per coordinate for a dimension
L	length of simulation space along one axis
η	number of electrons
L_n	number of nuclei
N_E	number of configurations in the mixed state
$\phi_{c,\,i,\,d,\,x}$	array holding data for electron orbital ϕ_i indexed by [energy configuration, orbital, dimension, position]
$\psi_{c,\,i,\,d,\,X}$	array holding data for nucleus orbital ψ_i
δt	step of time evolution for an electron
T_N	number of nucleus-level iteration steps
T_e	number of electron-level iteration steps within one nucleus-level iteration step

In this section, we show several examples of generated circuits and their subcircuits (Section 3.1) and describe the number of qubits required for the overall circuit for a selected range of parameters (Section 3.2).

3.1. Examples of the Generated Simulator Circuit.

The circuits presented in this section are organized hierarchically. There are four simulator configurations: (1) a basic one-dimensional configuration with a 3-bit coordinate, one electron, one nucleus, and one initial wave function ($d = 1, n = 3, \eta = 1, L_n = 1, N_E = 1$), (2) the first example with two dimensions instead of one ($d = 2, n = 4, \eta = 1, L_n = 1, N_E = 1$), (3) a one-dimensional configuration with four electrons ($d = 1, n = 2, \eta = 4, L_n = 1, N_E = 1$) and (4) a one-dimensional configuration with two electrons, with four wave functions in the initial mixed state ($d = 1, n = 2, \eta = 2, L_n = 1, N_E = 4$).

The circuits and their subcircuits are listed in Table 2.

3.1.1. Basic One-Dimensional Configuration.

The first example implements a minimal one-dimensional model with three coordinate bits and has one electron and one nucleus ($d = 1, n = 3, \eta = 1, L_n = 1$), namely, a one-dimensional model of a hydrogen atom. Since the model is one-dimensional, the circuit for the Hamiltonian (eq 10) uses a simple absolute value gate instead of a sequence of gates to calculate the norm for the distance between the electron and the nucleus. The overall circuit is shown in Figure 1. The total number of qubits is 19. Each line of the diagram represents a qubit. The order of the qubits is the least significant bit (LSB) first or in little-endian. This order is the default used by Qiskit and is different from the one assumed in the first part of this report.[1] The top six lines are qubits for the coordinate bits. The labels $e0x_0, ..., e0x_2$ denote the three bits for the x coordinate of electron 0. $n0x_0, ..., n0x_2$ are the three bits for the x coordinate of nucleus 0. Thus, the bit count is 6, or $(\eta + L_n)n$. The lower 13 lines with labels *tmpn* are temporary qubits. Those qubits are internally used at the gates, such as the electron potential energy (P.E.) time evolution gate Θ_{ep}; they have no effect outside of the gates. The subcomponents function as follows: The Slater determinant (SD) state preparation gate, Ψ_{sd}, prepares the initial state of the wave function on the coordinate qubits on the basis of orbital data provided as input at the circuit generation time. The rest of the circuit consists of nested time-evolution loops that are executed repeatedly. The outer loop operates with nucleus scale time steps and the inner loop operates with electron scale time steps. The electron scale loop is the first element of the nucleus scale loop. The first gate inside the inner loop is the Θ_{ep} gate. This gate calculates the electron−electron and electron−nucleus potential energy terms of the Hamiltonian, and each state in a superposition will have its phase altered according to the coordinates of the particles. Following Θ_{ep} comes the n-element inverse quantum Fourier transform gate, $n\mathrm{QFT}^\dagger$. This transforms the position representation into a momentum representation. Then follows the electron kinetic energy (K.E.) time evolution gate, Θ_{ek}, which alters the phases according to the coordinates in p-space. After that comes the n-element Quantum Fourier transform gate, $n\mathrm{QFT}$, that transforms the momentum representation back to the position representation. Outside the electron scale loop, the corresponding gates for nuclei will follow; the nucleus P.E. time evolution gate, Θ_{np}, the $n\mathrm{QFT}^\dagger$ gate, the nucleus K.E. time evolution gate, Θ_{nk}, and the $n\mathrm{QFT}$ gate.

The SD state preparation gate, Ψ_{sd}, is shown in Figure 2A. Since there is only one electron in this first example, the gates that perform the permutations to build the Slater determinant are not present. They will appear in the third example. The three-bit state embedding gate, emb(3), is applied to each of the two coordinate registers, e_0x and n_0x. Each emb(3) gate sets the amplitudes of 3-qubit states on the basis of data provided as an array. The internals of emb(3) are shown in Figure 2B. The emb(3) gate has two internal gates, embθ(3) that embeds amplitudes and embϕ(3) that embeds phases. The recursive structures of those two gates are illustrated in Figure 2C through H.

The content of the electron P.E. time evolution gate, Θ_{ep}, is shown in Figure 3. This gate has the greatest number of parameter qubits in the first example, which is 19. It is

Table 2. List of Sample Circuits

label	description
Figure 1	overall circuit for $d = 1$, $n = 3$, $\eta = 1$, $L_n = 1$, $N_E = 1$
Figure 2A	the Slater determinant preparation gate Ψ_{sd} for a single electron model
Figure 2B	the 3-bit state embedding gate emb(3)
Figure 2C	the 3-bit amplitude embedding gate embθ(3)
Figure 2D	the 2-bit amplitude embedding gate embθ(2)
Figure 2E	the 1-bit amplitude embedding gate embθ(1)
Figure 2F	the 3-bit phase embedding gate embϕ(3)
Figure 2G	the 2-bit phase embedding gate embϕ(2)
Figure 2H	the 1-bit phase embedding gate embϕ(1)
Figure 3	the electron P.E. time evolution gate Θ_{ep}
Figure 4A	the electron K.E. time evolution gate Θ_{ek}
Figure 4B	the n-element inverse quantum Fourier transform gate nQFT† for $d = 1$, $n = 3$, $\eta = 1$
Figure 5	Θ_{ep} for $d = 2$, $n = 4$, $\eta = 1$, $L_n = 1$
Figure 6	the overall circuit for $d = 1$, $n = 2$, $\eta = 4$, $L_n = 1$, $N_E = 1$
Figure 7A	the Slater determinant preparation gate Ψ_{sd} for a multiple electron model
Figure 7B	the unary-coded permutation gate S_u
Figure 8A	the unary-coded sequence preparation gate $ku_\Sigma(2)$
Figure 8B	the gate $ku_\Sigma(3)$
Figure 8C	the gate $ku_\Sigma(4)$
Figure 9A	the unary-coded conditional shuffling gate $\sigma u(2)$
Figure 9B	the gate $\sigma u(3)$
Figure 9C	the gate $\sigma u(4)$
Figure 10	the binary coded permutation gate S_b
Figure 11A	the binary-coded sequence preparation gate $kb_\Sigma(2)$
Figure 11B	the gate $kb_\Sigma(3)$
Figure 11C	the gate $kb_\Sigma(4)$
Figure 11D	the gate $kb_\Sigma(6)$
Figure 12A	the binary-coded conditional shuffling gate $\sigma b(2)$
Figure 12B	the gate $\sigma b(3)$
Figure 12C	the gate $\sigma b(4)$
Figure 13A	the general state preparation gate Ψ_g for $d = 1$, $n = 2$, $\eta = 2$, $L_n = 1$, $N_E = 4$
Figure 13B	the 2-bit configuration preparation gate
Figure A-1A	the unsigned adder uadd(3)
Figure A-1B	the unsigned mismatched bit length adder uaddv(3,4)
Figure A-2A	the signed adder sadd(3)
Figure A-2B	the signed constant value adder scoadd(3,1)
Figure A-3A	the unsigned subtractor usub(3)
Figure A-3B	the signed subtractor ssub(3)
Figure A-4	the unsigned multiplier umult(3)
Figure A-5A	the signed multiplier smult(3)
Figure A-5B	the signed square ssquare(3)
Figure A-6	the unsigned divider udiv(4,3)
Figure A-7A	the absolute value abs(4)
Figure A-7B	the square root sqrt(3)
Figure A-8A	the carry gate
Figure A-8B	the carry gate with one input given as constant 0 cocarry(0)
Figure A-8C	the carry gate with one input given as constant 1 cocarry(1)
Figure A-8D	the inverse carry gate icarry
Figure A-8E	the inverse carry gate with one input given as constant 0 icocarry(0)
Figure A-8F	the inverse carry gate with one input given as constant 1 icocarry(1)
Figure A-8G	the half carry gate hcarry
Figure A-8H	the half carry gate with one input given as constant 0 cohcarry(0)
Figure A-8I	the half carry gate with one input given as constant 1 cohcarry(1)
Figure A-8J	the inverse half carry gate ihcarry
Figure A-8K	the inverse half carry gate with one input given as constant 0 icohcarry(0)
Figure A-8L	the inverse half carry gate with one input given as constant 1 icohcarry(1)
Figure A-9A	the sum gate qsum
Figure A-9B	the sum gate with one input given as constant 0 cosum(0)
Figure A-9C	the sum gate with one input given as constant 1 cosum(1)
Figure A-9D	the inverse sum gate iqsum
Figure A-9E	the inverse sum gate with one input given as constant 0 icosum(0)
Figure A-9F	the inverse sum gate with one input given as constant 1 icosum(1)
Figure A-9G	the half sum gate qhsum
Figure A-9H	the half sum gate with one input given as constant 0 cohsum (0)
Figure A-9I	the half sum gate with one input given as constant 1 cohsum (1)
Figure A-9J	the inverse half sum gate iqhsum
Figure A-9K	the inverse half sum gate with one input given as constant 0 icohsum(0)
Figure A-9L	the inverse half sum gate with one input given as constant 1 icohsum(1)

constructed as follows: Θ_{ep} uses the arithmetic gates ssub (signed subtractor), abs (absolute value), and udiv (unsigned divider). Since the coordinates are one-dimensional in this example, the electron−nucleus distance is calculated using a simple absolute value gate without square roots. Once the calculated result is used to alter the phase through P gates, the calculation is reversed using the inverse versions of the gates, denoted by a dagger on their names. Within the right half, tmp_0, ..., tmp_2 are the three temporary qubits used for carry bits. These are required for internal use at the arithmetic gates; they do not affect Θ_{ep}. The lines with labels one_n, $sign_0$, msb_1, and zz_n are ancilla qubits that the Θ_{ep} circuit uses for intermediate values. These qubits are used internally at Θ_{ep}; they have no effect outside Θ_{ep}. The temporary qubits are labeled *tmpi* in the leftmost column in Figure 3. The number of temporary qubits is a constant independent of the number of electrons. When the number of particle pairs increases, this same circuit structure is repeated for different pairs, while the same set of temporary qubits is reused. In this 1-D model with three qubit coordinates ($d = 1$, $n = n_1 = 3$), the number of qubits for the coordinates of electrons and nuclei is $dn_1(\eta + L_n)$ and the required number of temporary qubits is $3n_1 + 4 = 13$. The sum is $dn_1(\eta + L_n) + 3n_1 + 4$. The breakdown is shown in Table 3. Note that this formula is only for the single-electron case, since for multiple electrons additional qubits must be introduced to implement antisymmetrization.

The content of the electron K.E. time evolution gate, Θ_{ek}, is shown in Figure 4A. The depicted circuit is implemented with phase gates and controlled phase gates without using any arithmetic gates.

Θ_{np} and Θ_{nk} are, respectively, the P.E. and K.E. terms for nuclei; their figures are omitted since they are analogous to the electron counterparts. In this example, Θ_{np} is empty since only one nucleus is present and there are no nucleus−nucleus interaction terms.

The content of the n-element inverse quantum Fourier transform gate, nQFT†, is shown in Figure 4B. The inverse quantum Fourier transform gate IQFT from the Qiskit library is

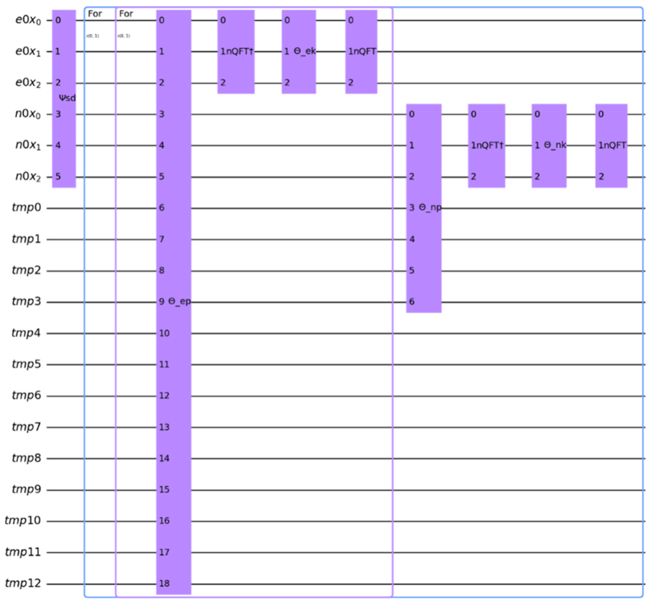


Figure 1. Circuit diagram of generated results for $d = 1$, $n = 3$, $\eta = 1$, $L_n = 1$, and $N_E = 1$. The overall circuit consists of state-preparation and time-evolution gates. The initial state is prepared by the SD state preparation gate, Ψ_{sd}, and then, nested time evolution loops are applied to the state. The inner electron time-scale loop consists of an electron P.E. time evolution gate, Θ_{ep}, an n-element inverse quantum Fourier transform gate, $n\text{QFT}^\dagger$, an electron K.E. time evolution gate, Θ_{ek}, and an n-element quantum Fourier transform gate, nQFT. The outer nucleus time-scale loop consists of the inner loop, the nucleus P.E. time evolution gate, Θ_{np}, the $n\text{QFT}^\dagger$ gate, the nucleus K.E. time evolution gate, Θ_{nk}, and the nQFT gate.

used here. IQFT is independently applied to each of the coordinate bit sets.

3.1.2. Two-Dimensional Configuration. The second example (Figure 5) is an extension of the first one to two dimensions ($d = 2$, $n_1 = 2$, $n = dn_1 = 4$, $\eta = 1$, $L_n = 1$). This is also a model of a single hydrogen atom but in two dimensions. Compared to the first example, a more complex norm circuit is used to calculate the Hamiltonian. Only the Θ_{ep} gate, which determines the overall circuit bit count, is shown; the other gates are omitted. The overall bit count is 36, and the number of coordinate qubits is 8, i.e., $(\eta + L_n)n$. As a result of having multiple dimensions, the electron−nucleus distance calculation requires a norm calculation, which involves two square gates and one square root gate. In the circuit, the difference of coordinates is calculated with ssub (signed subtraction) gates, and the result is fed to an ssquare (signed square) gate for the x and y coordinates. Those squares are then summed by a uaddv (unsigned adder) gate, and the sum is given to an sqrt (square root) gate. A udiv (unsigned divide) gate takes the inverse of the result of the sqrt gate. The number of temporary qubits for Θ_{ep} in this example is 28, i.e., $(2d + 6)n_1 + 8$. Their breakdown is shown in Table 4. The number of temporary qubits remains the same, even when the number of electrons is increased. The overall bit count for Θ_{ep} is $(\eta + L_n)n + (2d + 6)n_1 + 8 = 36$. This same formula can also be used for $d = 3$, i.e., 3-dimensional models.

3.1.3. Multiple Electron Configuration. The third example implements a model with multiple electrons and introduces antisymmetrization. The overall circuit is shown in Figure 6. This example implements a one-dimensional model with four electrons and one nucleus, using two-bit coordinates ($d = 1$, $n = 2$, $\eta = 4$, $L_n = 1$). This corresponds to a beryllium atom. The purpose of this example is to show the circuits related to antisymmetrization. As such, the one-dimensional model is used to simplify the circuit. The total number of qubits is 28. We will omit the formula for this number since the content of Θ_{ep} for this sample takes much space to show.

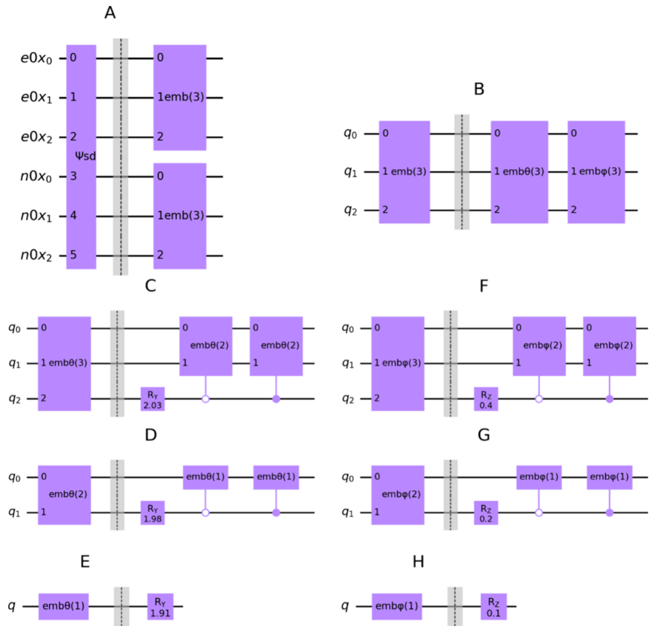


Figure 2. SD preparation gate, Ψ_{sd}, and its internals. (A) Ψ_{sd} shown as a unit on the left and its internals on the right. In this example, with only one electron, the gates for permutations that are characteristic of the Slater determinant are not present. The two state embedding gate, emb(3), embeds the orbital data in the two coordinate registers independently. (B) 3-bit state embedding gate, emb(3). This gate consists of a 3-bit amplitude embedding gate, emb$\theta(3)$, and a phase embedding gate, emb$\phi(3)$. (C) 3-bit amplitude embedding gate, emb$\theta(3)$. This gate is defined recursively with the two-bit emb$\theta(2)$ gate. The phase value shown on the R_y gate is an example; it varies depending on the amplitude data. (D) Gate embθ (2), same for 2 bits, (E) gate emb$\theta(1)$, same for 1 bit. (F) 3-Bit phase embedding gate, emb$\phi(3)$. This gate is also defined recursively. The difference from emb$\theta(3)$ is in the use of the R_z gate instead of the R_y gate. (G) Same for 2 bits. (H) Same for 1 bit.

When there are two or more electrons, the wave function of Fermions must be antisymmetrized. This is done using the SD state-preparation gate, Ψ_{sd}.

The SD gate, Ψ_{sd}, is shown in Figure 7A. This gate starts by embedding the individual orbital states and shuffles them to make 1 equiv of a Slater determinant. The shuffling is done inside a unary-coded permutation gate S_u. The subscript "u" signifies that the gate uses unary coding internally instead of binary coding. Later, we will show binary-coded versions of these gates.

After the coordinate qubits are set to a predetermined distribution by the state embedding gates, emb(n), the S_u gate is applied. The content of the S_u gate is shown in Figure 7B. Each ancilla register au$k$ is initialized to a k-sequence state $|k_\Sigma\rangle$ by the sequence preparation gate, $ku_\Sigma(k)$. $|k_\Sigma\rangle$ is used by the unary-coding-based shuffling gates, $\sigma u(k)$. A gate sequence of XZX is inserted before the shuffling gates when η is even. This is to cancel an artificial global phase π added in the circuit in Figure 9. The input to XZX is $|0\rangle$, so the XZX sequence adds a global phase of π. The purpose of this article is as follows. There are three, or $\eta - 1$, shuffle gates $\sigma u(k)$ shown in Figure 7B. As will be

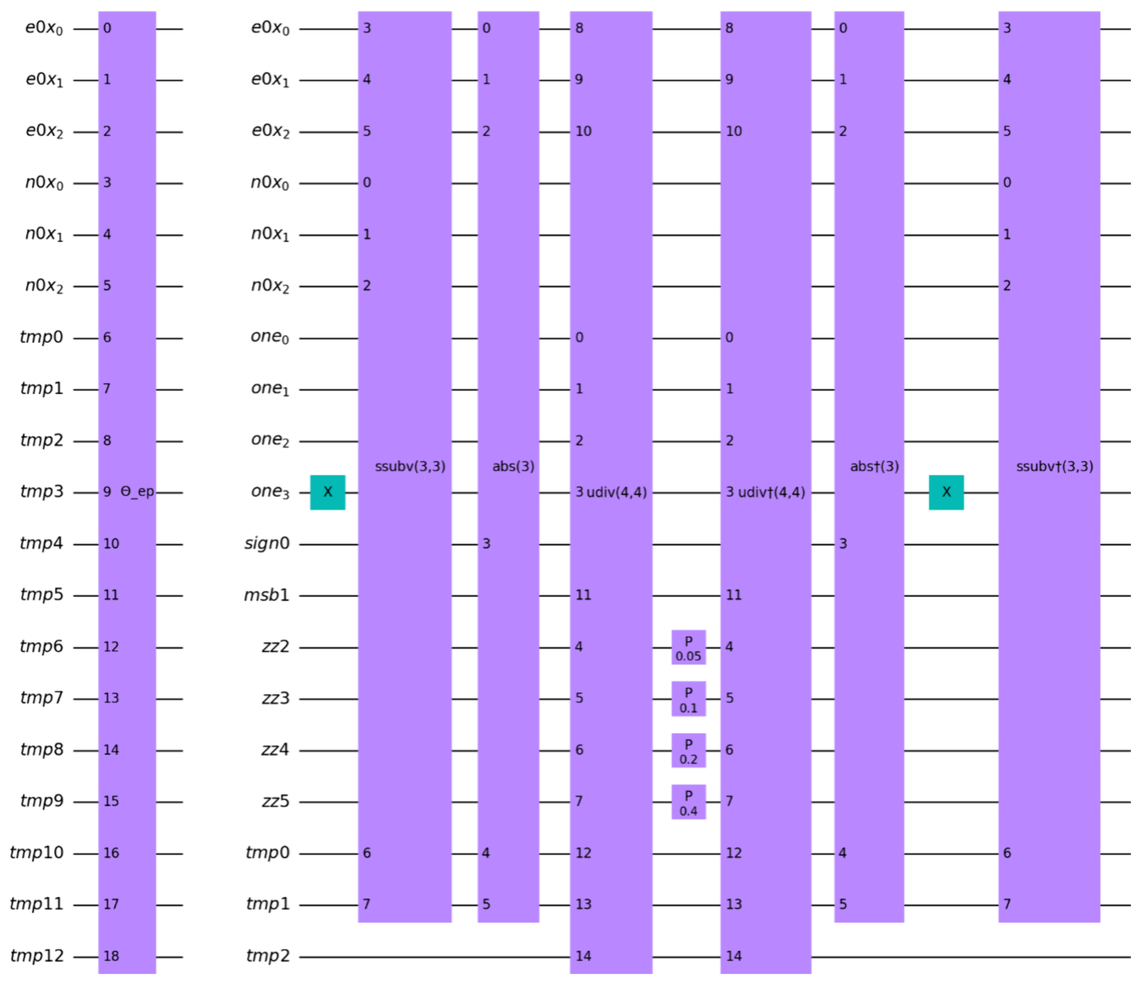


Figure 3. Electron P.E. time evolution gate, Θ_{ep}, shown as a single gate on the left and its internals on the right. $e0x_0$ through $n0x_2$ are the coordinate qubits, and $tmp0$ through $tmp12$ are temporary qubits on the outside of the gate. Those temporary qubits are assigned more specific names inside the gate. The first X gate sets the MSB of the 4-bit fixed-point value $(one_3, one_2, one_1, one_0)$ to 1.000 in binary. This value is later used as the numerator of $1/|e0_x - n0_x|$. The ssub(3,3) gate is a signed subtraction that computes $e0x - n0x$ and sets the result to $e0x$. The bit assignments of the arithmetic gates are explained in Appendix A. The abs(3) gate converts the subtraction result to its absolute value and stores the sign bit on the $sign0$ bit for later use. $msb1$, which is initialized to 0 is added to the denominator to make the bit-count of the numerator and denominator match, whereby our implementation of the divisor gate produces 1.1111 (all bits set to 1) as quotient for the division-by-zero case, which is mathematically wrong but most reasonable; in other cases, 1.1111 is not obtained. Then udiv(4,4) computes the quotient $1.000/|e0_x - n0_x|$. The following phase gates are applied to the results of the division. The phase shift is executed according to the bit-wise results. After that, the arithmetic operations are executed in reverse order, restoring the qubit values to their original values except for the phase, which is not restored. The second X gate on the one_3 bit should be rendered as the final gate in the sequence but appears in front of the ssubv(3,3) gate. This is due to the circuit drawing algorithm of Qiskit, which rearranges the display order of gates within a range that does not affect the result. In this case, the ssubv(3,3) gate does not affect the qubit labeled one_3 bit, so changing the location of the X gate to a point before or after the ssubv(3,3) gate does not change the result.

Table 3. Bit-count Breakdown of Registers Shown in Figure 3

label	count	description
$e0x_0$, ..., $e0x_2$	ηdn_1	coordinates of electrons
$n0x_0$, ..., $n0x_2$	$L_n dn_1$	coordinates of nuclei
one_0, ..., one_3	$n_1 + 1$	numerator 1.0
$sign_0$	1	ancilla for abs gate
msb_1	1	bit added to the MSB side of the divisor to make the bit-count coincide with that of the dividend
zz_2, ..., zz_5	$n_1 + 1$	quotient
tmp_0, ..., tmp_2	n_1	

explained in Figure 9, each shuffle gate adds a global phase of π. When η is odd, an even number of π phases are added so they cancel out. In contrast, when η is even, an odd number of π phases are added so the phase remains. The global phase does not affect the result of the circuit, but when the circuit is run on a quantum computer simulator for debugging and the state vector is inspected directly, this global phase appears and may be confusing when judging if the result is correct. Hence, the global phase is compensated here to ease debugging.

Shuffling gate $\sigma u(2)$ permutes $|\phi_0\rangle$ and $|\phi_1\rangle$ by using $|2_\Sigma\rangle = \frac{1}{\sqrt{2}}(|1\rangle + |0\rangle)$; shuffling gate $\sigma u(3)$ permutes $|\phi_0\rangle$, $|\phi_1\rangle$ and $|\phi_2\rangle$ by using $|3_\Sigma\rangle = \frac{1}{\sqrt{3}}(|100\rangle + |010\rangle + |001\rangle)$; the case for $k = 4$ follows this pattern. Note that $|2_\Sigma\rangle$ does not follow the rule of unary coding. That is, $|2_\Sigma\rangle$ should be $\frac{1}{\sqrt{2}}(|10\rangle + |01\rangle)$

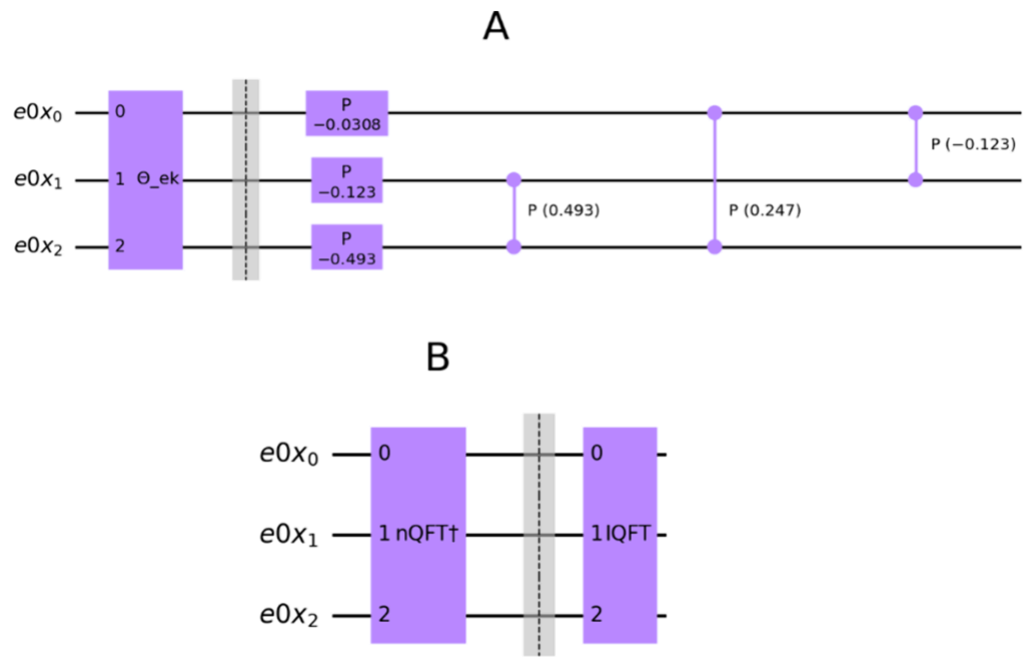


Figure 4. (A) Electron K.E. time evolution gate, $\Theta_{ek}(p)$ (B) n-element inverse quantum Fourier transform gate, $n\mathrm{QFT}^\dagger$, for $d = 1$, $n = 3$,and $\eta = 1$.

according to the rule, but the downstream circuits need only one of the two bits. Thus, the LSB of register $au1$ is omitted as a qubit-saving optimization.

The sequence preparation gate ku_Σ for $k = 2$, 3, and 4 is shown in Figure 8. A CNOT gate follows each R_y gate. Thanks to the pair, only one of the output qubits is $|1\rangle$; the other qubits are $|0\rangle$.

The unary-coded conditional shuffling gate, $\sigma u(k)$, for k = 2,3,4 is shown in Figure 9. This gate shuffles the coordinate registers based on the value of $au(k - 1)$. Since $au(k - 1)$ is unary-coded, its value is represented by the single bit within the bits of register $au(k - 1)$ that is set to $|1\rangle$. Note that for the case of $k = 2$, we omit the LSB of register $au1$, so the remaining bit is the second bit of the original $au1$. Now, let l be the index of the $|1\rangle$ bit within the bits of register $au(k-1)$, i.e., the bit $au(k-1)_l$ is $|1\rangle$. The basic intent of the shuffle gate is to do nothing when l takes the greatest value, $k - 1$; when $l < k - 1$, to swap the contents of the coordinate registers of index l and $k - 1$ and at the same time negate the sign by adding π to the phase. When implementing this function in gates, it is easier to add the π phase for the inverse condition, i.e., when $l = k - 1$ instead of when $l < k - 1$. This can be achieved using a single Z gate on the bit $\mathrm{au}(k - 1)_l$. Different from the basic intent of the gate, this would add a global phase of π, but the added phase does not affect subsequent operations. This is why the global phase of π is added by each $\sigma u(k)$ gate, as is mentioned in Figure 7.

Binary-coded versions of the sequence preparation gates and conditional shuffling gates can also be constructed. Here, the trade-off between the unary and binary versions is that the unary versions consume more qubits, but require fewer gates.[1] Figure 10 shows the binary-coding-based permutation gate S_b. The bit size of register $\mathrm{ab}(k - 1)$ is the number of bits required to express $k - 1$ in binary. S_b consists of sequence preparation gates, $kb_\Sigma(s, k)$, and conditional shuffling gates, $\sigma b(k)$. Its structure is similar to that of S_u. Here, s is the number of qubits for constructing the register ab$k$. For the same reason as the unary-coded version, a sequence of XZX is added when η is an even number to ease testing and debugging of the circuit based on the inspection of the state vector. The XZX gates are not needed otherwise.

Figure 11 shows the binary-coded sequence preparation gate $kb_\Sigma(s, k)$ for k = 2,3,4, and 6; s is 1,2,2 and 3, respectively. The structure of this gate is dependent on k. When k is a power of 2, the gate is constructed solely from H gates (the cases of $k = 2$ and $k = 4$ are illustrated). For the other cases, it is recursively constructed, as shown for $k = 6$.

Figure 12 shows binary-coded conditional shuffling gates $\sigma b(k)$ for k = 2,3, and 4. The coordinate registers are swapped, depending on the value of the ab$k$ register. This value is the binary-coded integer held in the bits of $\mathrm{ab}(k - 1)$. Let l be that value. The intent of this gate is the same as that of the unary-coded version shown in Figure 9; i.e., when $l = k - 1$, do nothing; when $l < k - 1$, swap the $(k - 1)$-th coordinate register for the l-th coordinate register and add a phase of π. However, it is easier to construct a circuit that adds the phase on the inverse condition, i.e., when $l = k - 1$. This can be achieved by using a Z gate in A, the controlled-$(-Z)$ gate in B, and the controlled-Z gate in C. These circuits each add a global phase of π, which causes the artificial global phase mentioned in the description of Figure 10.

3.1.4. Mixed State Preparation Configuration. The fourth and final example highlights how a mixed state can be prepared, which is a model of excitation, thereby making chemical reactions possible. We will consider a model with one dimension, 2-bit coordinates, two electrons, one nucleus, and four sets of initial wave functions. The difference between this circuit and the previous examples is the state preparation gate. In place of the SD preparation gate that we have shown, we introduce the general state preparation gate, Ψ_g, (Figure 13 A; see also ref 1, Section 4.2). Four Ψ_{sd} gates prepare four sets of wave functions. They are given different orbital data to initialize the coordinate qubits. The configuration preparation gate, ρ (Figure 13B) is responsible for setting the control bits for the wave function preparation gates according to the intended probability distribution.

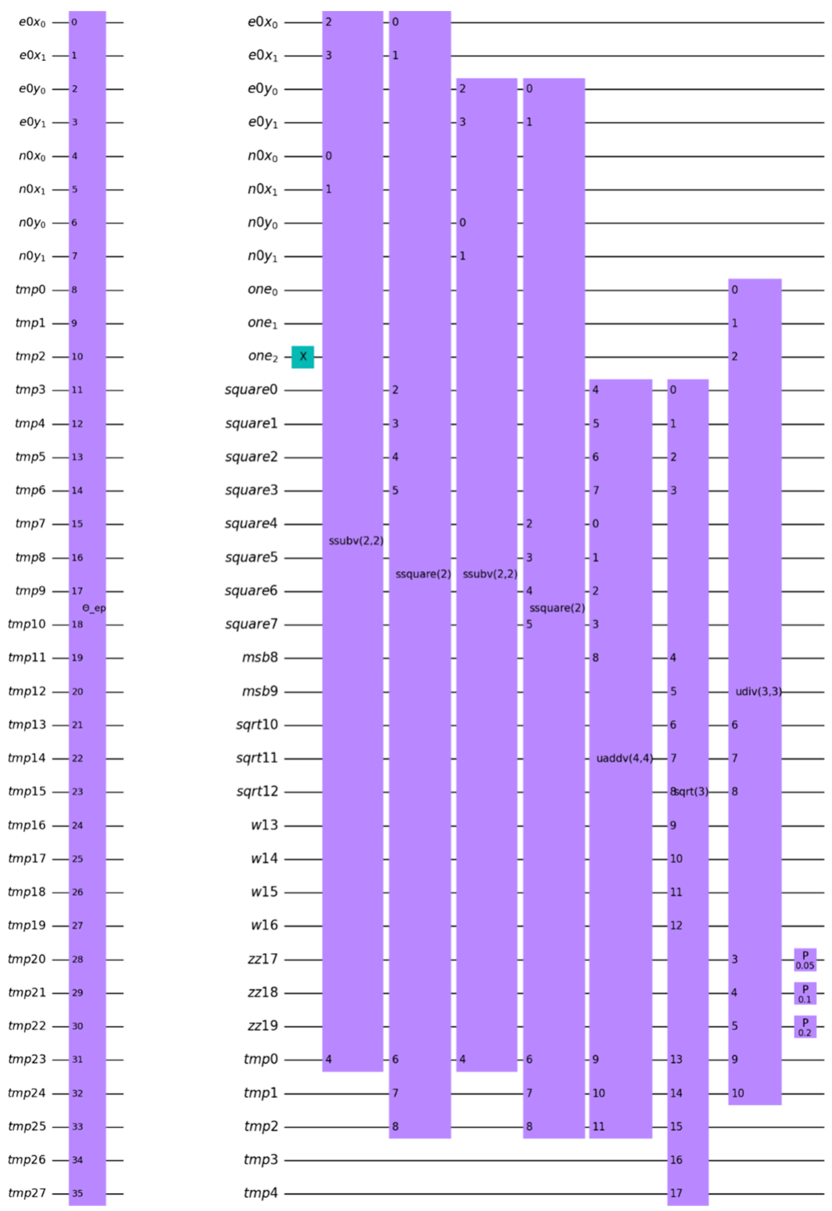


Figure 5. Θ_{ep} for 2-D coordinates ($d = 2$, $n = 4$, $\eta = 1$, and $L_n = 1$). The overall circuit is omitted since it is analogous to the first example. Only the electron P.E. time evolution gate, Θ_{ep}, is shown. It is followed by the inverses of all of the arithmetic gates in reverse order (not shown in the diagram) to restore the calculation result to the original state.

Table 4. Bit-count Breakdown of Registers Shown on Figure 5

label	bit count	description
one_0, one_1, one_2	$n_1 + 1$	the numerator 1.0
$square_0$, ..., $square_7$	$d \times 2n_1$	d sets of square results
msb_8	1	carry for the uaddv gate
msb_9	1	MSB added to make the bit-count of the square root input even
$sqrt_{10}$, ..., $sqrt_{12}$	$n_1 + 1$	result of square root
w_{13}, ..., w_{16}	$n_1 + 2$	ancilla bits required by square root
zz_{17}, ..., zz_{19}	$n_1 + 1$	quotient
tmp_0, ..., tmp_4	$2n_1 + 1$	carry bits for the sqrt gate

3.2. Evaluation of the Qubit Count for Different Model Sizes.

The qubit counts of the generated circuit for several atom combinations are shown in Figure 14. The qubit counts scale as $O(\eta \log \eta)$ with respect to the number of particles as given by eqs C-7 and C-8. The circuit generator can be configured to apply the Born–Oppenheimer approximation (BOA). When the BOA is applied, the coordinates of the nucleus are treated as constants and the qubits used to store the nuclei coordinates are omitted, thus requiring fewer qubits. Estimates by Kassal[4] are shown for comparison. The differences from Kassal's estimate are discussed in Section 6.1. The breakdown of the qubit count is described in Appendix C.

4. DESIGN OF THE IMPLEMENTATION

4.1. Program Structure.

The simulator program consists of four components (Figure 15). The first component is the arithmetic component, which provides functions to build circuits for operations such as addition and subtraction. The second component is the qubit heap component, which maintains a heap or pool of qubits that can be assigned as

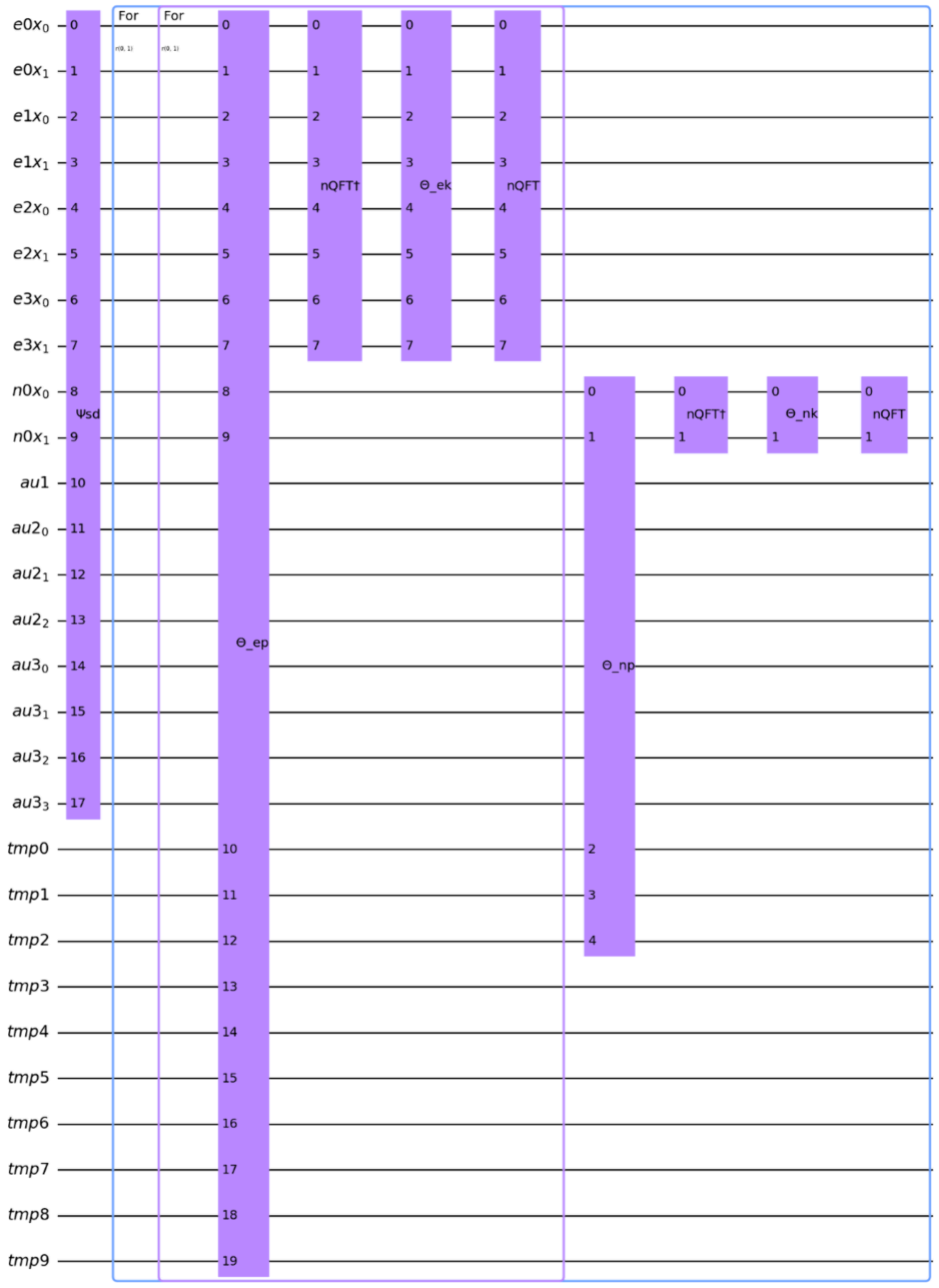


Figure 6. Circuit diagram of the overall circuit for the third example with four electrons ($d = 1$, $m = 2$, $\eta = 4$, and $L_n = 1$). The difference from the preceding examples is that the SD gate, Ψ_{sd}, appears as the first gate. It takes additional ancilla registers $au1$, $au2$, and $au3$ which comprise the register set "a" in the formulation for state preparation. The registers are used to create a representation of $|\eta\,!\rangle$ and that value is used for shuffling the coordinate qubits.

temporary qubits required by arithmetic circuits. The third is the abstract syntax tree component, which provides node classes for a tree structure that represents a calculation of a formula. It uses the arithmetic and qubit heap components as subcomponents and provides a high-level interface on top of the arithmetic component. The fourth is the simulator gates component, which contains all of the gates that are specific to Hamiltonian simulation. It uses the abstract syntax tree component.

These components are described in the following subsections.

4.2. Arithmetic Component. The arithmetic component provides generator functions for the arithmetic circuits. The time-evolution algorithm requires certain arithmetic operations to be executed on values represented as qubit states. These operations thus work on superposition states, wherein one execution of the circuit entails operations on multiple values. The circuits can be generated by calling one of the generator functions in this component. The functions come in two forms, instruction emitting functions and gate-creating functions, described as follows:

- Instruction emitting functions: These functions implement a specific arithmetic operation by emitting

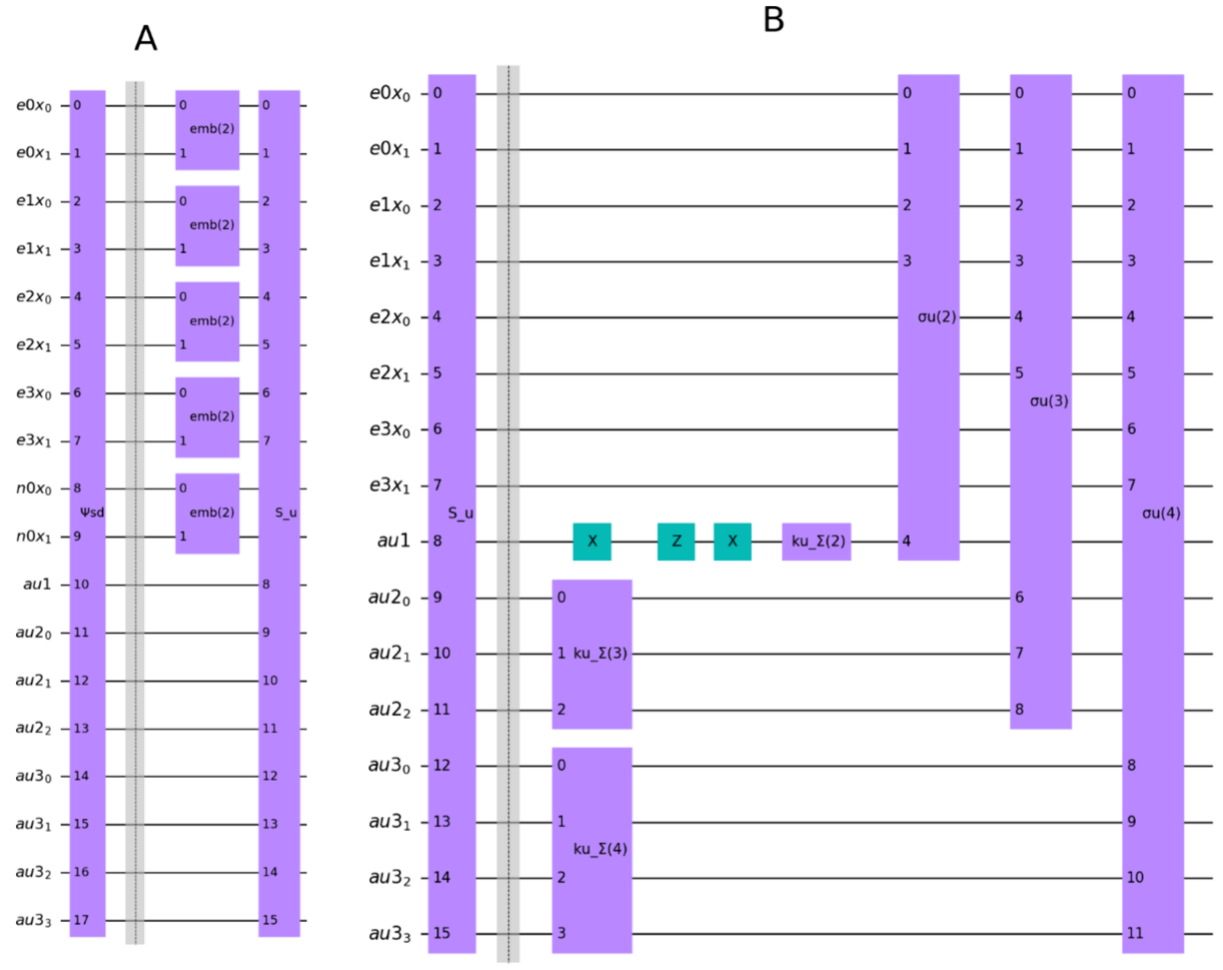


Figure 7. SD preparation gate, Ψ_{sd}, is based on unary encoding. (A) the Ψ_{sd} gate on the left and its internals on the right. The Ψ_{sd} gate consists of a number of state embedding gates, emb(2), and a unary-coded permutation gate, S_u. (B) Unary-coded permutation gate, S_u. This gate consists of unary-coded sequence preparation gates, $ku_\Sigma(k)$, that each produce a superposition state of $\sum_{i=0}^{k-1}|i\rangle/\sqrt{k}$ Index i is provided to the unary-coded conditional shuffling gates, $\sigma u(k)$. These gates conditionally swap the amplitudes of the coordinate qubits for electrons designated by the values of i and $k-1$ of the au$k$ registers. A sequence of XZX is inserted in front of the $ku_\Sigma(2)$ gate to cancel the global phase π that will be added by the three shuffle gates $\sigma u(2)\cdots\sigma u(4)$. The bit to apply this gate sequence is not restricted to the single qubit of register $au2$ and can be added to any qubit of B that has an initial value of $|0\rangle$. Any qubit of any register ak meets this condition.

elementary gates such as C-NOT gates. All such gates are visible in the resulting circuit diagram. This is useful when verifying the internal gates of an arithmetic operation.

- Gate-creating functions: These functions create a custom gate that implements an arithmetic operation. In Qiskit, a user-defined quantum circuit can be converted into a custom gate and can be used in the same way as standard gates provided by the quantum computer. In the resulting circuit diagram, the custom gate is represented by a single box, and the internal elementary gates are hidden. This is useful when verifying connections in a sequence of arithmetic gates because the internals of each gate are ignored.

In the context of the time-evolution calculation based on the Schrödinger equation, arithmetic operations are required for computing the potential energy terms of the Hamiltonian, which computes the inverse of the distance between two coordinates. This requires subtraction, multiplication, addition, square root, and division operations. For a one-dimensional model, the distance calculation can be simplified to an absolute value function. The choice of operations to be implemented is based on these needs. The design of the adder circuit is from Vedral et al.,[12] while the constant value adders and subtractors, and the absolute value were designed as part of this work. The rest are from Tomaru.[1] (See the references of Tomaru[1] for other circuit design proposals in the literature.)

- Adders: Several variations of adders are implemented. The unsigned adder takes two unsigned integer values of equal bit length and produces a value with one bit more to store the carry bit. The signed adder takes two signed integer values and produces a value with the same length as the inputs. A constant value adder takes the first input value in the form of qubits and the second input value in the form of a constant that is known at circuit generation time. Variations that take uneven length input values are also implemented.
- Subtractors: Variations analogous to the adders are implemented for subtractors. Unsigned, signed, and constant value subtractors, as well as their uneven length versions, are implemented.
- Multipliers: An unsigned multiplier, a signed multiplier are implemented.

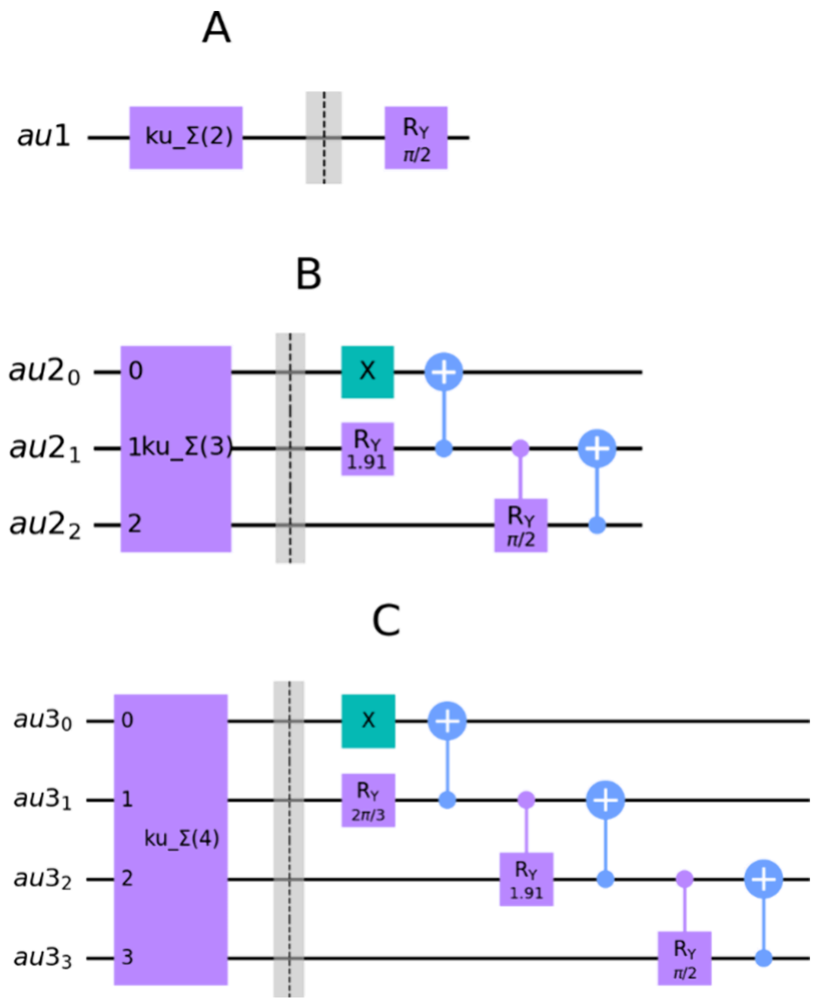


Figure 8. Unary-coded sequence preparation gates: (A) $ku_{\Sigma}(2)$, (B) $ku_{\Sigma}(3)$, (C) $ku_{\Sigma}(4)$.

- Dividers: An unsigned divider is implemented.
- Single operand functions: A signed square, a square root, and an absolute value are implemented.

4.3. Qubit Heap Component. The qubit heap component provides functions and classes to deal with temporary or ancilla qubit allocation and to create target qubit lists for custom gate parameters.

In Qiskit, a sequence of instructions on qubits can be grouped into an atomic unit. There are two types of atomic units: a custom instruction and a custom gate. The two are similar in functionality, but custom gates are unitary and have the additional capability of having control bits added afterward or being converted to a gate that has the inverse effect. All circuits can be converted to instructions, but only circuits that consist solely of gates without any instructions can be converted into gates. Here, we will refer to "gates", but the same discussion applies for instructions.

Custom gates are an effective means of organizing complex quantum circuits hierarchically and are used extensively in our development. They work well for circuits with a small number of input registers, such as arithmetic operations. However, when we tried to apply them to larger circuits, such as a gate to execute the Hamiltonian simulation, we found it challenging to avoid mistakes when passing argument qubits to the gate.

Custom gates take an array of qubit specifiers for the target of its operation. A qubit specifier is either an integer or a Qubit object. A Qubit object is not a physical qubit device but a class in the Qiskit SDK that holds information to identify a physical qubit by name. We mostly used Qubit objects as specifiers. The handling of the list of target qubit specifiers becomes error-prone as the number of parameters increases. Unlike in programming languages of classical computers, there is no compiler support to check whether the caller side arguments match the callee gate parameters. It is helpful to have some means to ensure that the two matches.

Besides the input and output qubits of the intended operation, the temporary qubits required by the gate must also be included in the list of parameters. Since the programmer is interested in the input and output of the gate, but not in the temporary qubits, the responsibility to supply the exact number of temporary qubits is a burden. Making matters worse is that the number of required temporary qubits is sometimes hard to determine. This is because custom gates can be nested, and the number of temporary qubits depends recursively on the nested inner gates. Any inner gate might be modified, and the number of temporary qubits they require might be changed as a result. Therefore, an automated mechanism for determining the number of temporary qubits is desired.

The situation is depicted in Figure 16. In the figure, circuit c invokes gate f, and gate f, in turn, invokes two gates g and h in sequence. Arrows represent the passing of target qubits from the caller to the callee. We classify target qubits into parameter

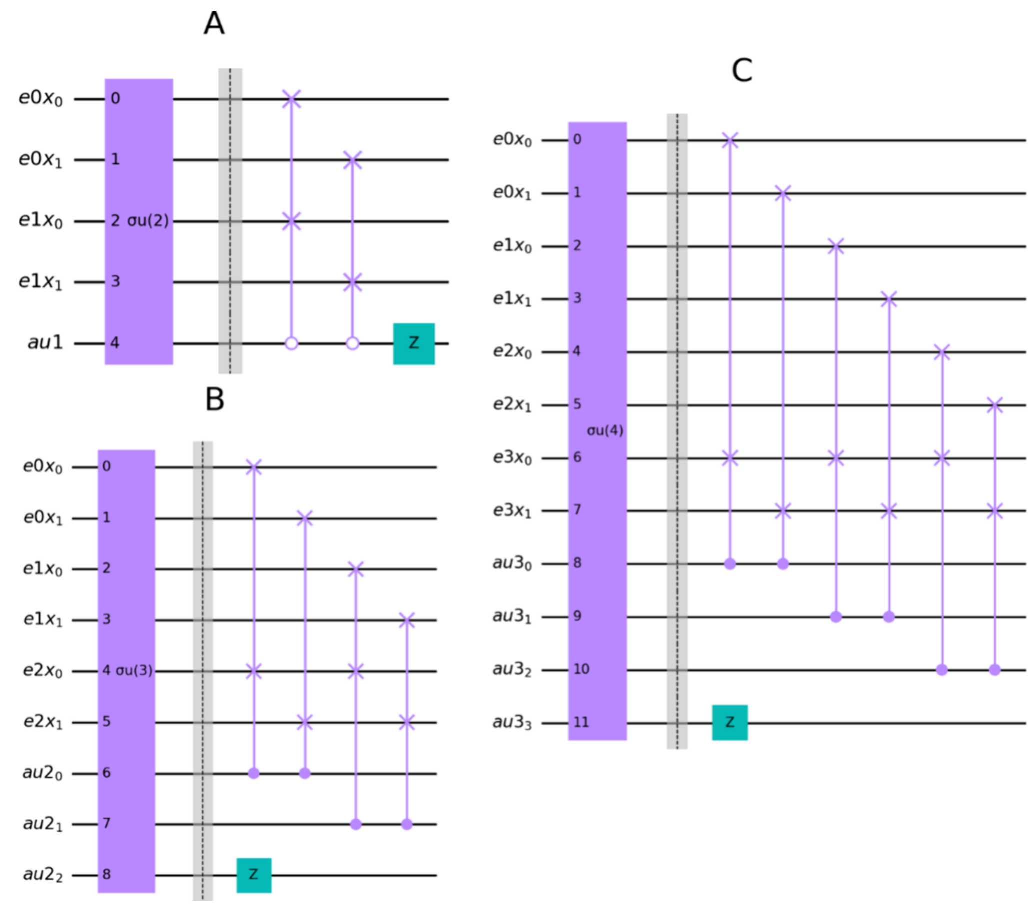


Figure 9. Unary-coded conditional shuffling gates: (A) $\sigma u(2)$, (B) $\sigma u(3)$, (C) $\sigma u(4)$.

target qubits and opaque target qubits. Parameter target qubits, denoted by white arrows, are target qubits whose meaning is known to the caller. Opaque target qubits, denoted by shaded arrows, are those whose meaning or usage is unknown to the caller. It only matters that the correct number of opaque qubits are passed and that their initial state is set to $|0\rangle$. When circuit c calls gate f, two parameters and four opaque target qubits are passed. Gate f uses the opaque qubits as two local qubits and two temporary qubits. Gate f passes two parameters and two opaque target qubits to gate g. Gate g uses the opaque qubits for two local qubits. Finally, gate f passes one parameter and one opaque target qubit to gate h. Gate h uses the opaque qubit for a local qubit.

Within a gate, the opaque qubits that have been provided by the caller of the gate are used for two purposes: (1) for use by the gate itself and (2) as a heap of temporary qubits from which opaque target qubits can be allocated when calling other subgates.

Deciding the heap size for each gate is nontrivial. For gate f, 2 is the maximum number of opaque qubits that is required at any single moment during the execution of this gate. Gate g requires 2 opaque qubits, and gate h requires 1, so the maximum of the two numbers is 2; therefore, the heap size for gate f is 2.

The orders of the parameter qubits of the caller and callee must match. The name given to the qubits for identification may be different between the caller side and callee side, such as $x1$, $y1$ at circuit c versus x, y at gate f. Some sort of mapping must take place.

In typical programming languages of classical computers, the temporary memory allocation and mapping of parameters are taken care of by function call mechanisms.[13] The callee defines a function. The caller passes parameters that match the function definition. The callee accepts those parameters. When temporary memory is required, the callee allocates space from the stack area, and no intervention from the caller is necessary. For the quantum computer case, the caller is responsible for the allocation of temporary qubits. After some exploration, we settled on a programming pattern involving two function calls to do one subcircuit call, as shown in Figure 17. The first call is on the callee side of a function named "bind". The caller provides qubit specifiers for the arguments. The bind function puts those qubit specifiers in a list that the callee circuit expects. That list is sent back to the caller in the form of a class named "Binding". The value is stored in a variable named "binding". This value is given as a parameter to the "invoke" function on the caller object. This is defined in the common superclass from which the caller class must inherit from. The invoke function incorporates the instructions in the callee's circuit into the caller circuit by using the target qubit list stored in the binding object. This pattern was inspired by the function objects found in the standard library for the programming language C++.[14]

The overall class structure to handle the aforementioned interaction is shown in the UML class diagram[15] in Figure 18. This is a set of classes that correspond to the scenario in Figure 16. The shaded classes, such as Frame and TemporaryQubitAllocator, are utility classes that can be reused for various circuits, and the white classes, such as FFrame and GFrame are classes that are specific to the circuit under concern. The white classes, CFrame, FFrame, GFrame, and HFrame correspond to circuit c and gates f, g, and h in Figure 16, and their main

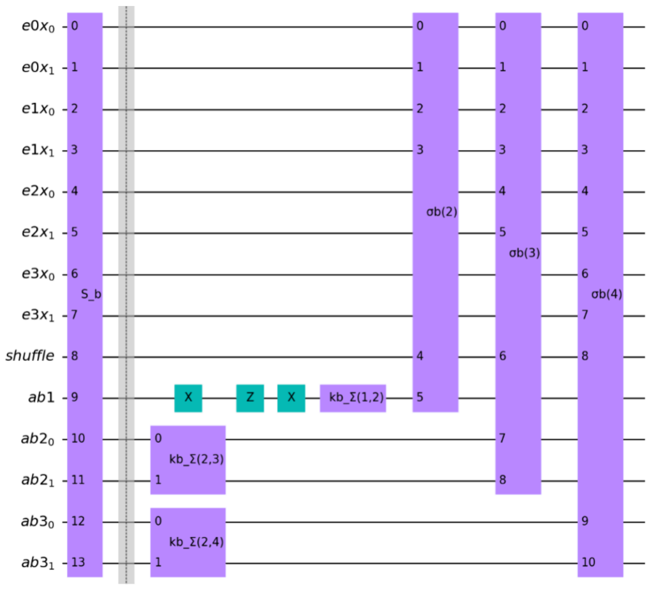


Figure 10. Binary-coded permutation gate S_b. After a sequence of XZX gates, or a "$-Z$" gate, three $kb_\Sigma(s, k)$ gates are applied to register set a. The results of the $kb_\Sigma(s, k)$ gates are input to the binary-coded conditional shuffling gates, $\sigma b(k)$. The XZX gates are for canceling an artificial global phase of π that is added when η is even. Each conditional shuffling gate $\sigma b(k)$ adds a global phase of π, as explained in Figure 12. The number of conditional shift gates is $\eta - 1$, so when η is odd, the sum of the phases cancels out, but when η is even, it equals π. That remaining phase is compensated by the XZX sequence to ease debugging by inspecting the statevector.

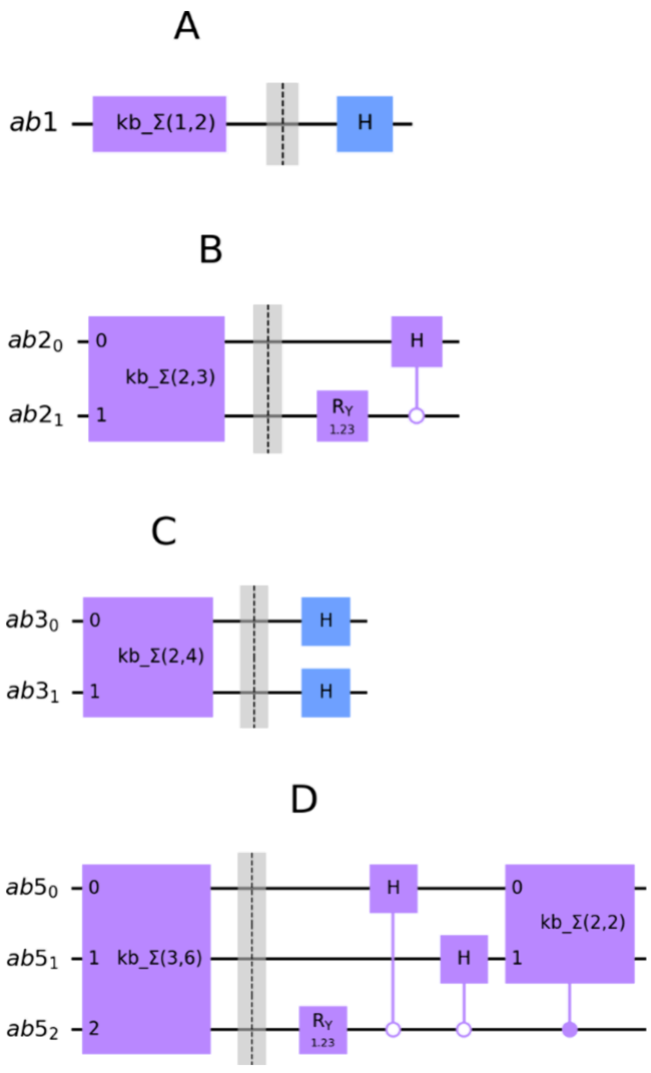


Figure 11. Binary-coded sequence preparation gates, kb_Σ (s, k). Each gate prepares an s-qubit state that has a superposition of $0..k-1$ in the binary representation. (A) $kb_\Sigma(1,2)$, (B) $kb_\Sigma(2,3)$, (C) $kb_\Sigma(2,4)$, (D) $kb_\Sigma(3,6)$.

responsibility is to hold QuantumRegister objects that identify the qubits that belong to each of the gates. Fframe, for example, has member variables x, y, b, and c that correspond to the qubits belonging to gate f. These four classes all extend a common superclass Frame, which provides functionalities for temporary qubit management and target qubit list preparation. The constructor method of each class has the responsibility of generating the instructions on their circuits. It recursively calls the constructors of the subcircuits, so the overall circuit is generated by calling the constructor of the CFrame class.

The class diagram of the utility class TemporaryQubitAllocator is shown in Figure 19. This class maintains a list of pooled qubits that can be borrowed from (allocated) for temporary use and returned after use. The number of qubits pooled in the list may grow to the maximum number of qubits that are simultaneously allocated during the circuit generation. It provides two methods, "allocate" and "free". The allocator has an associated QuantumCircuit object, and when qubits are added to the pool, they are also added to the circuit.

Appendix B describes the circuit generation sequence in detail using UML sequence diagrams.

4.4. AST Component. The abstract syntax tree (AST) component provides a set of classes that lie on top of the arithmetic component to provide a higher level of abstraction in programming style for arithmetic formulas.[10] The AST hides the details of arithmetic gate usage, such as keeping track of the input and output registers and allocating the temporary qubits. Once an AST is constructed, it can generate sequences of arithmetic gates to compute the formula on a quantum device.

An abstract syntax tree or an abstract parse tree is a term used in compiler design. A compiler reads the source code character-by-character to recognize the syntactic tokens and their grammatical structure; after that, it constructs a tree structure, called an abstract syntax tree, as an internal representation of the recognized code. A schematic example of an AST for the expression "sqrt($dx \times dx + dy \times dy + dz \times dz$)" is shown in Figure 20.

ASTs are especially useful for representing arithmetic expressions because arithmetic expressions have a recursive tree structure. Once an AST is constructed, the target code becomes straightforward to generate. To construct the AST, we used Python's customized arithmetic operators, whereby we can attach code that will be executed when operators such as "+" and "−" are used.

4.4.1. Node Objects. We designed the AST as a set of node objects that are tied together in the form of a tree. If there are two nodes a and b, then, in the program code "c = a + b", the "+" is a customized operator that produces a node for an add operation which points to "a" and "b" as its child nodes. The structure shown in Figure 21 is constructed in memory.

Later, when the circuit is generated, the tree will be traversed recursively and the gates for each node will be generated in postorder; i.e., each node is processed after all its child nodes

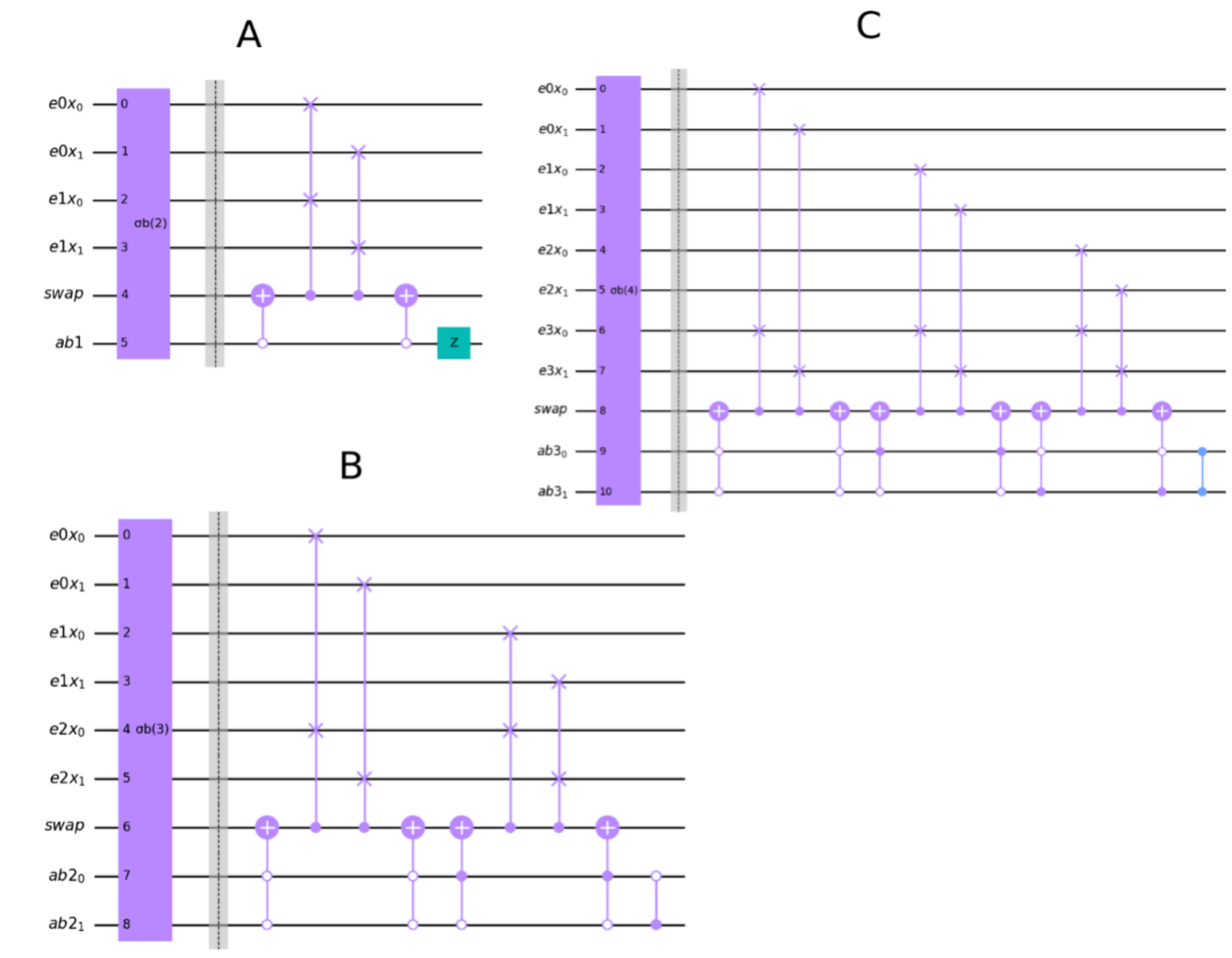


Figure 12. Binary-coded conditional shuffling gates $\sigma b(k)$: (A) $\sigma b(2)$, (B) $\sigma b(3)$, (C) $\sigma b(4)$.

have been processed recursively. For the case shown in Figure 21, the gates for a and b will be emitted, and after that, the gates for c will be emitted. The programmer does not need to worry about assigning the correct qubit registers for passing the results between the gates or allocating temporary work registers that the gates require.

4.4.2. Scope Class. A scope class is introduced in order to create the node objects and keep track of them. This class holds information related to AST nodes that belong to the same programming language scope. The scope object is obtained from the function crsq.ast.new_scope().

The scope object provides methods to create leaf nodes. In typical usage of AST, the first step is to create leaf node objects from QuantumRegister objects that are the input for the formula. In such cases, the register method is used to create an AST node that wraps the QuantumRegister, as shown in Figure 22.

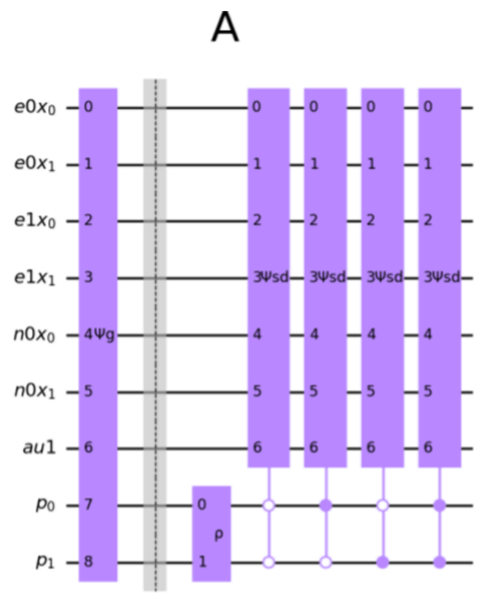
After the nodes are created, the circuit to compute the formula can be generated by using the build_circuit method. The inverse of the gates can also be generated by using the build_inverse_circuit method. This is often required in quantum algorithms when qubits must be returned to their initial state.

4.4.3. Supported Operators. The following operators are supported on node objects: + =, − =, *, /.

The scope object provides the following functions that take one operand: abs(x), square(x), and square_root(x).

4.4.4. Fixed-Point Arithmetic Support. All internal arithmetic functions are implemented as integer operations. Integer operations can be used as fixed-point fractional number operations by statically keeping track of the decimal point's location. The range of the value must fit in the register at all of the steps of the sequence of operations. A function to ensure this is implemented in the AST nodes. The numbers of fractional bits and the total bits (fractional bits and whole number bits together), a flag denoting whether the value is signed or unsigned, and the upper and lower bounds of the stored value are recorded on variables of the nodes. These variables are used for computing the total and fractional bit counts of the result of an operation or for checking the compatibility of operands before an operation. Table 5 shows the constraint on the bit count of both operands and the resulting bit count. The number of total bits is further adjusted based on the possibility of overflows, which can be determined from the value range information.

4.4.5. Bit Count Adjustment. In a fixed-point calculation, there may be a need to add or remove bits to or from the result of a calculation step before feeding that value to the next step. For this purpose, an "adjust_precision" method is created on AST nodes. This method allows bits to be added to or removed from either the LSB end or the MSB end of the value represented by the node. As an example, when calculating the square root of small integer values, it may be useful to add 2 or 4 bits below the



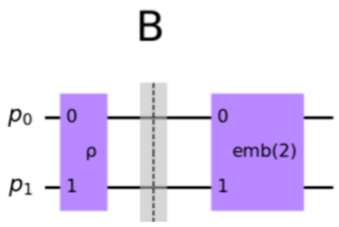


Figure 13. General state preparation gate, Ψ_g, and its internals. (A) The Ψ_g gate is shown on the left as a unit, and its internals are on the right. The 2-bit version of the configuration preparation gate, ρ, prepares p_0 and p_1 that together store the probability amplitude of four different states. p_0 and p_1 control the 4 SD preparation gates, Ψ_{sd}, and each is given a different set of initial orbital data. (B) The configuration preparation gate, ρ, and its internals. The state embedding gate, emb(2), sets the probability amplitudes.

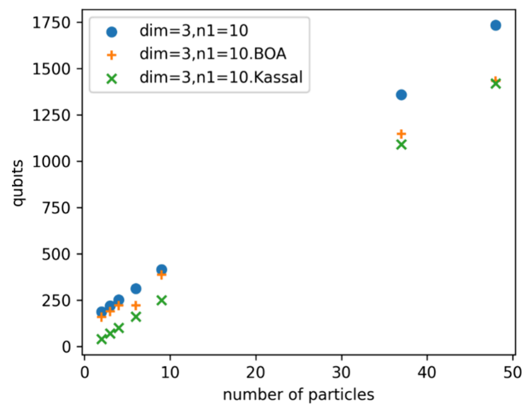


Figure 14. Qubit counts with respect to different numbers of particles. Number of particles refers to the sum of the numbers of electrons and nuclei. The plots correspond to H, He, Li, H + H_2, and O, glycine (C_2NOH_3), and alanine(C_3NOH_5). The plus(+) plots are results for when the circuit generator was configured to apply the BOA, i.e., to treat the coordinates of nuclei as fixed ones and omit qubits for those coordinates. The x plots are based on the formula by Kassal.[4]

decimal point to make it a fractional number before applying the square root gate. In this way, the result of the square root will have 1 or 2 digits below the decimal point.

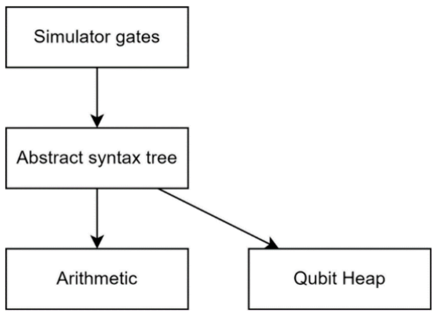


Figure 15. Program structure.

4.5. Simulator Gate Component. The simulator gate component is the top-level component of the structure of the simulator program shown in Figure 15. The hierarchy of the Python classes that produce the gates is shown as a tree-structure call graph in Figure 23. It contains gates that are specific to the Hamiltonian simulation. All of the other low-level components are general-purpose and can be used in other programs besides Hamiltonian simulations. The classes in the simulator gates component generate custom gates that are shown in different configurations in Figures 1 and 13. The generated gates are described in the following subsections.

4.5.1. Wave Function Preparation Gates. Wave function preparation gates implement the initial wave function on the coordinate qubits. The following gates are included: the state embedding gate, emb(n), which embeds a single orbital to a coordinate register; the SD preparation gate, Ψ_{SD}, and its subcomponents (the unary-coded gates S_u, $ku_\Sigma(k)$, $\sigma_u(k)$ or the binary-coded gates S_b, $kb_\Sigma(k)$, $\sigma_b(k)$), which work together to permute the amplitudes of the coordinate register to implement an SD; the general state preparation gate, Ψ_g, which creates a superposition of multiple SDs based on different sets of orbital data.

4.5.2. Time Evolution Gates. The time evolution gates implement time evolution based on the Suzuki–Trotter formula. The time evolution gates consist of the electron P.E. time evolution gate, Θ_{ep}, electron K.E. time evolution gate, Θ_{ek}, nucleus P.E. time evolution gate, Θ_{np}, nucleus K.E. time evolution gate, Θ_{nk}, n-element quantum Fourier transform gate, nQFT, and its inverse, $n\mathrm{QFT}^\dagger$.

Regarding the P.E. time evolution gates, the AST module is used to generate circuits for performing the binary arithmetic used in the Hamiltonian calculation. For all pairs of particles, electrons, or nuclei, arithmetic gates are used to compute the Coulomb energy due to the pair, and the phase is shifted using the resulting value in a binary representation with a series of phase gates bit-by-bit. Then, the reverse operation for the calculation is executed, and all of the qubits are returned to their original states except for the phase. The whole P.E. term is calculated by processing all particle pairs in this way.

When calculating the Coulomb energies, the distance between the two particles appears in the divisor of the formula. This raises the possibility of division by zero. There are several ways to deal with this problem. The first is to replace zero with a constant value Δq that is smaller than the discretization step δq.[16] The second is to replace the division result with zero, which is a valid solution if all electrons have the same spin.[1] The third is to compute the Coulomb energy term in p-space, which will eliminate the problem.[17] We implemented the first and second methods and have left the third for future work.

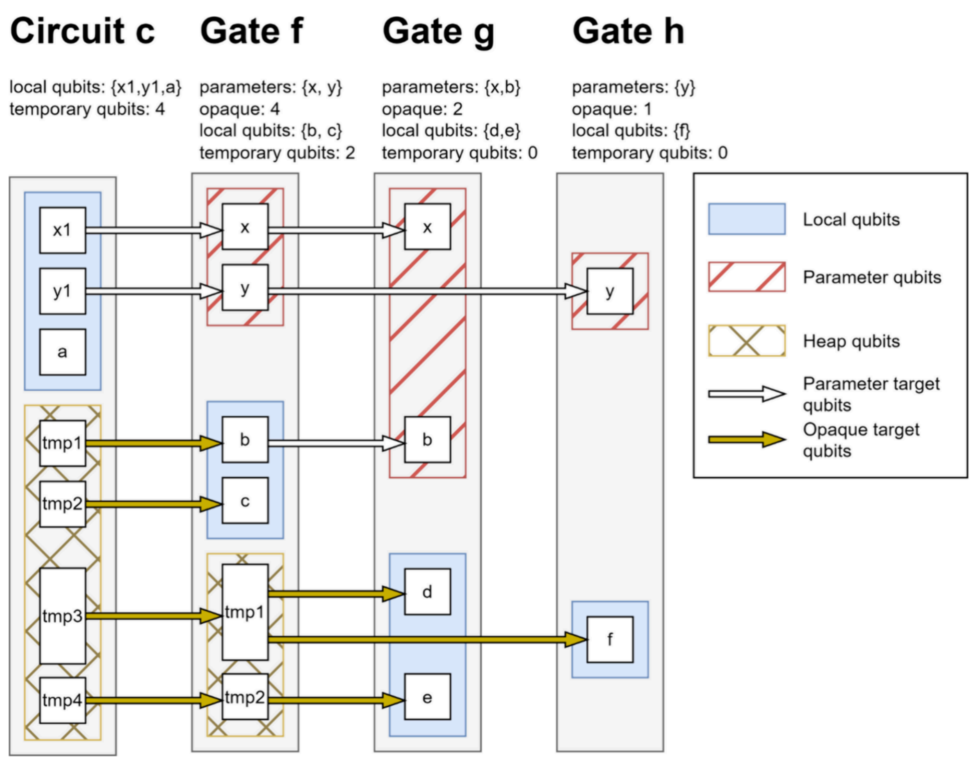
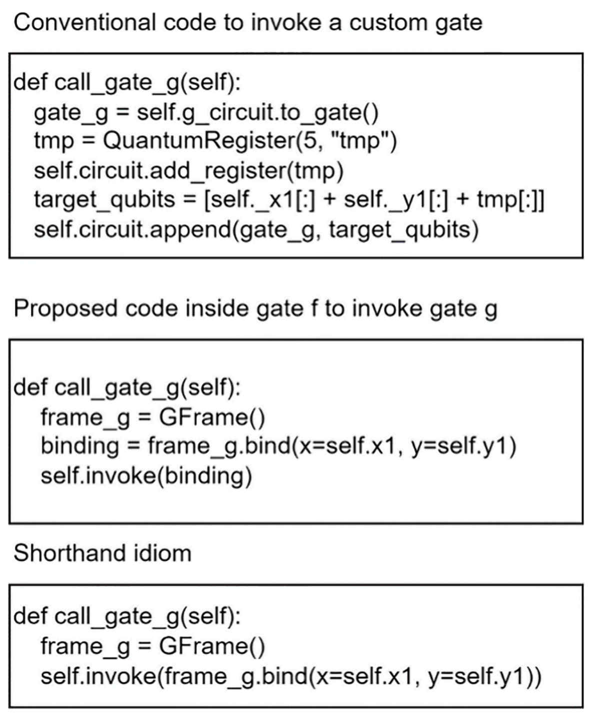


Figure 16. Problem of target qubit allocation. Example of nested circuit gates.



Conventional code to invoke a custom gate

```
def call_gate_g(self):
  gate_g = self.g_circuit.to_gate()
  tmp = QuantumRegister(5, "tmp")
  self.circuit.add_register(tmp)
  target_qubits = [self._x1[:] + self._y1[:] + tmp[:]]
  self.circuit.append(gate_g, target_qubits)
```

Proposed code inside gate f to invoke gate g

```
def call_gate_g(self):
   frame_g = GFrame()
   binding = frame_g.bind(x=self.x1, y=self.y1)
   self.invoke(binding)
```

Shorthand idiom

```
def call_gate_g(self):
   frame_g = GFrame()
   self.invoke(frame_g.bind(x=self.x1, y=self.y1))
```

Figure 17. Code fragments for custom gate invocation in conventional and proposed forms.

Regarding the K.E. time evolution gates, since the calculation of p_i^2 is simple in momentum space, we do not use the arithmetic gates to compute the square. Instead, we implement the square operation directly with bit-wise phase gates and controlled phase gates.

The *n*-element quantum Fourier transform gate and its inverse apply the QFT and inverse QFT gates to the coordinate qubits to convert the wave function from the positional representation to the momentum representation and back again. The QFT gates provided by Qiskit were used.

5. VERIFICATION

Verifications of the circuits were done by running the gates on the quantum computer simulator provided with the Qiskit SDK. The Aer simulator implements several simulation methods; the basic "statevector" method was used.[7] This method allows inspection of the state vector data, which is impossible with an actual quantum computer. Execution of the code and inspection of the result was done automatically in the form of a test suite run by the pytest tool.[11] In this section, we describe what kind of tests were performed on each component. We also show some illustrative graphs of the results for a subset of tested cases.

5.1. Verification of Arithmetic Gates.

We wrote arithmetic gate generator functions that could generate arithmetic gates with arbitrary bit counts. We could not test the generated gates for all infinitely possible bit counts. For testing purposes, we chose a range of bit counts so that all of the conditional statements within the generator code could run at least once. For those selected bit counts, the arithmetic gate was generated, and all possible input values were verified on it. The input data used in the code in the test suite is a single state, not a superposition because when an unexpected output is produced due to a bug, it would be much easier to diagnose the problem for a nonsuperposition value. For example, to test an adder gate that takes two input values on qubit registers X and Y, and produces an output value on register Z, the test program would choose a pair of values x and y, and prepare the input state as $|x\rangle_X|y\rangle_Y|0\rangle_Z$. After the adder is applied, the resulting state vector is obtained from the simulator and compared with the expected outcome $|x\rangle_X|y\rangle_Y|x + y\rangle_Z$. If the result matches the expected value, then the test is a success. This was repeated for all combinations of x and y.

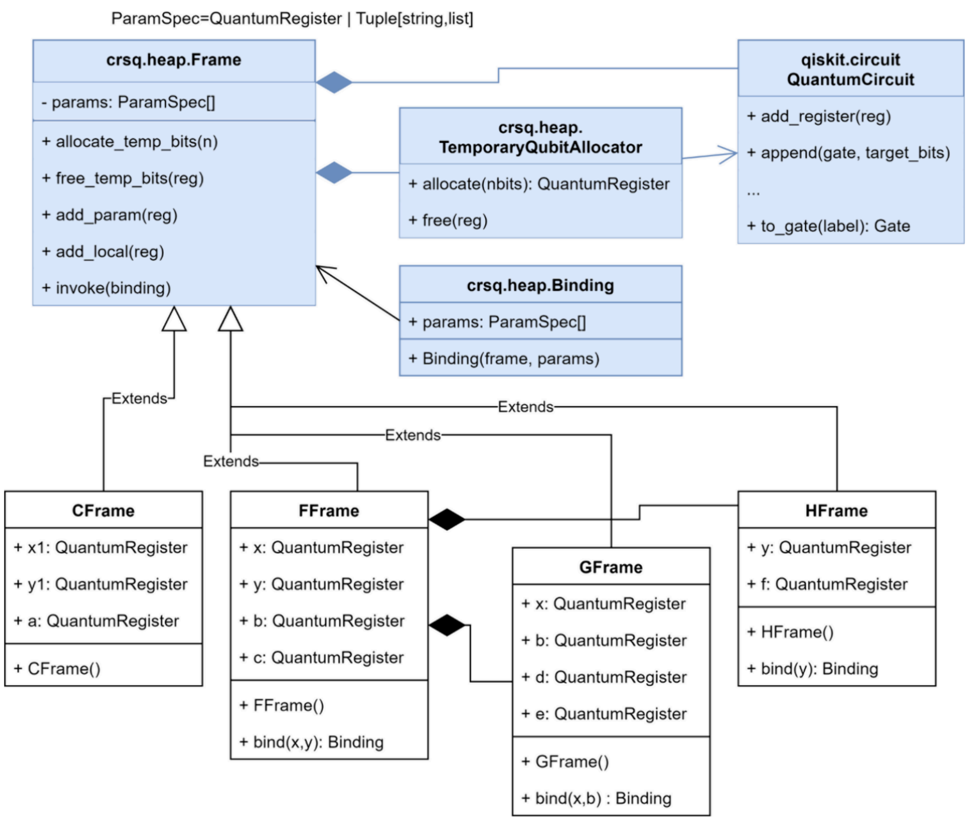


Figure 18. Class diagram for the Frame class and related classes.

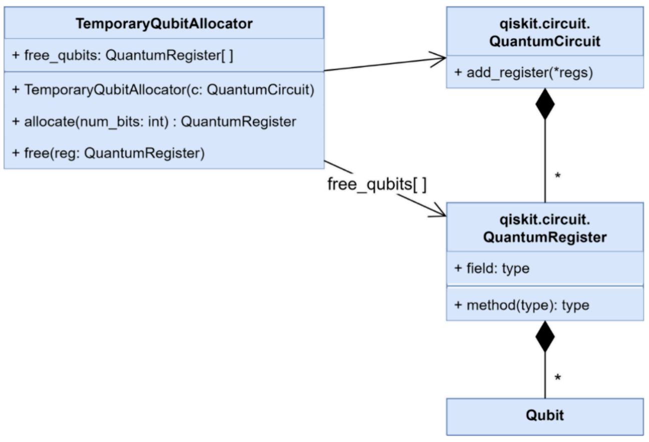

TemporaryQubitAllocator
+ free_qubits: QuantumRegister[ ]
+ TemporaryQubitAllocator(c: QuantumCircuit)
+ allocate(num_bits: int) : QuantumRegister
+ free(reg: QuantumRegister)
qiskit.circuit. QuantumCircuit
+ add_register(*regs)
free_qubits[ ]
*
qiskit.circuit. QuantumRegister
+ field: type
+ method(type): type
*
Qubit

Figure 19. Class diagram of the TemporaryQubitAllocator class.

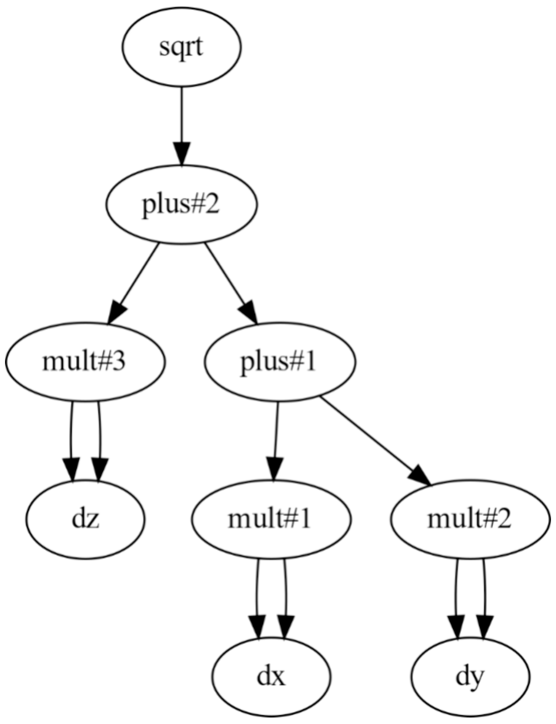


Figure 20. Schematic example of an AST for the expression "sqrt($dx \times dx + dy \times dy + dz \times dz$)".

Figures 24 and 25 show selected graphs of the input and output of the arithmetic gates. The graphs were created by programs written for each arithmetic operation. Each program generated a circuit where input qubits were prepared, and the arithmetic operation was executed on those qubits. Since the arithmetic gates used for obtaining Figures 24 and 25 have been debugged as described, we prepared the input qubits in a superposition of all possible input values. The program then ran the circuit on a simulator and extracted the state vector from the simulator. We checked each term of the state vector whether it has a nonzero amplitude (amplitude with a norm greater than a threshold). For example, to produce a graph for $Z = X + Y$, the input state vector was prepared as $\frac{1}{\sqrt{N}}(|0\rangle + |1\rangle + \cdots + |N-1\rangle)_X \otimes \frac{1}{\sqrt{N}}(|0\rangle + |1\rangle + \cdots + |N-1\rangle)_Y \otimes |0\rangle_Z$, which is a superposition of all possible values for registers X and Y, and Z set to 0. When the adder gate was applied to this state, the resulting state can be expressed as $\sum c_i|x_i\rangle_X|y_i\rangle_Y|z_i\rangle_Z$. From the superposition state, we extracted only the values x_i, y_i, and z_i that satisfy $|c_i| > \epsilon$, where ϵ is chosen to be large enough to filter out

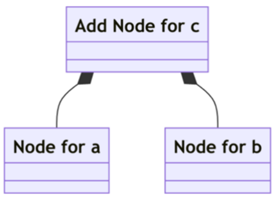


Figure 21. Class diagram of an AST for a + b.

noise entries. The extracted value z_i is the result of $x_i + y_i$. Figures 24 and 25 plot the extracted x_i, y_i, and z_i for various operations.

5.2. Verification of Abstract Syntax Trees. Abstract syntax tree classes were tested by creating a minimal tree for each operation type and having the tree generate the quantum circuit, which was then run, and the resulting state vector was inspected. The goal of this test was to run all of the code in the AST component, not necessarily all of the code in the arithmetic component. Thus, the test input was selected for verifying different combinations of operations but not all combinations of numerical values for the same operation.

5.3. Verification of Time-Evolution Calculation. The four types of time evolution gates, Θ_{ep}, Θ_{ek}, Θ_{np}, and Θ_{nk}, were tested by running them on tailored wave functions, as described below.

For the P.E. time evolution gates, a nucleus was placed at $X_0 = 8$, and an electron at a superposition of several positions $x_0 = 0,1, ...,2^{n_1} - 1$. For this discussion, we will assume a 1-D model, where n_1 represents the bit count of the spatial coordinates. This corresponds to the wave function $|\Psi\rangle = |X_0\rangle|x_0\rangle = |8\rangle \otimes \frac{1}{\sqrt{2^{n_1}}}(|0\rangle + |1\rangle + \cdots + |2^{n_1} - 1\rangle)$. When the P.E. time evolution gate is applied to this state, each element of the tensor product has its phase shifted by $\Theta_{ep}(X_0, x_0)$: $\Theta_{ep}(|X_0\rangle \otimes |x_0\rangle) = \exp\left(-iH_{ep}(X_0, x_0)\delta t/\hbar\right)|X_0\rangle|x_0\rangle$. We verified the correctness of the circuit by inspecting the phase shift of each element of the resulting state vector. The result of such a test with $|x_0\rangle$ set as $\frac{1}{\sqrt{16}}(|0\rangle + |1\rangle + \cdots + |15\rangle)$ is shown in Figure 26A. The model parameters in this case are as follows: 4-bit coordinates $(n_1 = 4)$, $L = 2$, $\delta q = L/2^{n_1} = \frac{1}{8}$1 electron with charge $q_e = -e$ at positions $x_0 = 0,1, ...,2^{n_1} - 1$, and 1 nucleus with charge $Q_A = e$ positioned at $X_0 = 8$ (i.e., $Q_0 = 8\delta q$). This configuration is similar to the first circuit example, where the Θ_{ep} gate was shown in Figure 3, with the bit-count n_1 changed from 3 to 4. The division operation in the Coulomb potential energy was done using $(n_1 + 1)$-bit fixed decimal point numbers. The dividend was 1.0, which was represented as 1.0000 with four bits used for the fractional part. The divisor was the distance between the two particles with no fractional part. One bit was added as the MSB to make the divisor and dividend bit counts match (see the caption of Figure 3 for an explanation of this bit).

Table 5. Constraints and Resulting Bit Counts for Fixed-Point Arithmetic[a]

operation	constraint	result: total bits	result: fractional bits
add, subtract	$n_1 \geq n_2, f_1 = f_2$	$n_1 + 1$	f_1
multiply		$n_1 + n_2$	$f_1 + f_2$
divide	$n_1 \geq n_2$	n_1	$f_1 - f_2$
square		$2n_1$	$2f_1$
square root	n_1, f_1 are even	$n_1/2$	$f_1/2$
absolute		n_1	f_1

[a]n_1 and f_1 are the total bit count and fractional bit count of the left operand, while n_2 and f_2 are those of the right operand.

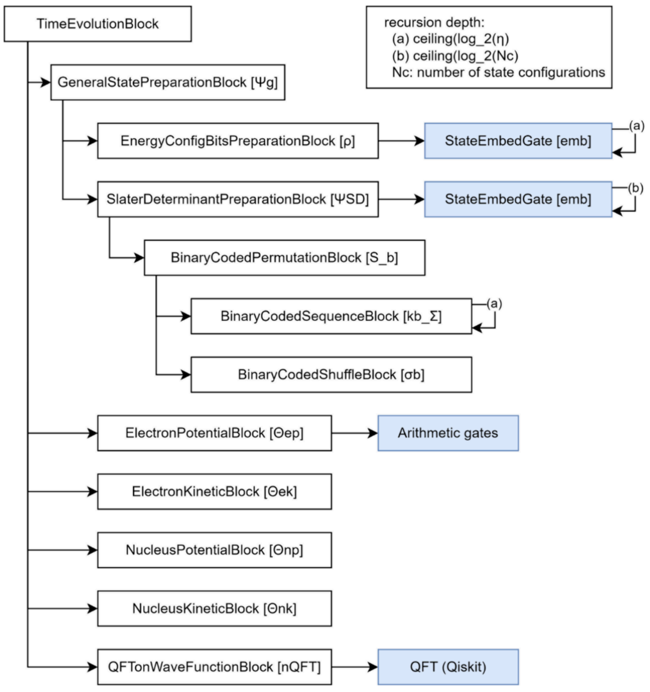

TimeEvolutionBlock
recursion depth:
(a) ceiling(log_2(η))
(b) ceiling(log_2(Nc))
Nc: number of state configurations
GeneralStatePreparationBlock [Ψg]
EnergyConfigBitsPreparationBlock [ρ]
StateEmbedGate [emb]
(a)
SlaterDeterminantPreparationBlock [ΨSD]
StateEmbedGate [emb]
(b)
BinaryCodedPermutationBlock [S_b]
BinaryCodedSequenceBlock [kb_Σ]
(a)
BinaryCodedShuffleBlock [σb]
ElectronPotentialBlock [Θep]
Arithmetic gates
ElectronKineticBlock [Θek]
NucleusPotentialBlock [Θnp]
NucleusKineticBlock [Θnk]
QFTonWaveFunctionBlock [nQFT]
QFT (Qiskit)

Figure 23. Call graph of python classes that belong to the simulator gates component. Arrows denote usage or a "call" of the arrow target by the arrow origin. Shaded boxes are general purpose classes. The names on the boxes are python class names. The corresponding gate names are shown in brackets. An arrow pointing to itself denotes a recursion. Depending on the configuration, some elements may be omitted or replaced by alternative classes. The maximum depth of the nested structure, excluding calls due to recursion, is four, as in the calls to BinaryCodedSequenceBlock and BinaryCodedShuffleBlock.

```
from qiskit import QuantumCircuit, QuantumRegister
import firstq.ast as ast
qc = QuantumCircuit()
rega = QuantumRegister(4,'a')
regb = QuantumRegister(4,'b')
qc.add_register(rega, regb)

scope = ast.new_scope(qc)
ast_a = scope.register(rega)  # create leaf node for A
ast_b = scope.regitser(regb)  # create leaf node for B
ast_b += ast_a                # create an in-place add node for B
scope.build_circuit()         # emit the gates to the circuit qc.
```

Figure 22. Usage of AST module.

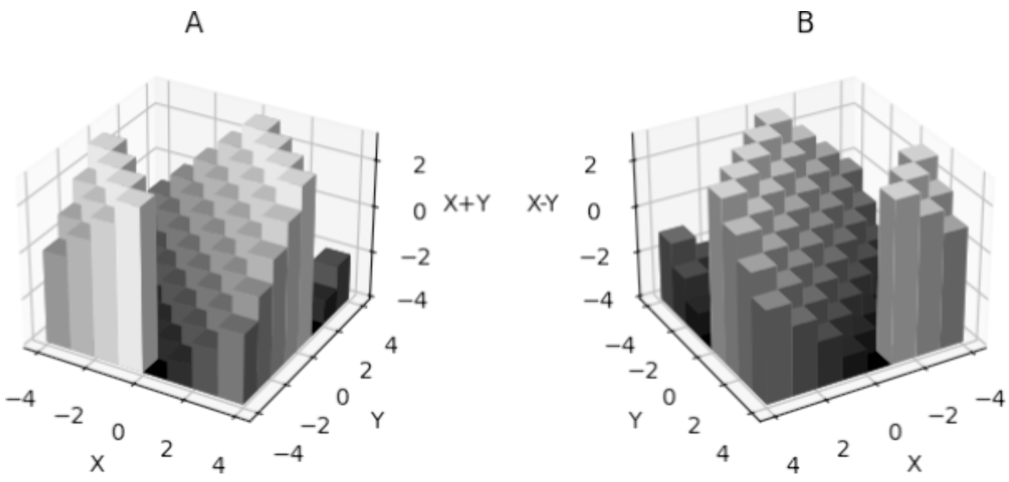


Figure 24. Graphs of the input and output of an arithmetic gate. (A) Results of a signed adder gate, $Z = X + Y$, (B) Results of a signed subtractor gate, $Z = X - Y$. In both graphs, input registers X and Y were initialized to a superposition of all values expressible by a 3-bit signed integer, i.e., $1/\sqrt{8}\,\{|-4\rangle + |>-3\rangle + \cdots + |3\rangle\}$; then the signed adder or signed subtractor gate was executed on those inputs. The combination of X, Y, and Z was gathered in the resulting state vector for components with amplitude norms greater than a threshold. Finally, the gathered combinations are plotted.

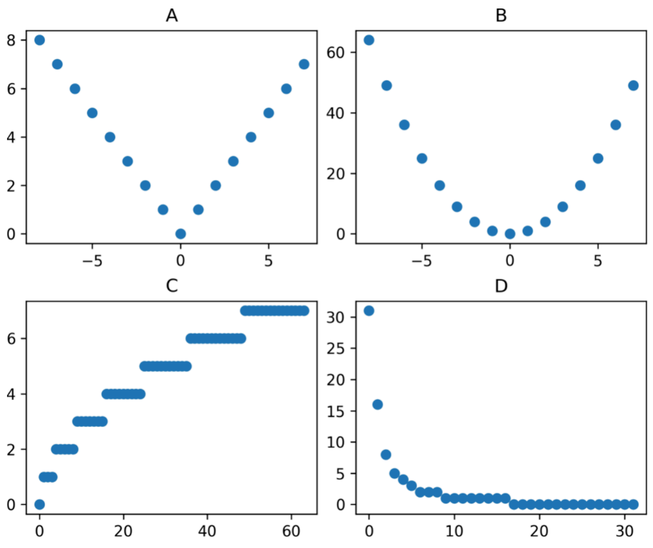


Figure 25. Graphical results of the arithmetic gates. (A) Results of the absolute value gate, $Y = |X|$. Note that $|-8| = 8$. This means that the input is a 4-bit signed value, but the output has to be treated as 4-bit unsigned value. (B) Results of square gate, $Y = X^2$, and (C) Results of square root gate, $Y = \lfloor\sqrt{X}\rfloor$, (D) Results of unsigned divider gate: $Y = \lfloor 16/X \rfloor$. For the case of division by zero, the resulting value is the maximum value expressible by the output register, which is $2^5 - 1 = 31$ in this case. In all graphs, the input register X was initialized to a superposition of all possible input values for the arithmetic gate; then, the arithmetic operation was executed on that input. The combination of X and Y were gathered in the resulting state vector for components with amplitude norms greater than a threshold; and those combinations were plotted.

The formula to obtain the phase shift value, $\Theta_{ep}(X_0, x_0)$, includes a parameter δt, which can be chosen arbitrarily. For testing purposes, we chose δt such that the resulting phase shift value will be an easily recognizable value, for example, π; we chose δt such that when the electron and nucleus position is δq apart, i.e., $x_0 = 7$ or 9, the phase shift will be π. This will result in $\delta t = \frac{\delta q}{e^2}\hbar\pi$. Figure 26 A shows the phase shift value for this choice of δt. The phase shift of the wave function for $x_0 = 9$ is

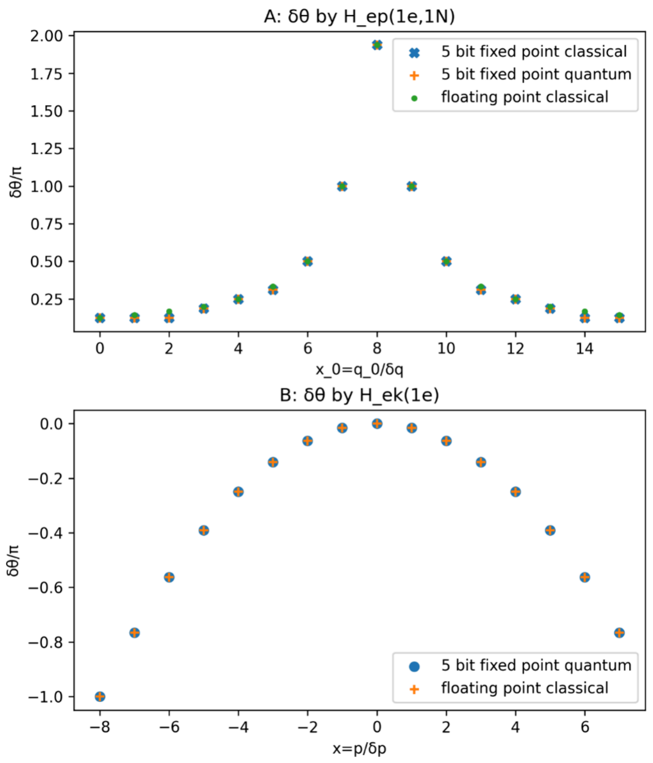


Figure 26. Test results of the time evolution calculation. (A) Phase shift of the potential-energy term of an electron−nucleus pair, (B) phase shift of the kinetic-energy term of an electron. The plots labeled "5 bit fixed point classical" are values computed by the Python code using integer arithmetic to emulate a fixed point calculation. The values should have the same rounding errors as the integer operations by the generated quantum circuit. The plots labeled "5 bit fixed point quantum" are values computed by the generated quantum circuit being run on a (noise-free) quantum computer simulator. The plots labeled "floating point classical" are values computed by Python code using the floating point arithmetic. This has a much smaller rounding error compared to the results of fixed-point arithmetic.

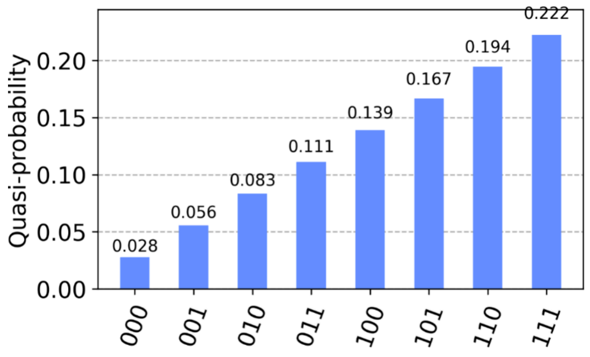


Figure 27. Test result of the 3-bit state embedding gate, emb(3), which sets the normalized data to the qubits. Distribution data of $\phi_{k-1} = \sqrt{k}$, which is not normalized, were given as input.

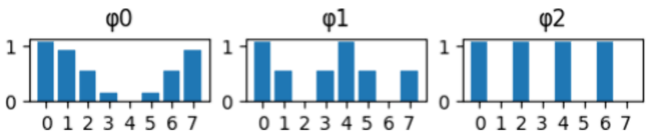


Figure 28. Input wave function data for the test of the Slater determinant preparation gate, Ψ_{sd}.

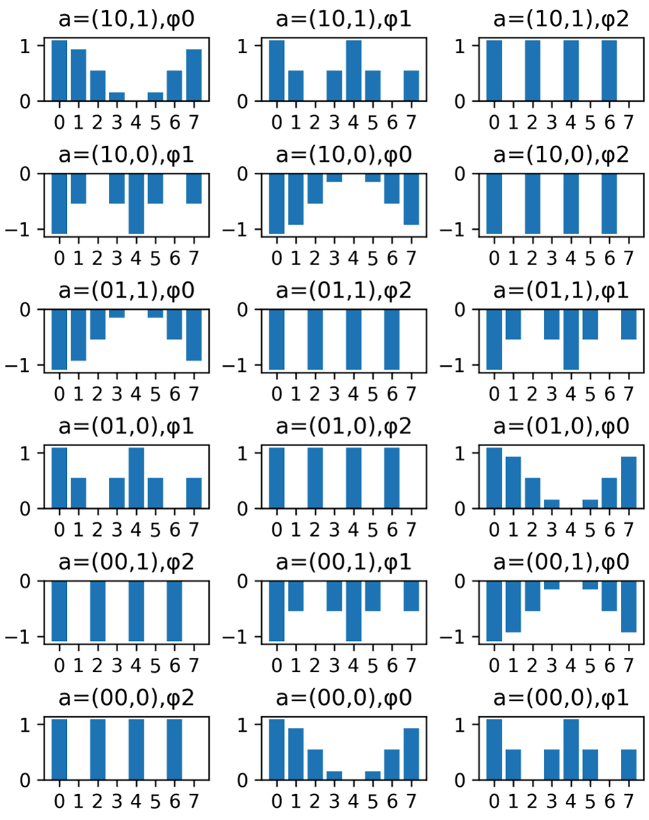


Figure 29. Test results for the Slater determinant gate.

$\delta\theta = -H_{ep}(8, 9)\delta t/\hbar = -\frac{q_e Q_A}{|8-9|\,\delta q}\frac{\delta q}{e^2}\frac{\hbar\pi}{\hbar} = \pi$. This is confirmed in the upper part of Figure 26A, where $x_0 = 9$ and $\delta\theta/\pi = 1.0$. Thus, the expected phase shift value was obtained. This means that the arithmetic gates and phase gates shown in Figure 3 are correctly generated.

Since the arithmetic is performed using fixed-point numbers with only 5 bits on the simulated quantum computer, there is a discrepancy between the result computed by the quantum gates and the result computed on a classical computer using 64-bit floating point numbers. This can be seen, for example, on q = 2,5,11,13 where the "+" plots and the dot plots slightly disagree. This discrepancy is due to rounding errors and is not due to any characteristic of quantum computers or the quantum computer simulator. The division-by-zero case $q = 8$ produces the expected value of 1.1111 in binary (1.9375 in decimal).

The above discussion is for a one-dimensional model, which does not require square and square root gates to calculate the distance between the two particles. We do not have test results for two- or three-dimensional models, which use square and square root gates, because a minimal circuit for a two-dimensional model requires more than 40 qubits and could not be tested with the hardware resources available. Square gates and square root gates were tested in isolation but not as a combined circuit to calculate the 2-norm.

The K.E. time evolution gates can be verified with a similar approach. Here, a 1-D model with 4-bit coordinates ($n = 4$) with one electron and no nuclei was used. This corresponds to the circuit of Θ_{ek} that was shown in Figure 4, with n_1 changed from 3 to 4. In the momentum representation, the electron is given a momentum value of a superposition of several index values x = 0,1,2, ...,2^{n-1}, where the magnitude of the momentum $p = x\delta p$. Here, the phase shift resulting from the K.E. time evolution gate is $\delta\theta = -\frac{p^2}{2m_e}\frac{\delta t}{\hbar} = -\frac{(x\delta p)^2}{2m_e}\frac{\delta t}{\hbar}$. As with the P.E. case, we can choose δt arbitrarily. For testing purposes, we choose δt so that the phase shift value is $-\pi$ for the largest value of $|x|$, which is 8. This will result in $\delta t = \frac{2}{\pi}\frac{m_e}{\hbar}\delta q^2$. Figure 26B shows the phase shift for the K.E. term with this choice for δt. The phase shift for $x = -8$ is $\delta\theta = -\pi$. This can be confirmed in the lower part of Figure 26B where $x = -8$ and $\delta\theta/\pi = -1$. The expected phase shift value was obtained, and thus, the circuit in Figure 4 was verified to be correct.

Since the K.E. term does not involve any arithmetic gates, there is no discrepancy due to rounding errors.

5.4. Verification of Initial State Preparation.

The top-level gate of the initial state preparation is the general-state preparation gate, Ψ_g. This gate has two subcomponents: the SD preparation gate, Ψ_{sd}, and the configuration preparation gate, ρ. The Ψ_{sd} gate is further decomposed into the state embedding gate, emb(n), and the permutation gate, in either its unary-coded version, S_u, or its binary-coded version, S_b. These gates were tested.

The state embedding gate was tested by applying the gate on a set of qubits for test distribution data and then inspecting the state vector. The following is an example of the resulting state vector for a test input of amplitude data set to an unnormalized distribution $\phi_{k-1} = \sqrt{k}$ for $k = 1\cdots8$. The expected resulting state vector is normalized to $|\phi\rangle = \sum_{k=1}^{8}\sqrt{k/36}\,|k-1\rangle$. The output shown in Figure 27 corresponds to these values.

The SD state preparation gates were tested by running the circuit on test wave function data for an orbital and inspecting the state vector by extracting the wave function entangled with each value of register set a. Example plots are shown in Figures 28 and 29. Figure 28 shows the input, which consists of data on three wave functions, and Figure 29 shows the results of the permutation performed by S_u on the data. The first row is identical to the input, and it corresponds to the identity permutation. The second row is the result of the permutation τ =

Table A-1. Bit Assignments for Arithmetic Gates

name	1st operand	2nd operand	output	temporary qubits	ancilla qubits	number of qubits
uadd(n)	$a_{n-1}...a_0$	$b_n...b_0$	$b_n...b_0$	$cr_{n-2}...cr_0$		$3n$
uaddv(m,n) ($m \leq n$)	$a_{m-1}...a_0$	$b_n...b_0$	$b_n...b_0$	$cr_{n-2}...cr_0$		$m + 2n$
sadd(n)	$a_{n-1}...a_0$	$b_{n-1}...b_0$	$b_{n-1}...b_0$	$cr_{n-2}...cr_0$		$3n - 1$
scoadd(n,y)		$b_{n-1}...b_0$	$b_{n-1}...b_0$	$cr_{n-2}...cr_0$		$2n - 1$
usub(n)	$b_n...b_0$	$a_{n-1}...a_0$	$b_n...b_0$	$cr_{n-2}...cr_0$		$3n$
ssub(n)	$b_{n-1}...b_0$	$a_{n-1}...a_0$	$b_{n-1}...b_0$	$cr_{n-2}...cr_0$		$3n - 1$
umult(n)	$a_{n-1}...a_0$	$b_{n-1}...b_0$	$d_{2n-1}...d_0$	$cr_{2n-1}...cr_0$		$6n - 1$
smult(n)	$a_{n-1}...a_0$	$b_{n-1}...b_0$	$d_{2n-1}...d_0$	$cr_{2n-1}...cr_0$		$6n - 1$
ssquare(n)	$a_{n-1}...a_0$		$d_{2n-1}...d_0$	$cr_{2n-1}...cr_0$		$5n - 1$
udiv(m,n) ($m \geq n$)	$z_{m-1}...z_0$	$d_{n-1}...d_0$	$zz_{n-1}...zz_0 \mid z_{m-1}...z_0 = q_{m-1}...q_0 \mid r_{n-1}...r_0$	$cr_{n-2}...cr_0$	$zz_{n-1}...zz_0$	$m + 3n - 1$
abs(n)	$a_{n-1}...a_0$		$a_{n-1}...a_0$	$cr_{n-2}...cr_0$	sgn	$2n$
sqrt(n)	$z_{2n-1}...z_0$		$r_{n-1}...r_0$	$cr_{2n-2}...cr_0$	$w_n...w_0$	$6n$

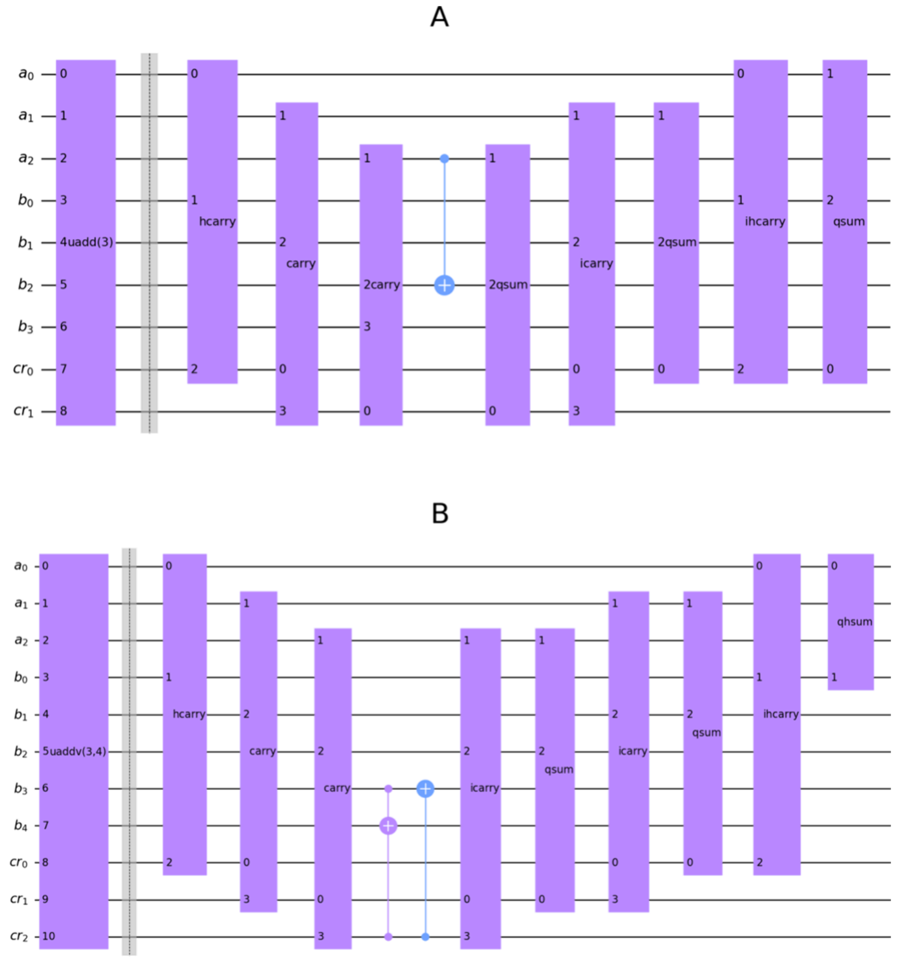


Figure A-1. Unsigned adder gates. The gate is shown on the left, and its internals are shown on the right. (A) 3-bit unsigned adder gate, uadd(3), implementing $a, b \mapsto a, a + b$, (B) 3-bit to 4-bit unsigned mismatched bit length adder, uaddv(3,4), for $a, b \mapsto a, a + b$. The no-input bit is treated as zero.

(0 1); since $sgn(\sigma) = -1$ for this case, the amplitude values are inverted. The third row is for $\tau = (1\ 2)$, the fourth is for $\tau = (0\ 2\ 1)$, and so forth. The binary-coded version S_b produced equivalent results.

6. DISCUSSION

6.1. Resource Requirements. First, let us refer back to Figure 14 in Section 3.2, which compares the number of qubits for the simulator circuit including state preparation and time-evolution circuits, with the estimates given by Kassal et al.'s method.[4] As shown, our results appear offset from Kassal's results. This is because our data include additional qubits for arithmetic calculations and antisymmetrization (see Appendix C). Kassal chose several methods to minimize the number of temporary qubits including Draper's quantum addition algorithm,[18] whereas we chose simple algorithms that require extra temporary qubits, compared to Kassal's. This results in a constant offset.

Regarding antisymmetrization, a register set described in Section 3.1.3 is needed, and the number of those required qubits scales as $O(\eta \log \eta)$ (see eq C-4). Kassal suggests applying a method for preparing antisymmetric wave functions[19] for

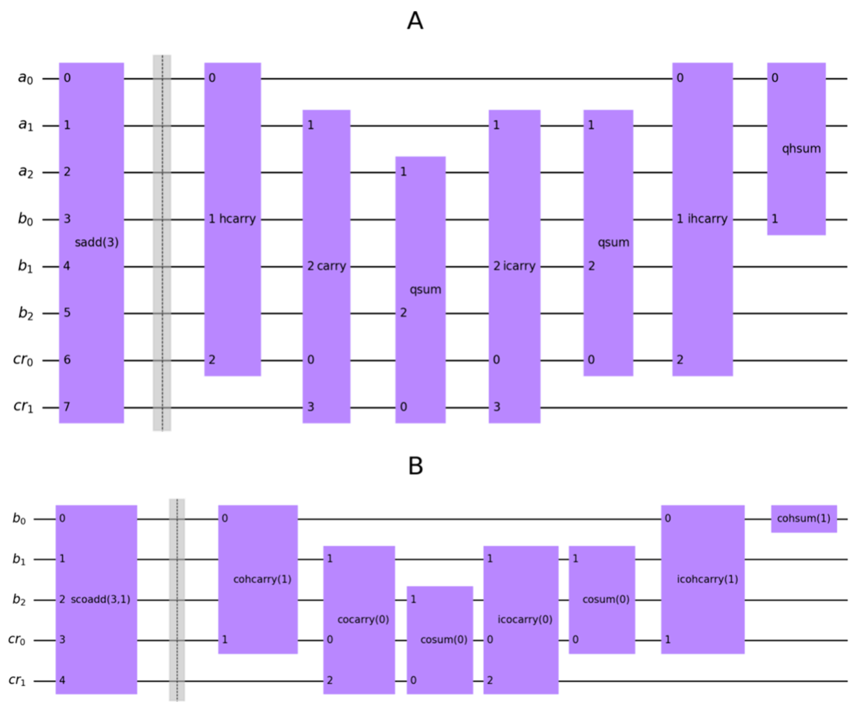


Figure A-2. (A) 3-bit signed adder gate sadd(3) for a, $b \mapsto a$, $a + b$, (B) 3-bit signed constant value adder scoadd(3,1), for $b \mapsto y + b$, where y is a constant.

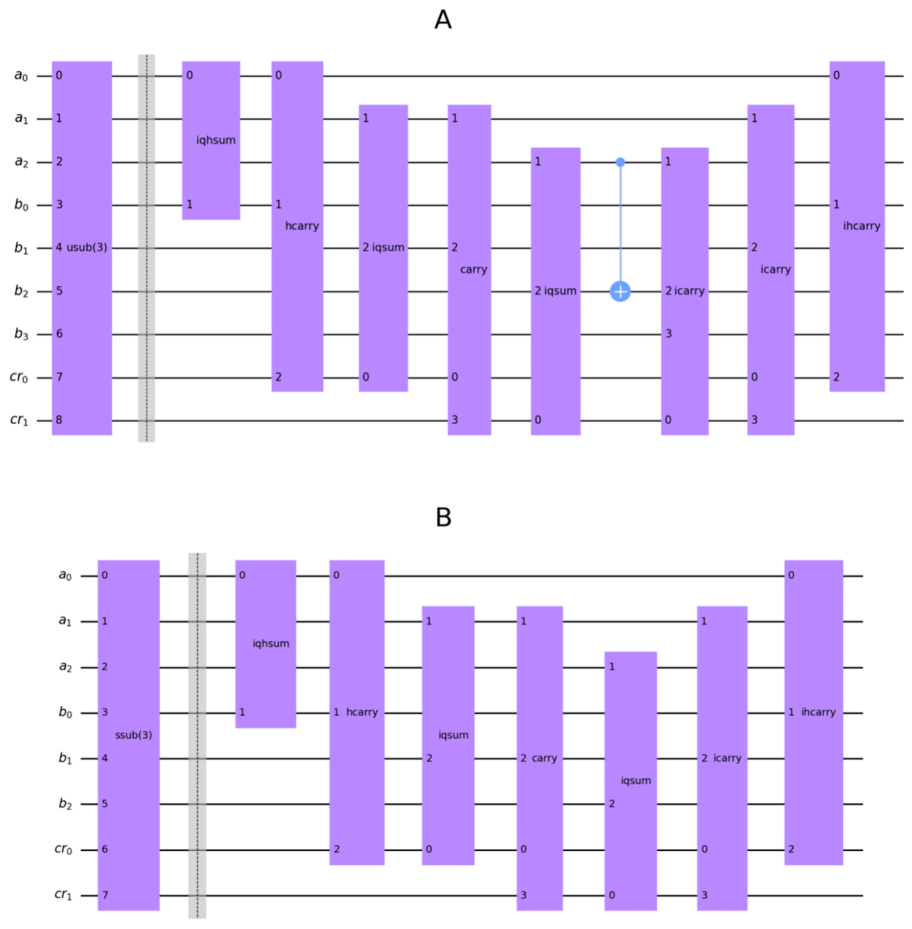


Figure A-3. Unsigned and signed subtractor gates. (A) 3-Bit unsigned subtractor gate, usub(3), for a, $b \mapsto a$, $b{-}a$, (B) 3-bit signed subtractor gate, ssub(3), for a, $b \mapsto a$, $b{-}a$.

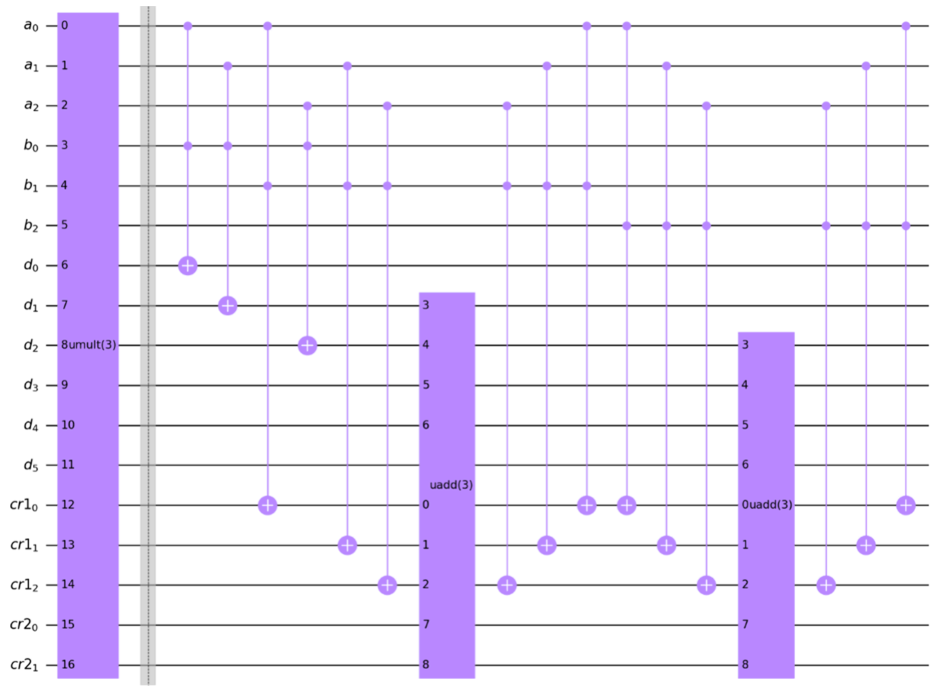


Figure A-4. 3-bit ×3-bit unsigned multiplier gate, umult(3), for a, b, d; $d = 0 \mapsto a, b, a \times b$.

multielectron wave functions, but their estimation formula does not include the number of extra qubits that would be required by such a method. This forms another offset, depending on η. Besides these differences, our data agree with Kassal's.

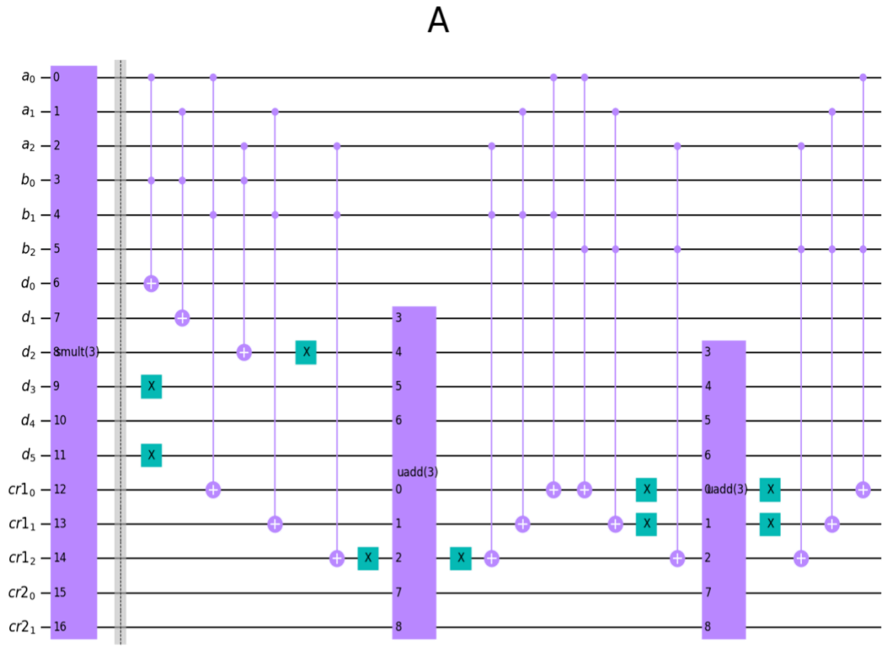
Note that grid-based first-quantization methods have been considered to require many ancilla, or temporary qubits for arithmetic operations, increasing the total number of qubits significantly, but as shown in Section 3.1.1, the number of temporary qubits is constant with the number of electrons, so this issue does not seem to be a critical one. Here, a trade-off exists between the qubit count and circuit execution time. The above results were achieved when all temporary qubits were reused. Reusing temporary qubits introduces the serialization of gate operations before and after reuse. If we give up on reuse and allocate different qubits for independent temporary qubits, gate operations on those independent qubits could be executed in parallel at the cost of extra qubits; i.e., the number of required temporary qubits will increase. Therefore, optimizations to reduce the number of temporary qubits have become more desirable.

Another type of cost for reusing temporary qubits occurs when a released qubit is reused at a distant physical location on the quantum computer device. In such a case, the qubit content must be relocated to the location of its usage by using swap operations between adjacent physical qubits along the path from the released location to the reused location. This may add significant overhead to the execution time. If a temporary qubit allocation can be performed in conjunction with a transpiler, an unused physical qubit (close to where it is needed) can be allocated as a temporary qubit. It can then be returned to an unused state after it is released so that it can be reallocated to another temporary qubit if need be. Such a feature would be helpful for quantum circuits such as this chemical simulator that require a lot of temporary qubits.

6.2. Prospects of Protein Computations. We would like to model the chemical reactions of protein molecules, which is a central concern of industrial applications such as drug and catalyst development. The plots in Figure 14 (Section 3.2) for 37 and 48 particles represent the qubits required for computations of the amino acids glycine and alanine in the form of amino acid residues as part of a protein molecule. A protein is a polymer of amino acids and the number of particles, i.e., electrons and nuclei, is proportional to the number of amino acid residues. As discussed in Appendix C, the required number of qubits, n, is at most $O(\log \eta)$. This indicates that the crsQ simulator of this work is capable of computing large molecules such as proteins with resource requirements on the order of $O(\eta \log \eta)$. (See eq C-7)

Using the formula to compute qubit counts shown in Appendix C, we can predict the required number of qubits to simulate protein molecules. Let us take insulin and cytochrome c as examples. Insulin ($C_{257}H_{383}N_{65}O_{77}S_6$) has $L_n = 788$ nuclei and $\eta = 3092$ electrons. We choose the coordinate bit-count n_1 such that the number of 3-D grids per nucleus is constant with the numbers in Appendix C ($2^{30}/nuclei$), which gives us $n_1 = \left\lceil \frac{1}{3}\log_2(2^{30} \times 788) \right\rceil = \left\lceil 10 + \frac{1}{3}\log_2 788 \right\rceil = 14$ and $n = 3n_1 = 42$. Using eq C-5, the total number of required qubits $N_{\text{tot}} < 3092(42 + \lceil \log_2 3092 \rceil) + 42 \times (788 + 4) + 9 = 200{,}241$. The second example of cytochrome c ($C_{555}H_{864}N_{146}O_{151}S_5Fe$) has $L_n = 1690$ nuclei and $\eta = 6498$ electrons. Using the same coordinate bit-count as insulin, the number of required qubits $N_{\text{tot}} < 6498(42 + \lceil \log_2 6498 \rceil) + 42 \times (1690 + 4) + 9 = 428{,}547$. When these numbers of qubits become available as fault-tolerant qubits, simulations of such proteins shall become feasible.

When we look at the overall procedure of a protein simulation, an initial wave function such as the Hartree–Fock (HF) wave function is a common requisite both in quantum- and classical computer calculations. Currently, this is performed by classical computers. This is the case for both first-quantization methods, such as the one in this work, as well as configuration interaction (CI)-based second-quantization quantum computing. However, performing an accurate wave function calculation for large molecules such as proteins is a major challenge. On classical



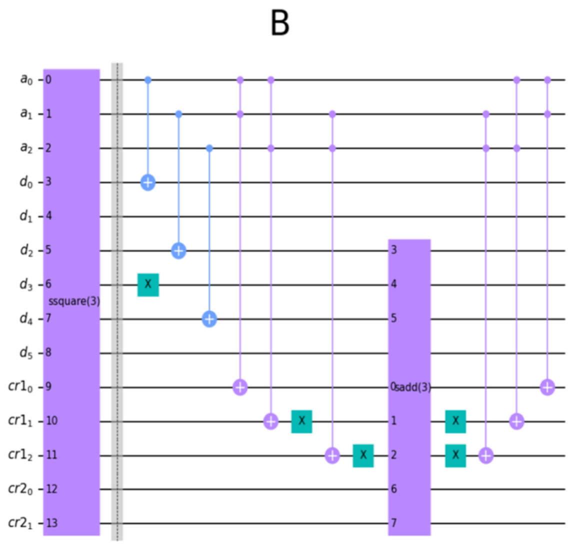


Figure A-5. Variations of the multiplier gate. (A) 3-bit ×3-bit signed multiplier gate, smult(3), for a, b, d; $d = 0 \mapsto a, b, a \times b$, (B) 3-bit signed square gate, ssquare(3) for $a \mapsto a \times a$.

computers, it can only be acquired through a successful self-consistent field (SCF) calculation. To safely guide the SCF calculation to convergence while selecting the correct orbitals to be occupied by electrons, techniques that are different from large-scale fast computation methods used on supercomputers become necessary.[20,21] Since this task is not handled efficiently by classical computers, tackling this problem on quantum computers, which is beyond the scope of this study, is an important area of research.

6.3. Effectiveness of Implementation. The automatic temporary qubit allocation by the heap component that we designed (Section 4.3) was confirmed to be effective. The simulator circuit has the structure shown in Figure 23, which illustrates more than 15 examples of gate usage. Each gate invocation is accompanied by target qubit passing, where the temporary qubit count must be determined. The ability to calculate the number of temporary qubits automatically was found valuable in general but especially so in the case of invoking the P.E. time evolution gates for electrons and nuclei. If there were no automatic mechanism, then the programmer would have to analyze the circuit and determine a formula that gives the number of qubits. In the case of P.E. time evolution gates, there are many temporary qubits for various purposes, and constructing a formula for the count is time-consuming (See Figure 5 for an example of the gate, and eq C-2 in Appendix C, for an example of a formula giving the total number of qubits). The formula would change when the algorithm is modified. Such modifications did occur frequently during this work, and if it were not for the automatic handling of the temporary qubits, we would have needed to revise the formula at each change.

The binding and invocation mechanism ensured that the caller provided the correct number of parameter qubits. There were several occasions when we decided to modify the parameters of a certain gate but failed to update some of the caller-side code accordingly. The resulting parameter mismatch was displayed on the editor screen and detected upon circuit

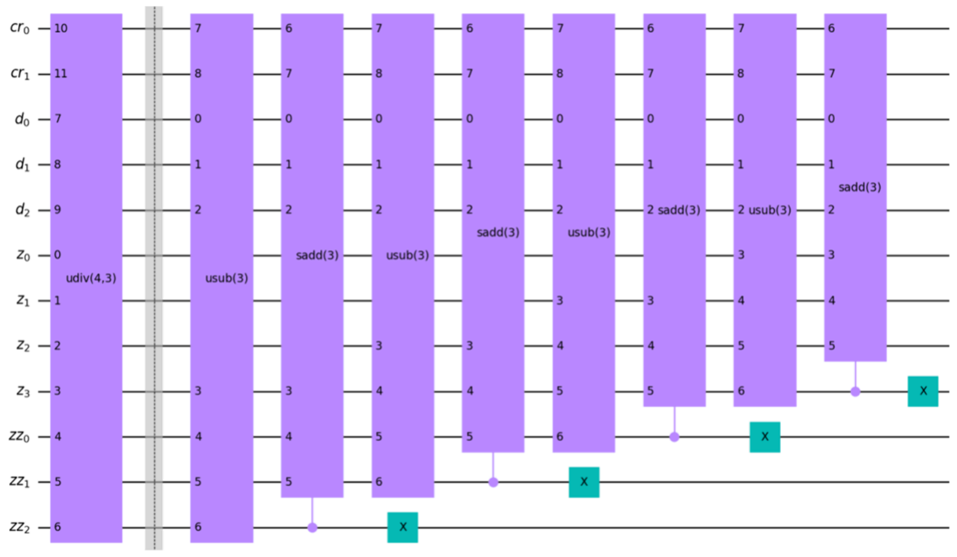


Figure A-6. 4-bit by 3-bit unsigned divider gate, udiv(4,3), for z, $zz = 0$, d; $z|zz \mapsto z \bmod d \,|\, \lfloor z/d \rfloor$, d, where | denotes bit list concatenation in little endian bit order.

generation. This allowed for quick diagnosis and correction of the mismatch.

6.4. Resource Usage during Circuit Construction. We noticed that custom gates with many internal instructions substantially increase the circuit construction and transpilation time. We have not yet investigated the cause of this time increase.

When we evaluated larger configurations of the simulator, memory limits were reached not only during circuit execution time on the quantum computer simulator, which was expected, but also during circuit construction time or transpilation time before the circuit was run. We have not yet determined whether the memory usage is within the expected level or not.

Our programs have not been optimized yet. Once we gain more insight into the above memory issues, we may be able to optimize our programs to reduce memory usage.

7. CONCLUSIONS

We have implemented and verified a quantum circuit generator program that can construct grid-based first quantization chemical simulator circuits. The generated circuits can serve as a basis for various evaluations, and the generator program can be used as a basis for experimenting with algorithm improvements. The modular design of the circuit generator program allows for the easy replacement of circuit blocks while minimizing the impact of changes to the circuit block interfaces.

The current implementation has the following limitations: (1) Some generated circuit elements could not be verified as a run on a quantum computer simulator, because the required qubit count exceeded the treatable one. For example, the P.E. time evolution gate Θ_{ep} for two- or three-dimensional models could not be run. (2) We implemented the most basic circuit, although various improvements have been proposed. (3) Evaluation of resource requirements is limited to qubit count, and we have not yet evaluated gate count or circuit depth in our software. These are potential areas for future investigation. For (2), we can implement and evaluate improved versions of the circuit elements of the simulator. Not all improvements lead to fewer qubits, but some of them do. Especially, arithmetic gates that require fewer qubits will reduce the overall qubit count, which would improve the situation of (1). For (3) we need to add a feature to our software that counts the number of elementary gates.

APPENDIX A. CIRCUITS FOR ARITHMETIC OPERATIONS

This appendix shows the arithmetic gates that appeared in the potential energy terms of the samples. The circuits in this study are based on Tomaru.[1] For other circuit design proposals, see the references of Tomaru.[1]

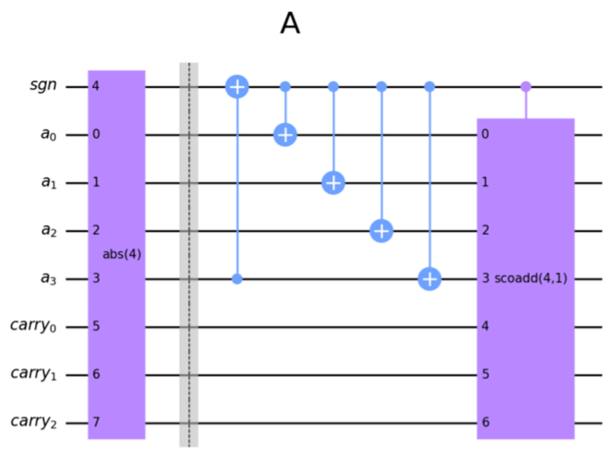
The assignment of qubits for each of the gates is summarized in Table A-1.

The first gate shown is the unsigned adder for an n-bit input, uadd(n), for n = 3 in Figure A-1 A. This gate takes two 3-bit unsigned integers, i.e., a_2, a_1, a_0 and b_2, b_1, b_0 and one carry bit b_3 as input and produces the sum as a 4-bit unsigned integer on bits b_3, ..., b_0, including the carry bit. The circuit consists of the bitwise arithmetic gates carry, hcarry, qsum, icarry, and ihcarry, which are illustrated in Figures A-8 and A-9. The bits cr_1, cr_0 are used to store internal carry bits. The bit-count for an n-bit version is $n + (n + 1) + (n - 1) = 3n$. The internal carry bits are temporary qubits. They must be initialized to $|0\rangle$ as input, and at the end of the circuit will be returned to $|0\rangle$. These temporary qubits can be used for other purposes immediately after the circuit is executed.

Figure A-1B shows a variation of the unsigned adder named uaddv(m, n) that can take input with unequal bit lengths m and n + 1. The gate for m = 3 and n = 4 is shown. This circuit is equivalent to the one constructed by substituting no-input bits with $|0\rangle$ in the bitwise gates. This circuit requires $m + 2n$ qubits, including $n - 1$ internal carry qubits.

An adder for signed integers with equal bit lengths, sadd(n), is shown in Figure A-2 A. For signed arithmetic, the carry from the MSB is always ignored so there is one less output bit (the missing b_3 bit) compared with the unsigned version uadd(n). The number of qubits is $2n + (n - 1) = 3n - 1$.

A signed adder that can be used for adding a constant value to a set of qubits, scoadd(n, y), is shown in Figure A-2 B. This is an n-bit adder that takes the value y which is specified at circuit generation time, as the second configuration parameter. This



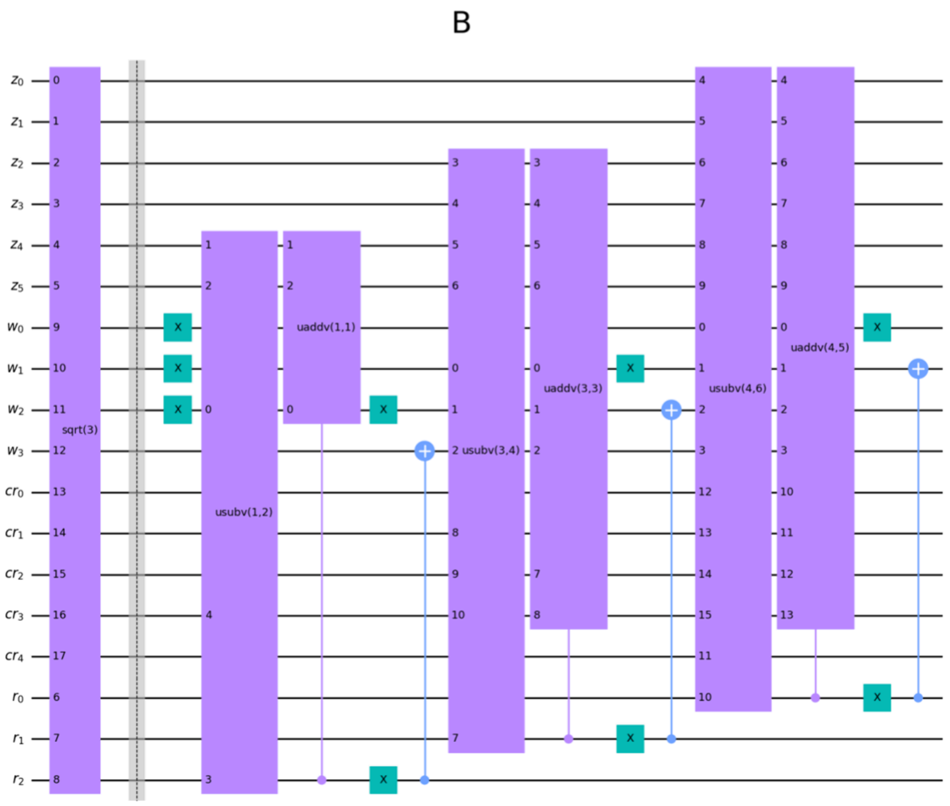


Figure A-7. Absolute value and square root gates. (A) 4-bit absolute value gate, abs(4), for $a \mapsto |a|$, (B) 3-bit output square root gate, sqrt(4), for $z,r;r = 0 \mapsto z - r^2, r;\ r = \lfloor \sqrt{z} \rfloor$.

circuit takes just one set of qubits for input and output. Compared to sadd(3), the qubits for the *a* bits are removed, and the number of qubits is $2n - 1$.

The next gate is a subtractor gate. The *n*-bit unsigned subtractor gate, usub(n), for $n = 3$ is shown in Figure A-3 A. This is made by inverting the order of the gates of the unsigned adder gate uadd(n).

The signed subtractor gate, ssub(n), for $n = 3$, shown in Figure A-3 B, is constructed from the signed adder, sadd(n), in the same way as the unsigned subtractor.

The unsigned multiplier gate, umult(n), for $n = 3$ is shown in Figure A-4. This gate uses Toffoli gates to compute the product of two bits; then it sums the intermediate results using unsigned adder gates. Unlike the adders and subtractors, the result of the multiplication is stored in a set of $2n$ qubits, d_0 through d_5, separate from the input values. $2n - 1$ temporary bits are required for both the intermediate product bits and the carry bits for the internal uadd(n) gates. All this adds up to $n + n + 2n + 2n - 1 = 6n - 1$ qubits.

The signed version of the multiplier, smult(n), for $n = 3$ is shown in Figure A-5 A. Based on the unsigned multiplier, it bit-flips several qubits to implement the semantics for signed integers (ref 1, Subsection 6.2.2). The number of required qubits is the same as umult(n) and is $6n - 1$ qubits.

An extension of the signed multiplier is the signed square gate, ssquare(n), for $n = 3$, as shown in Figure A-5 B (ref 1, Subsection 6.2.3). This gate requires $n + 2n + (2n - 1) = 5n - 1$ qubits.

The unsigned divider gate, udiv(m, n), for $m = 4$ and $n = 3$ is shown in Figure A-6. The parameters *m* and *n* are the bit count of the dividend and divisor, respectively. As output, the quotient in *m* bits, and the remainder in *n* bits are produced. The divider circuit is not a simple inversion of the multiplier circuit since the

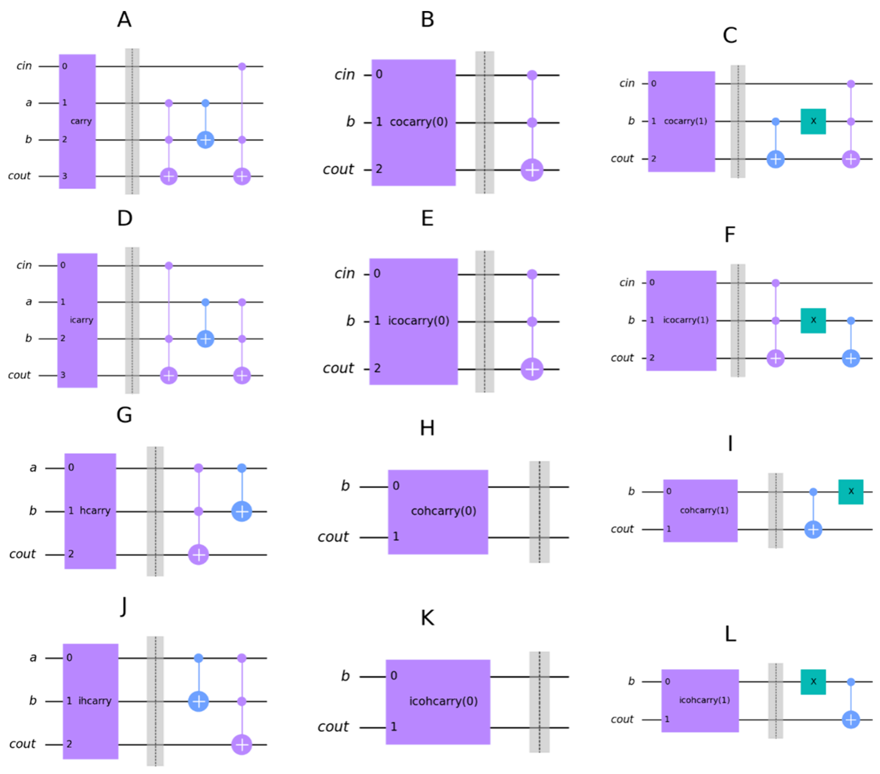


Figure A-8. Carry gate and its variations. (A) Full carry gate, carry, (B) Constant input full carry gate, cocarry(a), with $a = 0$, (C) The same gate with $a = 1$, (D) inverse full carry gate, icarry, (E) constant input inverse full carry gate, icocarry(a), with $a = 0$, (F) the same gate with $a = 1$, (G) half carry gate, hcarry, (H) constant input half carry gate, cohcarry(a), with $a = 0$, (I) same gate with $a = 1$, (J) inverse half carry gate, ihcarry, (K) constant input inverse half carry gate, icohcarry(a), gate with $a = 0$, and (L) same gate with $a = 1$.

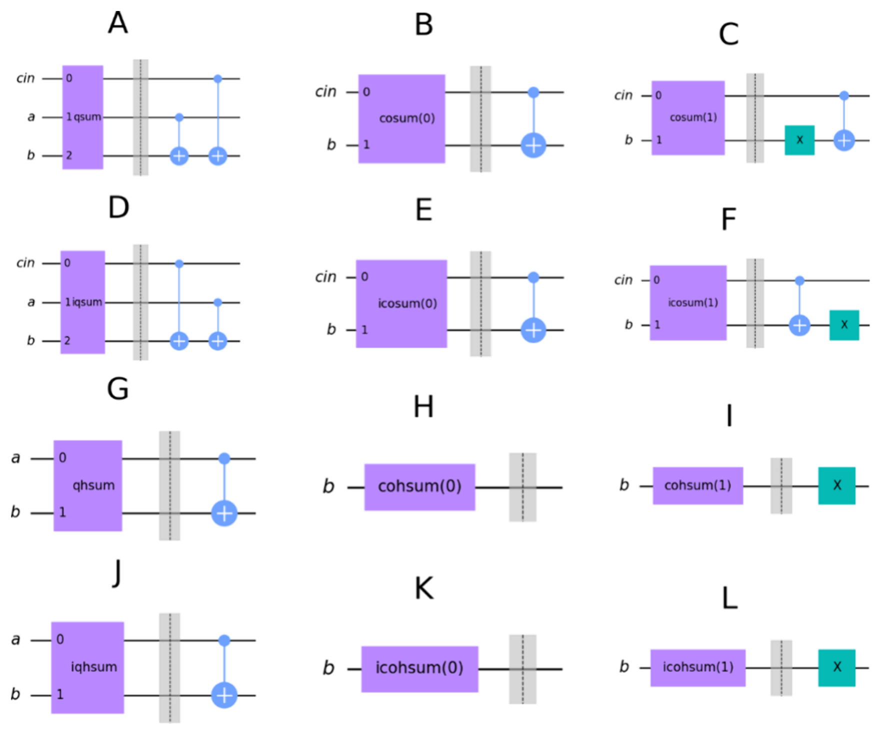


Figure A-9. Sum gate and its variations. (A) Full sum gate, qsum, (B) Constant input full sum gate, cosum(a), with $a = 0$, (C) same gate with $a = 1$, (D) Inverse full sum gate, iqsum, (E) Constant input inverse full sum gate, icosum(a), with $a = 0$, (F) The same gate with $a = 1$, (G) Half sum gate, qhsum, (H) Constant input half sum gate, cohsum(a), with $a = 0$, (I) The same gate with $a = 1$, (J) Inverse half sum gate, iqhsum, (K) Constant input inverse half sum gate, icohsum(a), with $a = 0$, (L) The same gate with $a = 1$.

input and output of the two gates are not the same. The division is processed by subtracting a multiple of the divisor from the dividend, and revoking the subtraction if an underflow is caused. The MSB of the result at this stage is in the inverse of the

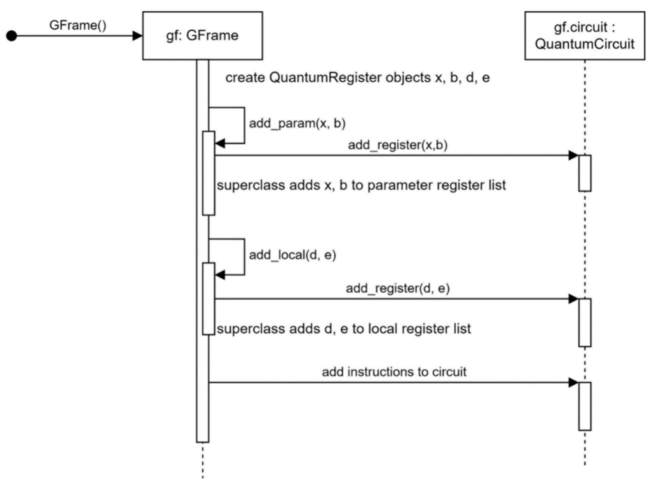

GFrame()
gf: GFrame
gf.circuit : QuantumCircuit
create QuantumRegister objects x, b, d, e
add_param(x, b)
add_register(x,b)
superclass adds x, b to parameter register list
add_local(d, e)
add_register(d, e)
superclass adds d, e to local register list
add instructions to circuit

Figure B-1. Sequence diagram of the constructor of GFrame.

quotient value, so an X gate is applied to flip the bit. The input and output of this gate are somewhat complex. For input, the dividend is stored in z_3, ..., z_0 and the divisor in d_2, ..., d_0. An additional group of ancilla bits $zz_2...zz_0$ must also be provided with values set to 0. The resulting quotient is stored in the 4 bits $zz_2...zz_0$ and z_3, which means that the higher bits are stored in the ancilla bits zz, and the lower bits are stored in the high bits of z. The remainder is stored in the low bits of z, which is $z_2...z_0$. Together, the concatenated bitstring $zz_2...zz_0 \mid z_3...z_0$ stores the quotient $q_3...q_0$ and remainder $r_2...r_0$, concatenated as $q_3...q_0 \mid r_2...r_0$. The symbol ∣ is used to represent bitstring concatenation. Since the ancilla bits store the calculation result, they cannot be reused for other purposes until they are reset to 0 by performing the reverse of the udiv(m, n) gate. In addition to this, the internal usub and sadd gates require $n - 1$ carry bits. The same $n - 1$ bits can be reused in different usub and sadd gates. As a result, the udiv(m, n) gate requires $m + n + n + (n - 1) = m + 3n - 1$ qubits.

Figure A-7 A shows the absolute value gate, abs(n), for $n = 4$. This gate conditionally negates the input qubits based on the MSB of that input. A constant-value signed adder gate, scoadd, is used internally, and 3 carry bits are required by the scoadd gate. An ancilla bit sgn is required to store the sign bit of the input. The abs(n) gate requires a total of $n + (n - 1) + 1 = 2n$ qubits.

The final gate of this series is the square root gate, sqrt(n), for $n = 3$ shown in Figure A-7 B. This gate takes $2n$ bits as input and produces a square root in n bits and the remainder in n bits. For this circuit, the input value is stored in z_5, ..., z_0, and ancilla bits w_3, ..., w_0 are used to store work register values. Moreover, temporary carry bits cr_4, ..., cr_0 are required by the internal arithmetic gates, the square root value is produced in r_2, ..., r_0, and the remainder in z_2,..., z_0. This sums up to $2n + (n + 1) + (2n - 1) + n = 6n$ qubits in total.

The arithmetic gates include bitwise gates, such as carry gates and sum gates. We also introduce half versions of these gates which treat the input carry bit of a constant $|0\rangle$. These half versions can be used for the LSB on adder and subtractor

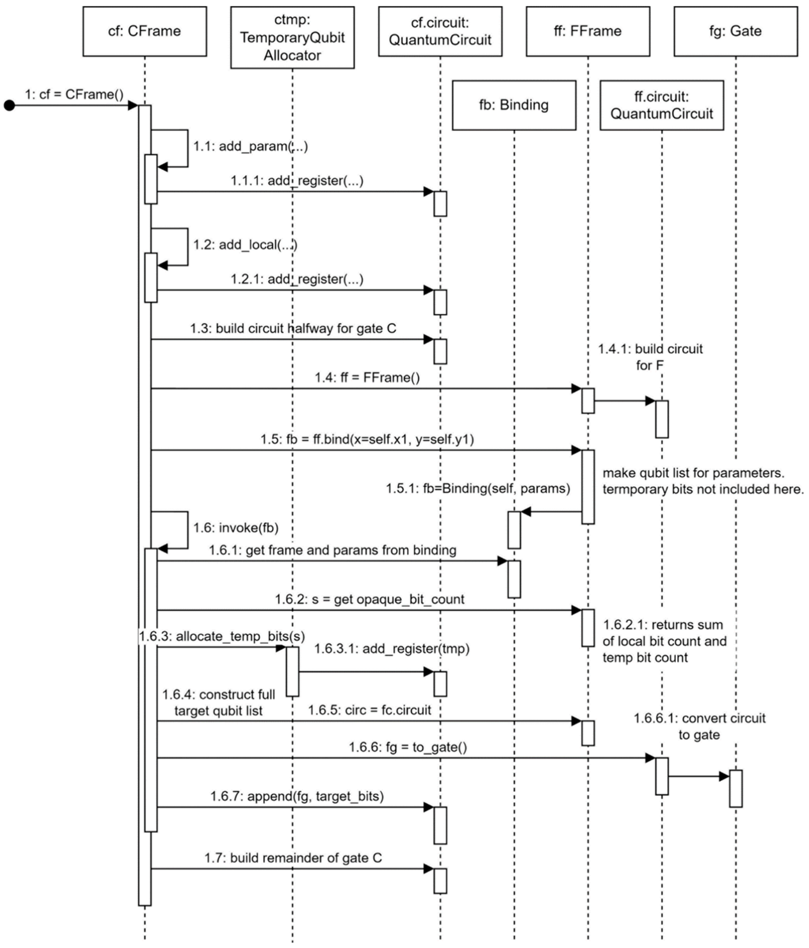

cf: CFrame
ctmp: TemporaryQubit Allocator
cf.circuit: QuantumCircuit
ff: FFrame
fg: Gate
fb: Binding
ff.circuit: QuantumCircuit
1: cf = CFrame()
1.1: add_param(...)
1.1.1: add_register(...)
1.2: add_local(...)
1.2.1: add_register(...)
1.3: build circuit halfway for gate C
1.4.1: build circuit for F
1.4: ff = FFrame()
1.5: fb = ff.bind(x=self.x1, y=self.y1)
make qubit list for parameters. termporary bits not included here.
1.5.1: fb=Binding(self, params)
1.6: invoke(fb)
1.6.1: get frame and params from binding
1.6.2: s = get opaque_bit_count
1.6.2.1: returns sum of local bit count and temp bit count
1.6.3: allocate_temp_bits(s)
1.6.3.1: add_register(tmp)
1.6.4: construct full target qubit list
1.6.5: circ = fc.circuit
1.6.6.1: convert circuit to gate
1.6.6: fg = to_gate()
1.6.7: append(fg, target_bits)
1.7: build remainder of gate C

Figure B-2. Sequence diagram of the constructor of CFrame.

Table C-1. Decomposition of Qubit Counts

model	η	L_n	N_{tot}	N'_{tot}	N_{Kassal}	N_Ψ	N'_Ψ	N_σ	N_H
H	1	1	188	158	40	60	30	0	128
He	2	1	220	190	70	90	60	2	128
Li	3	1	252	222	100	120	90	4	128
$H + H_2$	3	3	312	222	160	180	90	4	128
O	8	1	416	386	250	270	240	18	128
Gly	30	7	1358	1148	1090	1110	900	120	128
Ala	38	10	1734	1434	1420	1440	1140	166	128

circuits. The carry gate and its variants are shown in Figure A-8. Variants are defined for the combination of full- or half-carry, forward or inverse, whether input a is a qubit, a constant 0, or a constant 1. The sum gate, qsum, and its variants are shown in Figure A-9. Variants of the sum gate are defined in the same manner as the carry gates.

APPENDIX B. SEQUENCE DIAGRAM OF THE FRAME CLASSES

In this section, we show sequence diagrams that explains how the heap framework classes are used to create Qiskit custom gate objects.

The first diagram shown in Figure B-1 is the constructor method of GFrame. This method creates a QuantumCircuit object labeled gf.circuit that holds the internal instructions (including gates) that comprise the gate G, as shown in Figure 16. After an empty QuantumCircuit is created, QuantumRegisters must be added to the circuit before instructions are added to the circuit. The diagram shows that the GFrame constructor creates QuantumRegister objects and registers them in its superclass Frame through the methods add_param and add_local specifying whether the QuantumRegister is used as a parameter or a local register. The superclass maintains a separate list for each of the two categories of registers, and at the same time adds those registers to the QuantumCircuit object, gf.circuit. Internal instructions are added to the circuit at the final step in the diagram.

The second sequence diagram, Figure B-2, shows the constructor method of CFrame. Similar to the previous case with GFrame, the constructor of CFrame creates a QuantumCircuit that holds the internal instructions of gate C. The CFrame constructor is more complicated than that of FFrame because it calls an inner custom gate, F. Gate F is created by constructing an FFrame and then converting it to a Qiskit custom gate. In Figure B-2, each method invocation arrow has been numbered hierarchically for reference. Step 1 is the constructor call. When this call is complete, the circuit for C will be constructed on the QuantumCircuit object labeled cf.circuit. Steps 1.1 and 1.2 comprise the creation of QuantumRegisters and the purpose is the same as the case of GFrame. After the QuantumRegisters are prepared, the instruction generation begins. Step 1.3 adds instructions to the cf.circuit, up to the point where gate F needs to be called. Gate F will be created from the FFrame object later on at step 1.6.6.1. Step 1.4 calls the FFrame constructor. This call completes all the circuit generations in FFrame on the QuantumCircuit labeled ff.circuit, so the number of temporary qubits that FFrame requires is determined at this point. In step 1.5, the caller CFrame calls bind on the callee FFrame passing arguments $x1$ and $y1$. The arguments are passed using Python's keyword argument syntax to avoid the risk of argument order mismatch. In step 1.5.1, FFrame makes a list of received QuantumRegisters in the order that the quantum circuit of FFrame expects. This list does not include temporary qubits yet. The list is stored in a binding object and sent back to the caller. Step 1.6 is the invoke part. "invoke" is a method implemented by the superclass of CFrame. In step 1.6.1, invoke extracts the frame and the list of registers that were created at the bind call. In step 1.6.2, invoke queries FFrame for the number of temporary qubit registers. In step 1.6.3, invoke requests for the number of temporary qubits from the TemporaryQubitAllocator object that is associated with CFrame. The allocator tries to satisfy the request by using pooled temporary qubits, but when no stock is left, it will add new registers with names such as "tmp1" to cf.circuit, as shown in step 1.6.3.1. In step 1.6.4, the temporary qubit list is ready and is appended to the list of parameters. All the QuantumRegisters in the list are converted to a list of Qubits that comprise those QuantumRegisters. This forms the complete target qubit list that is required to invoke gate F. In step 1.6.6, the ff.circuit is converted to a custom gate F, labeled fg. Step 1.6.7 gives fg, along with the target qubit list, to the append method of cf.circuit. The FFrame has been invoked. Step 1.7 adds the remainder of instructions after invoking the cf.circuit.

APPENDIX C. DETAILS OF QUBIT COUNT EVALUATION

Table C-1 shows the decomposition of qubit counts for the plot shown in Figure 14. The circuits were generated with parameters set as $d = 3$, $n_1 = 10$, and $n = dn_1 = 30$, where the binary-coded version of the antisymmetrization gates was used.

The meaning of the columns are as follows; η: number of electrons, L_n number of nuclei, $N_{tot} = N_\Psi + N_\sigma + N_H$: number of total qubits counted on the generated circuit without the BOA, $N'_{tot} = N'_\Psi + N_\sigma + N_H$: the same but with the BOA, N_{Kassal}: estimations according to Kassal,[4] $N_\Psi = dn_1(\eta + L_n)$: number of qubits for the wave function without BOA, $N'_\Psi = dn_1\eta$: number of qubits for the wave function with the BOA, N_σ: number of ancillary qubits required for antisymmetrization of the wave function using binary-coded gates, $N_H = (2d + 6)n_1 + 8$: number of temporary qubits required for the Hamiltonian simulation (Section 3.1.2).

N_σ is the number of ancillary qubits required by the binary-coded permutation gate and is defined as

$$N_\sigma(\eta) = \begin{cases} 1, & \eta = 2 \\ 1 + \sum_{k=2}^{\eta} \lceil \log_2 k \rceil, & \eta \geq 3 \end{cases} \tag{C-1}$$

Here, the "1" before the sum means the single ancilla bit used in swapping. In the following, we will consider the case for $\eta \geq 3$. By applying (eq C-1) to N_{tot} and N'_{tot}, we obtain the following:

$$N_{\text{tot}} = N_\Psi + N_\sigma + N_{\text{H}} = n(\eta + L_n) + \sum_{k=2}^{\eta} \lceil \log_2 k \rceil + 12n_1 + 9 \tag{C-2}$$

$$N'_{\text{tot}} = N'_\Psi + N_\sigma + N_{\text{H}} = n\eta + \sum_{k=2}^{\eta} \lceil \log_2 k \rceil + 12n_1 + 9 \tag{C-3}$$

As an approximation for the sum, we can use the following:

$$\begin{aligned} \sum_{k=2}^{\eta} \lceil \log_2 k \rceil &< \sum_{k=2}^{\eta} \lceil \log_2 \eta \rceil \\ &= (\eta - 1)\lceil \log_2 \eta \rceil \\ &< \eta \lceil \log_2 \eta \rceil \end{aligned} \tag{C-4}$$

Applying eqs C-4 to C-2 and C-3 gives

$$\begin{aligned} N_{\text{tot}} &< n(\eta + L_n) + \eta \lceil \log_2 \eta \rceil + 12n_1 + 9 \\ &= \eta(n + \lceil \log_2 \eta \rceil) + n(L_n + 4) + 9 \end{aligned} \tag{C-5}$$

$$\begin{aligned} N'_{\text{tot}} &< n\eta + \eta \lceil \log_2 \eta \rceil + 12n_1 + 9 \\ &= \eta(n + \lceil \log_2 \eta \rceil) + 4n + 9 \end{aligned} \tag{C-6}$$

When the spatial size of the molecule is taken into consideration, the grids should be increased in accordance with the molecule volume. Here, the appropriate n will likely depend on the 3-D structure of the molecule as well as the intent of the simulation; therefore, n is not a simple function of L_n. Although generally describing $n(L_n)$ is not easy, grid size 2^n that is appropriate for describing an η electron system is estimated to be at most $O(\eta)$, i.e., $n = O(\log \eta)$. Then, because $L_n \le \eta$, eqs C-5 and C-6 are simplified to

$$N_{\text{tot}} = O(\eta \log \eta) \tag{C-7}$$

$$N'_{\text{tot}} = O(\eta \log \eta) \tag{C-8}$$

ASSOCIATED CONTENT

Data Availability Statement

The Python software csrQ discussed here is distributed as free software at https://github.com/crsq-dev/. The scripts used to generate the results and figures are hosted at https://github.com/crsq-dev/crsq-papers.

AUTHOR INFORMATION

Corresponding Authors

Hideo Takahashi − *Department of Mechanical Engineering, School of Engineering, The University of Tokyo, Bunkyo-ku, Tokyo 113-8656, Japan;* orcid.org/0009-0004-8948-9451; Email: takahashi-hideo543@g.ecc.u-tokyo.ac.jp

Fumitoshi Sato − *Institute of Industrial Science, The University of Tokyo, Meguro-ku, Tokyo 153-8505, Japan;* Email: satofumi@iis.u-tokyo.ac.jp

Authors

Tatsuya Tomaru − *Center for Exploratory Research, Research and Development Group, Hitachi Ltd., Kokubunji, Tokyo 185-8601, Japan;* orcid.org/0000-0002-7645-8240

Toshiyuki Hirano − *Institute of Industrial Science, The University of Tokyo, Meguro-ku, Tokyo 153-8505, Japan*

Saisei Tahara − *Institute of Industrial Science, The University of Tokyo, Meguro-ku, Tokyo 153-8505, Japan*

Complete contact information is available at: https://pubs.acs.org/10.1021/acs.jctc.4c00708

Author Contributions

H.T. done the programming and preparation of the manuscript. T.T. designed the quantum circuits. All authors were engaged in discussions.

Funding

This research is partially supported by UTokyo Quantum Initiative.

Notes

The authors declare no competing financial interest.

REFERENCES

(1) Tomaru, T. To be submitted.
(2) Benenti, G.; Strini, G. *Am. J. Phys.* **2008**, *7*, 657−662.
(3) Zalka, C. *Proc. R. Soc. A: Math. Phys. Eng.* **1998**, *1969*, 313−322.
(4) Kassal, I.; Jordan, S. P.; Love, P. J.; Mohseni, M.; Aspuru-Guzik, A. *Proc. Natl. Acad. Sci. U. S. A.* **2008**, *48*, 18681−18686.
(5) Chaganti, S. K.; Kesari, A. S.; Chowdhury, C. *Chem. Phys.* **2024**, *580*, No. 112195.
(6) Chan, H. H. S.; Meister, R.; Jones, T.; Tew, D. P.; Benjamin, S. C. *Sci. Adv.* **2023**, *9*, No. eabo7484.
(7) Javadi-Abhari, A.; Treinish, M.; Krsulich, K.; Wood, C. J.; Lishman, J.; Gacon, J.; Martiel, S.; Nation, P.; Bishop, L. S.; Cross, A. W.; Johnson, B. R.; Gambetta, J. M. *arxiv.org/abs/2405.08810* 2024.
(8) Van Rossum, G.; Drake, Jr, F. L. *Python reference manual;* Centrum voor Wiskunde en Informatica: Amsterdam, 1995.
(9) Gamma, J.; Helm, E.; Johnson, R.; Vlissides, R. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional: Boston, 1994.
(10) Appel, A. W. *Modern Compiler Implementation in Java*; Cambridge University Press: New York, 1998.
(11) Krekel, H.; Oliveira, B.; Pfannschmidt, R.; Bruynooghe, F.; Laugher, B.; Bruhin, F. *pytest*. https://github.com/pytest-dev/pytest (accessed Aug. 24, 2024).
(12) Vedral, V.; Barenco, A.; Ekert, A. *Phys. Rev. A* **1996**, *54* (1), 147−153.
(13) Bryant, R. E.; O'Hallaron, D. R. *Computer Systems: A Programmer's Perspective*; Pearson: Boston, 2015; 274−290.
(14) Josuttis, N. *The C++ Standard Library: A Tutorial and Reference*, 2nd Ed.; Addison-Wesley Professional: Boston, 2012; 486−497.
(15) Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*; Addison-Wesley Professional, 2003.
(16) Kivlichan, I. D.; Wiebe, N.; Babbush, R.; Aspuru-Guzik, A. *J. Phys. A: Math. Theor.* **2017**, *50* (30), 305301.
(17) Babbush, R.; Berry, D. W.; McClean, J. R.; Neven, H. *npj Quantum Inf.* **2019**, *5* (1), 92.
(18) Draper, T. G. *arXiv preprint quant-ph/0008033* 2000.
(19) Abrams, D. S.; Lloyd, S. *Phys. Rev. Lett.* **1997**, *79* (13), 2586.
(20) Kashiwagi, H.; Iwai, H.; Tokeida, K.; Era, M.; Sumita, T.; Yoshihiro, T.; Sato, F. *Mol. Phys.* **2003**, *101* (1−2), 81−86.
(21) Nishino-Uemura, N.; Hirano, T.; Sato, F. *J. Chem. Phys.* **2007**, *127* (18), 184106.