Piquasso: A Photonic Quantum Computer Simulation Software Platform

Zoltán Kolarovszki[1,2], Tomasz Rybotycki[3], Péter Rakyta[4], Ágoston Kaposi[1,2], Boldizsár Poór[2,5], Szabolcs Jóczik[1,2,6], Dániel T. R. Nagy[1,4], Henrik Varga[2], Kareem H. El-Safty[1,7], Gregory Morse[2], Michał Oszmaniec[3], Tamás Kozsik[2], and Zoltán Zimborás[1,2,9]

[1]Quantum Computing and Quantum Information Research Group, HUN-REN Wigner Research Centre for Physics, Konkoly–Thege Miklós út 29-33, H-1525 Budapest, Hungary
[2]Department of Programming Languages and Compilers, Eötvös Loránd University, Pázmány Péter sétány 1/a, H-1117 Budapest, Hungary
[3]Center for Theoretical Physics, Polish Academy of Sciences, Al. Lotników 32/46, 02-668 Warszawa, Poland
[4]Department of Physics of Complex Systems, Eötvös Loránd University, Pázmány Péter sétány 1/a, H-1117 Budapest, Hungary
[5]Quantinuum, 17 Beaumont Street, Oxford, OX1 2NA, United Kingdom
[6]Robert Bosch Kft., Gyömrői út 104., H-1103 Budapest, Hungary
[7]Department of Computer Engineering, Technical University of Munich, Arcisstraße 21, 80333 München, Germany
[8]NASK National Research Institute, Kolska 12, 01-045 Warsaw, Poland
[9]Algorithmiq Ltd, Kanavakatu 3C 00160 Helsinki, Finland

29th February, 2024

We introduce the Piquasso quantum programming framework, a full-stack open-source software platform for the simulation and programming of photonic quantum computers. Piquasso can be programmed via a high-level Python programming interface enabling users to perform efficient quantum computing with discrete and continuous variables. Via optional high-performance C++ backends, Piquasso provides state-of-the-art performance in the simulation of photonic quantum computers. The Piquasso framework is supported by an intuitive web-based graphical user interface where the users can design quantum circuits, run computations, and visualize the results.

1 Introduction

In the last decade, there has been great progress in creating quantum computer prototypes. Among the many proposals, the relevance of photonic quantum computers increased due to recent demonstrations of possible photonic quantum advantage schemes [1, 2, 3] and the development of feasible fault-tolerant quantum computation methods [4, 5, 6].

In parallel with the progress on quantum hardware prototypes, the need for quantum computer simulators, and generally quantum software, has been steadily increasing. There are manifold reasons why classical simulators are needed: Current quantum devices are still noisy, so it is instructive to compare experimental results with the ideal noiseless outcomes obtained from the simulator. Moreover, one can study with noiseless simulators the performance of new heuristic algorithms, e.g., quantum neural networks or variational quantum eigensolvers. In addition to this, by implementing flexible noise models in the simulator, one could test the noise tolerance of quantum algorithms and evaluate the usefulness of different error mitigation or even error correction schemes.

Consequently, in recent years plenty of quantum computing simulation platforms have been developed [7]. However, most of these focus on qubit-based quantum computing. Much less development has been done for photonic quantum computation. Strawberry Fields [8], developed by Xanadu, is an open-source quantum programming platform built using Python [9], which contains a simulator and can also serve as an interface for existing hardware, e.g., the Borealis

Zoltán Kolarovszki: kolarovszki.zoltan@wigner.hun-ren.hu
Zoltán Zimborás: zimboras.zoltan@wigner.hun-ren.hu

chip [3]. We should also mention here Perceval [10], Bosonic Qiskit [11], GraphiQ [12], MrMustard [13] and BosonSampling.jl [14]. Perceval developed by Quandela is also a framework for simulating optical elements, however, it does not aim to treat continuous-variable models of photonic quantum computation. Bosonic Qiskit is also capable of simulating optical elements, however, it is primarily aimed at modeling hybrid quantum computation containing both bosonic and qubit-based objects; and is less emphasized for photonic systems, it naturally lacks some gates specific for photonic quantum computation such as the Kerr and Cross-Kerr gates. GraphiQ is a library dedicated to the simulation of photonic graph states, also selecting a different application domain. Finally, MrMustard is dedicated for differentiable simulation of Gaussian circuits using the Bargmann representation of Gaussian states, which is not yet implemented in Piquasso. In Table 1, we summerize the features of the relevant software packages admitting similar applications as Piquasso is designed for.

Considering the narrow selection of photonic quantum computer simulators, we developed a new photonic quantum computer simulator software called Piquasso (PhotonIc QUAntum computer Simulator SOftware). Creating a new framework is beneficial for several reasons, for example: (i) Rethinking existing classical simulations might help in the development of more efficient classical algorithms (see our torontonian implementation [15] that was also taken over by Strawberry Fields). (ii) New design choices can be implemented that enable practical features different from the existing ones. In our case, these are, e.g., the repeatability of the simulations via seeding, the increase in the simulable number of modes by the choice of Fock space truncation, and the possibility of replacing or extending default calculations via plugins (e.g., see Sec. 3.3). (iii) It can be useful to have multiple simulators testing photonic hardware. (iv) One can enable different numerical computing frameworks, e.g., TensorFlow [16] or JAX [17]. Piquasso has found many applications recently, such as simulation of non-deterministic gates [18], continuous-variable Born-machines [19] and quantum machine learning [20].

The article is structured as follows: Sec. 2 shortly summarizes the goals and main features of Piquasso, while Sec. 3 describes the hierarchy of the Piquasso platform and gives several code examples. Sec. 4 briefly discusses the computational performance of Piquasso. The web user interface of Piquasso is presented in Sec. 5. Finally, Appendix A summarizes the primary concepts and basic calculations in photonic quantum computing.

2 Goals

Our main aim with Piquasso is to enhance research on quantum optical computation by providing an accessible platform for the simulation of photonic quantum computers. On the one hand, we intend to provide a user-friendly and easy-to-extend system. On the other hand, we also focus on boosting the efficiency of the computations executed during the simulation. With the development of Piquasso, we seek to fulfill the following criteria:

- **Speed**: In the PiquassoBoost extension, calculations of several classically computationally hard quantities are rewritten in C/C++ in order to enable more efficient calculations.
- **Extendability**: Piquasso comes with a public interface to enable users to customize instructions and calculations or even to create new simulators and quantum state datatypes.
- **Repeatability**: For non-deterministic computations, one can extract the seed of the random number generation, save it, and repeat the same simulation later.
- **Intuitive user interface**: In the user interface, one can specify high-level instructions like an interferometer, a general Gaussian gate, or even a graph embedding.
- **Quantum Machine Learning support**: TensorFlow is an end-to-end machine learning platform [16], which supports automatic differentiation. The simulation of pure Fock states can be performed using TensorFlow as a calculation backend. This way, one can compute the gradient of certain simulations and use it for machine learning purposes. Moreover, Piquasso also supports the JAX machine learning framework for performing backend calculations.

3 Piquasso Platform

Piquasso is an open source software, designed to provide a simple, accessible interface, which enables users to extend the built-in simulators or define new ones. It is written using Python [9], a language already familiar with scientific computations. The source code is made available at [21], and the documentation is published at [22].

Using `pip`, Piquasso can simply be installed from PyPI using

```
pip install piquasso
```

As a dependency, the NumPy scientific computing package is installed [23], which is also a helpful companion for writing programs for scientific computing purposes. It is important to note, that using Piquasso only assumes basic knowledge of Python, as illustrated by the Code snippet 1.

3.1 Hierarchy

The schematic dependence between Piquasso objects is illustrated in Fig. 1. Piquasso is structured in a way that a single `Program` instance only contains `Preparation`, `Gate`, `Measurement` and `Channel` instances. Strictly speaking, no other information is needed in the program definition. To specify the instruction, the following syntaxes could be followed:

```
# Single instruction on RHS
pq.Q([MODES]) | [INSTR]([PARAMS])

# Single instruction on LHS
[INSTR]([PARAMS]) | pq.Q([MODES])

# Multiple instructions on RHS
pq.Q([MODES]) | (
    [INSTR_1]([PARAMS])
    | ...
    | [INSTR_N]([PARAMS])
)
```

where

- `[MODES]` is a sequence of non-negative integers representing the modes on which the instruction should act. As a syntactic sugar, it is permitted to write `all` to specify all modes if applicable, or to leave it out entirely;
- `[INSTR]`, `[INSTR_1]`, `...`, `[INSTR_M]` are instruction classes, by which preparations, gates, channels, or measurements can be defined;
- `[PARAMS]` are the parameters specific for each instruction.

The `Simulator` instance contains the `Config` instance and the `State` instance. `Config` contains the necessary data to perform simulation (e.g., Planck constant, Fock space cutoff), and the `State` instance holds the representation of the quantum state described in Sec. A.3.

After the simulation is executed, the `result` object contains all the samples under `result.samples` and the resulting quantum state under `result.state`.

3.2 Built-in simulators

Piquasso allows the simulation of special scenarios, such as computation using solely Gaussian states or pure Fock states, since this yields benefits in execution time and memory usage. For example, considering the case when only pure Gaussian states and gates are used in the photonic circuit, one might consider using the `GaussianSimulator`, which is more efficient than the `PureFockSimulator` for several use cases.

3.2.1 Fock simulators

Piquasso supports a separate simulator dedicated to working with quantum states in the Fock representation. In this simulator, the states are represented in the occupation number basis (see Appendix A). When no mixed states are required during the simulation, it is sufficient to use the `PureFockSimulator` class instead of the more general `FockSimulator`. An example usage of the `PureFockSimulator` class is shown in Code snippet 2.

Naturally, Fock simulators have a high memory usage due to storing the state vector or density matrix. The Fock space (see, Eq. (A.8) in Appendix A) has to be truncated with a cutoff particle number c determined by the user, i.e., the simulator calculates in the space defined by

$$\mathcal{F}_B^c(\mathbb{C}^d) = \bigoplus_{\lambda=0}^{c-1} \left(\mathbb{C}^d\right)^{\vee\lambda}. \tag{1}$$

The cutoff c sets a limit on the total particle number in the system, which is more memory efficient than the limit on the particle number on each mode, as will be discussed in detail in Sec. 4.3.

	Gaussian-state based simulations	Fock-space based simulations	Efficient BS	Efficient GBS	PQC with differentiabilty
Strawberry Fields	✓	✓	✗	✓	✓
MrMustard	✓	✗	✗	✗	✓
Perceval	✗	✓	✓	✗	✗
Piquasso	✓	✓	✓	✓	✓

Table 1: Comparison of different photonic quantum computer simulation frameworks. Notation: BS – Boson Sampling, GBS – Gaussian Boson Sampling, PQC – Parametric Quantum Circuits.

```
1  import numpy as np
2  import piquasso as pq
3
4  # Beginning of program definition
5  with pq.Program() as program:
6      # Initializing vacuum state
7      pq.Q(all)  | pq.Vacuum()
8
9      # Quantum Gates
10     pq.Q(0)    | pq.Displacement(r=1.0)
11     pq.Q(1)    | pq.Displacement(r=1.0)
12     pq.Q(0)    | pq.Squeezing(r=0.1, phi=np.pi / 3)
13     pq.Q(0, 1) | pq.Beamsplitter(theta=np.pi / 3, phi = np.pi / 4)
14
15     # Measurement
16     pq.Q(0, 1) | pq.ParticleNumberMeasurement()
17
18 # Choosing a simulator
19 simulator = pq.GaussianSimulator(d=2)
20
21 # Execution
22 result = simulator.execute(program, shots=1000)
```

Code snippet 1: Basic Piquasso code example. On line 5, the program definition starts with a `with` statement that will result in a `Program` instance. In this block, all the instructions should be specified. The `pq.Q` class is used to specify the modes on which the instructions on the other side of the `|` or `__or__` operator are supposed to act. After the program definition, a `GaussianSimulator` instance is created with two modes, and the result is acquired by executing `program` with `simulator`. The execution parameter `shots=1000` is specified in order to acquire 1000 samples from the measurement.

3.2.2 Pure Fock simulator with TensorFlow support

A substantial part of Quantum Machine Learning algorithms are developed for photonic architectures. Consequently, we gave special attention to enabling the differentiation of photonic circuits.

Piquasso uses the automatic differentiation capabilities of TensorFlow [16]. When simulating a photonic circuit, the Jacobian of the resulting state vector after applying the layers consisting of optical gates can automatically be differentiated. However, in TensorFlow Eager mode, custom gradients can be implemented to increase performance instead of using the automatic differentiation of built-in TensorFlow functions. Notably, the Jacobians of gate matrices corresponding to linear optical gates can be calculated [24]. Moreover, one can also differentiate the operation of applying a gate to a state vector, which is implemented in Piquasso.

In Piquasso, automatic differentiation is implemented through `TensorflowConnector`, where the simulation is restricted to pure Fock states. An example usage can be seen in Code snippet 5 implementing a CVQNN layer for a single mode. For multiple modes, the circuit definition is similar but less concise.

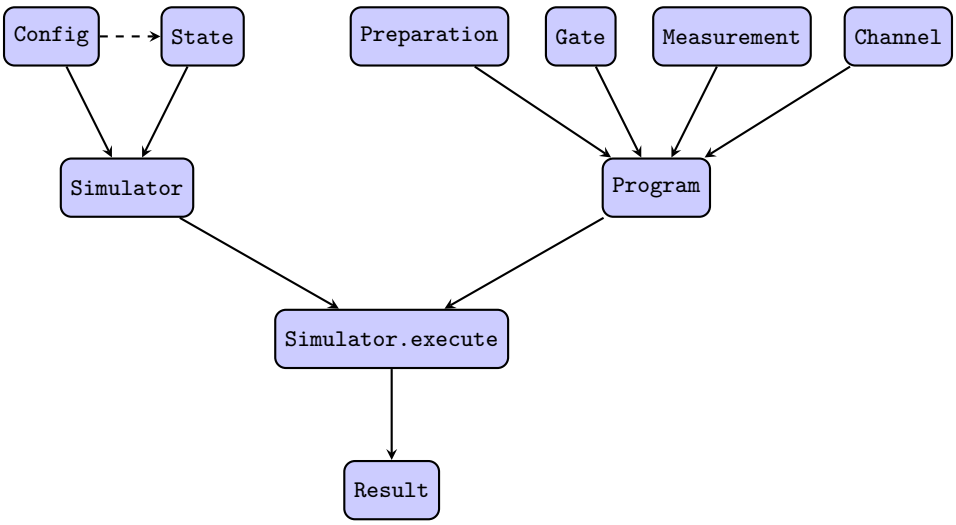


Figure 1: Schematic diagram of executing a Piquasso program. The `Config` and (optional) initial `State` object should be specified to the `Simulator`. The dotted line between `Config` and `State` indicates that `State` also depends on the `Config` for calculations after execution, but it is generally injected into `State` by the `Simulator`. The `Instruction` instances should be specified for the single `Program` instance. Then the `Program` instance should be specified to `Simulator.execute` to perform the calculations and yield a `Result` instance.

3.2.3 Gaussian simulator

The Gaussian simulator represents Gaussian quantum states in the phase space formalism, i.e., the states are stored via their mean vectors and their covariance matrices according to Eq. (A.31) in Appendix A. An example usage of the `GaussianSimulator` class is shown in Code snippet 3.

The state inside the Gaussian simulator is evolved by

$$\begin{aligned} \mu &\mapsto S\mu, \\ \sigma &\mapsto S\sigma S^T, \end{aligned} \tag{2}$$

where S is the symplectic representation corresponding to the unitary evolution and relates to $S_{(c)}$ from Eq. (A.23) in Appendix A.

The supported gates are the ones with only up to quadratic terms in their Hamiltonians. Note that while non-Gaussian measurements execute the sampling during Gaussian simulations, the final state after the measurement is not calculated since it is not Gaussian in general.

The Gaussian simulator is generally faster and more precise than performing the same simulation with the Fock simulator from Sec. 3.2.1. It also has a lower memory usage, since the mean vector and covariance matrix in Eq. (A.31) from Appendix A requires considerably less memory than a density matrix over the Fock space.

When executing Gaussian measurements, the Gaussian simulator uses high-complexity matrix functions to calculate the probabilities corresponding to different measurement outcomes, as described in Appendix A.4.2 and Appendix A.4.3. The efficient calculation of these functions is crucial for many applications, hence, Piquasso implements cutting-edge algorithms. The speedup of Piquasso (version 5.0.0) over the TheWalrus (version 0.21.0) is presented in Sec. 4.

3.2.4 Boson Sampling simulator

Boson Sampling (BS) is a well-known linear optical quantum computing protocol introduced by Aaronson and Arkhipov [25], which consists of sampling from the probability distribution of identical bosons scattered by a linear interferometer. Although in principle it applies to any bosonic system, its photonic version is the most natural one. While not universal, BS is strongly believed to be a classically computationally hard task. Traditionally its hardness has been shown in the *collision-free* regime $d \gg n^2$ (where d is the number of modes and n is the number of particles) [25], but recently proof techniques have been extended to cover also a more feasible scenario in which d scales linearly with n [26].

In the standard BS scenario, depicted in Fig. 2, the input state is a Fock basis state $|\mathbf{s}\rangle$ labeled by the occupation numbers $\mathbf{s} = (s_1, \ldots, s_d)$ with

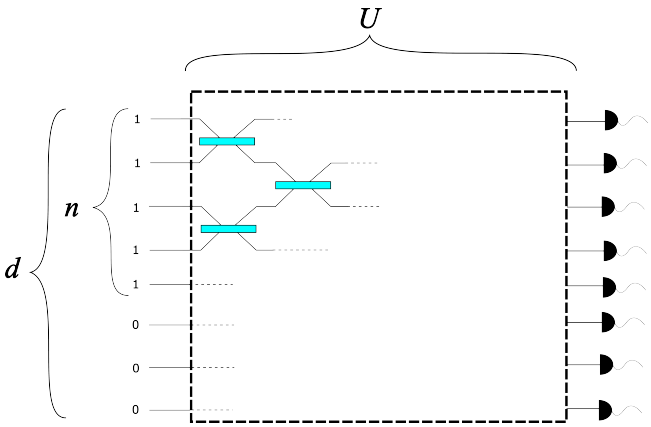


Figure 2: Boson Sampling experiment setup. The input state is an n-particle occupation number basis state, with typically only one or zero particles on each mode. The particles are then scattered through a (particle number preserving) interferometer U, which can be decomposed into two-mode beamsplitters. The output state is then measured by number-resolving detectors.

$n = \sum_{i=1}^{d} s_i$ total number of photons. Typically the s_i's are chosen to be 1's and 0's. The photons are then scattered in a generic passive linear optical interferometer described by a $d \times d$ unitary U. As a passive linear circuit preserves the particle number, the state vector is an element of the $\binom{n+d-1}{d-1}$ dimensional n-particle subspace during the entire evolution. Finally, a particle number measurement is performed on each mode, and the probability of the measurement outcome $\mathbf{t} = (t_1, t_2, \ldots, t_d)$ is given by the permanent introduced in Eq. (A.43) from Appendix A.

An example usage of the `SamplingSimulator` class is shown in Code snippet 4. The available gates are passive linear gates, and the only available measurement for this simulator is photon detection.

The effect of Gaussian losses on Boson Sampling can also be simulated in Piquasso. The simplest type is the pure-loss channel (with loss characterized by transmissivity η) on a mode, which can be modeled as an interaction of the mode with an environmental mode through a beamsplitter with transmittivity and reflectivity parameters $t = \sqrt{\eta}$ and $r = \sqrt{1-\eta}$, see Fig. 3.

Given that the number of particles in the output states is generally smaller than in the inputs, one can no longer use a unitary (particle number preserving) matrix to model the interferometer. Instead, one has to use a matrix A for which $AA^\dagger \leq \mathbb{I}$, where $\mathbb{I}$ denotes the identity (of proper size). Using SVD, A can be rewritten

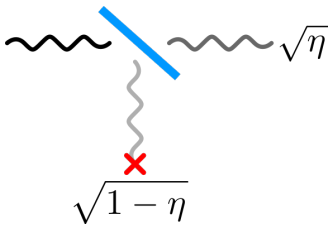


Figure 3: Losses modeled by a beamsplitter. Assume that a photon in a given mode has a survival probability $\eta \in [0, 1]$. Such losses can theoretically be replaced by a beamsplitter transporting photons to an inaccessible mode, with a reflectance amplitude $\sqrt{1-\eta}$.

as $A = V \operatorname{diag}(\mu) W$, where V and W are unitaries, and $\mu = (\sqrt{\eta_1}, \sqrt{\eta_2}, ..., \sqrt{\eta_m})$ is the vector of singular values. The values $\eta_i \in [0, 1]$ can be interpreted as transmissions on the i-th mode [27, 28, 29].

3.3 Extending Piquasso

Piquasso is customizable by subclassing the abstract classes in the API as shown in Code snippet 8. As an example, if the user only wants to customize a computation-heavy function, the new function could be specified by overriding the `NumpyConnector` class, which could be defined in a `Simulator` class as seen in Code snippet 9. Currently, supported built-in connectors are `NumpyConnector` (default), `TensorflowConnector` and `JaxConnector`.

One notable example of custom implementation is the PiquassoBoost plugin, delivering a capability beyond what conventional quantum computer emulator libraries offer. Specifically, it provides FPGA-based support for emulating Boson Sampling experiments [30], a capability that sets it apart from other libraries. PiquassoBoost can be easily used along Piquasso as demonstrated by Code snippet 10. The source code of PiquassoBoost is made available at [31].

4 Benchmarks

Piquasso aims to provide a collection of optimized algorithms to speed up the evaluation of specific computational tasks. During the development, several questions arose regarding performance. To answer these present and future questions, the algorithms are regularly examined and profiled using certain benchmarks and scripts. In this section, we present some of the current benchmarks

in detail.

4.1 Gaussian Boson Sampling

The concept of the Gaussian Boson Sampling (GBS), i.e., the photon-detection measurement of a general Gaussian state, was first introduced in Refs. [32, 33]. Similarly to the conventional Boson Sampling, the classical resources needed to take a sample from a complex probability distribution of Gaussian photonic states tend to scale exponentially with the number of the involved photons making the problem classically intractable. The complexity of taking a sample from a non-displaced Gaussian state is characterized by the evaluation of a matrix function, called the hafnian (for details, see Refs. [32, 34, 35]). The hafnian of a matrix can be considered as a generalization of the permanent: while the permanent enumerates the number of perfect matchings of a bipartite graph using its adjacency matrix, the hafnian yields the number of perfect matchings of a general graph. It was first introduced in Ref. [36] in a study of bosonic quantum field theory. For more details regarding Gaussian Boson Sampling see Appendix A.4.2.

Currently, the state-of-the-art calculation of the hafnian is the power trace algorithm with time complexity $O(n^3 2^{n/2})$ as described in Ref. [37], which can be further reduced in actual use cases when multiple photons occupy the same optical mode [38]. These works also generalized the concept of hafnian for graphs including loops (i.e., generalized adjacency matrices containing nonzero diagonal elements) which enables the simulation of GBS with displaced Gaussian states as well [37]. The so-called loop hafnian differs from the hafnian by including corrections related to the nonzero diagonal matrix elements. The introduction of the loop hafnian also enabled a quadratic speed-up in the classical simulation of GBS [35].

The implementation of the hafnian can be given in a Ryser-type (inclusion-exclusion) or a Glynn-type strategy, as in the implementation of the permanent function given in Refs. [39, 40]. It turns out that the Glynn-type implementation is more precise than the Ryser formula. In the Ryser formulation, the inner addends are computed from submatrices of the input matrix, which can potentially lead to a wide range of addend magnitudes. In contrast, the Glynn variant derives its addends from matrices of consistent size, which might be the reason behind the significantly better numerical stability of this approach. The same conclusion holds on for the case of the loop hafnian and the permanent. We benchmarked the Glynn-type implementations of the hafnian and the loop hafnian between Piquasso and TheWalrus on Figure 4 and Figure 5, respectively. For completeness, we also compare the execution times of GBS between Piquasso and Strawberry Fields on Fig. 6. In Piquasso, we implement the sampling algorithm as described in Ref. [35], and we implement the same batching strategy as in Ref. [38] for the loop hafnian, and we also use the halving of the Glynn-type iterations.

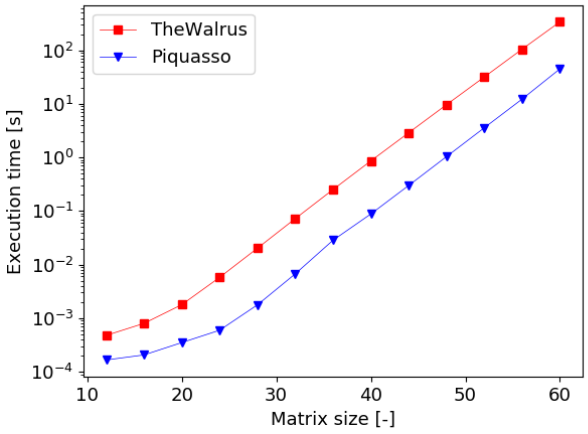


Figure 4: Comparing the average repeated hafnian execution times of Piquasso (version 5.0.0) and TheWalrus (version 0.21.0) with increasing input matrix sizes. The repetitions were chosen to be $(2, \ldots, 2)$ in each case. When the execution times are under 1 second, they are averaged over 100 runs, otherwise, a single run is executed. The benchmark was executed on a *Intel Xeon E5-2650* processor platform. The script is available at [41]. The script is available at [42].

The computational cost of evaluating the torontonian function, which was introduced for GBS with threshold detection [44, 45], was extensively studied in our previous work reported in Ref. [15]. Here, we reported on a polynomial reduction on the computational reduction, that was based on the reuse of intermediate results. Quantum computing simulator softwares including Piquasso and Strawberry Fields already take advantage of this approach, providing the fastest way of computing the torontonian. By Ref. [46], the method was further extended with loop cor-

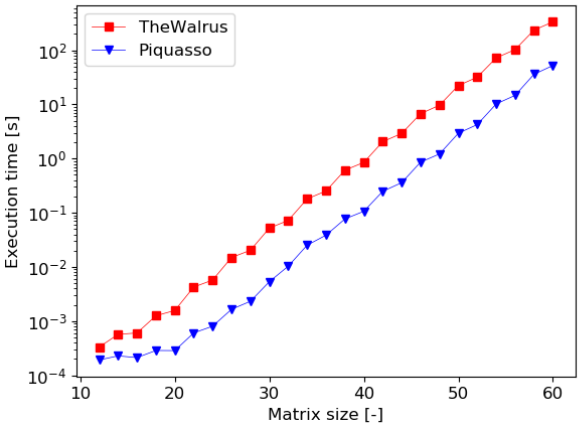


Figure 5: Comparing the average repeated loop hafnian execution times of Piquasso (version 5.0.0) and TheWalrus (version 0.21.0) with increasing input matrix sizes. The repetitions were chosen to be $(2, \ldots, 2)$ in each case. When the execution times are under 1 second, they are averaged over 100 runs, otherwise, a single run is executed. The benchmark was executed on a *Intel Xeon E5-2650 processor* platform. The script is available at [43].

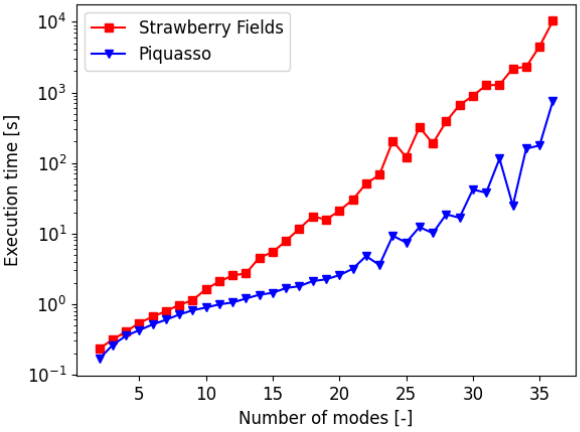


Figure 6: Performance benchmark of Gaussian Boson Sampling with Piquasso (version 5.0.0) and Strawberry Fields (version 0.23.0). The circuit was chosen to be a layer of squeezing gates with squeezing parameters $r = \operatorname{arcsinh}(1)$ consistently (to ensure that the average number of particles equal the number of modes), followed by an interferometer parametrized by a Haar-random unitary matrix. For each data point, 100 samples were taken. The benchmarks were executed on a *AMD EPYC 7542 32-Core* processor platform. The script is available at [41].

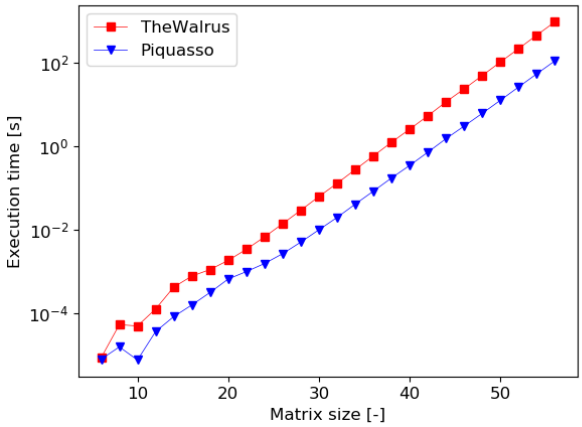


Figure 7: Comparing the average torontonian execution times of Piquasso (version 5.0.0) and TheWalrus (version 0.21.0) with increasing input matrix sizes. When the execution times are under 1 second, they are averaged over 100 runs, otherwise, a single run is executed. The benchmark was executed on a *Intel Xeon E5-2650 processor* platform. The script is available at [47].

rections introducing the concept of loop torontonian, applicable for displaced Gaussian states.

We show the performance benchmark results of the torontonian and the loop torontonian implementation on Figure 7 and Figure 8, respectively; comparing Piquasso (version 5.0.0) and TheWalrus (version 0.21.0).

4.2 Boson Sampling

As discussed in Sec. 3.2.4, the permanent is the key mathematical function determining the performance of the Boson Sampling simulations. In our recent work, we have given a detailed analysis of state-of-the-art permanent calculation methods, including their performance and numerical precision [30]. We have shown that the main advantage of the Balasubramanian-Bax-Franklin-Glynn (BB/FG) formula [40] over the Ryser variant [39] lies in the precision of the calculated permanent value. The numerical experiments conducted with the MPFR multi-precision library [49] revealed that Ryser's formula significantly lags behind the BB/FG method in terms of accuracy. In order to increase the maximal number of photons in a simulation of an optical interferometer, we have extended the basic BB/FG approach to accommodate scenarios where the input matrix exhibits column or row multiplicities, representing multiple particles oc-

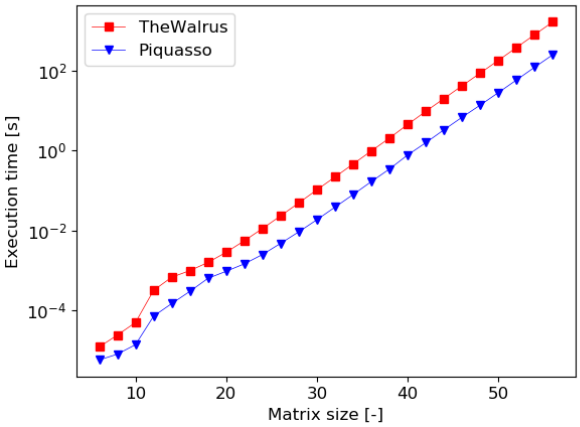


Figure 8: Comparing the average loop torontonian execution times of Piquasso (version 5.0.0) and TheWalrus (version 0.21.0) with increasing input matrix sizes. When the execution times are under 1 second, they are averaged over 100 runs, otherwise, a single run is executed. The benchmark was executed on a *Intel Xeon E5-2650 processor* platform. The script is available at [48].

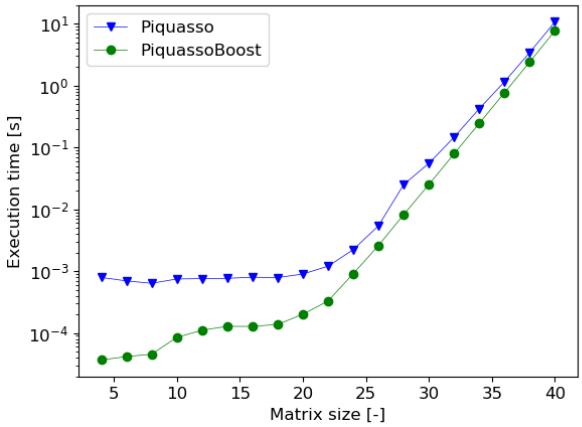


Figure 9: The average repeated permanent execution times of Piquasso (version 5.0.0) and PiquassoBoost with increasing input matrix sizes. The repetitions were chosen to be $(2, \ldots, 2)$ in each case. The data points are calculated from an average of 100 runs. The benchmark was executed on a *AMD EPYC 7542 32-Core* processor platform. The script used for benchmarking is available at [53].

cupying a single photonic mode. This expansion is achieved by utilizing a generalized n-ary Gray code ordering for the outer sum in the BB/FG permanent formula. The digits in this code range from zero to the occupation number of individual optical modes. This generalization extends the applicability of the BB/FG formula, reducing computational complexity, as exemplified in our recent work on high-performance BS simulation [30]. Piquasso implements the proposed algorithm, providing better numerical accuracy in evaluating the permanent than alternative approaches [50, 51, 52] based on the Ryser formula, while not losing computational performance. The perfornance benchmark of the Piquasso and PiquassoBoost implementation is shown in Fig. 9.

In Piquasso, the permanent function is utilized for classically simulating ideal and lossy BS with Algorithm A in Ref. [52], which is the current state-of-the-art approach. This algorithm enables fast classical simulation of BS, as depicted in Figure 10. Here, we compare Piquasso, PiquassoBoost, and Perceval as frameworks implementing efficient classical BS algorithms. This shows, that the execution times of Piquasso and Perceval are closely aligned, while PiquassoBoost has a significant speedup over both for smaller systems. For larger systems, the execution times of Perceval and PiquassoBoost are similar. Using PiquassoBoost, it is also possible to execute an enhanced algorithm with the help of data-flow engines, as described in Ref. [30].

4.3 Fock-space based simulations

As already discussed in Sec. 3.2.1, when using `PureFockSimulator` and `FockSimulator`, the truncation of the bosonic Fock space in Eq. (1) may introduce loss of probability during the simulation. Storing a state vector in the truncated bosonic Fock space requires

$$\dim_{\mathbb{C}} \left(\mathcal{F}_B^c(\mathbb{C}^d) \right) = \binom{d+c-1}{c-1} \tag{3}$$

complex numbers. This truncation amounts to a "global" cutoff, discarding any contribution to the state vector corresponding to a higher total particle number than the cutoff c. Note, that storing the state vector with a "local" cutoff instead would restrict the particle number for each mode individually. As a result, a pure state would practically be implemented as a tensor, which would require storing c^d complex numbers, significantly higher than the value using global cutoff. Hence, for equal values of local cutoff and global cutoff, the state vector in the truncated Fock space scenario generally has fewer vector elements, possibly leading to a decrease in precision. On the

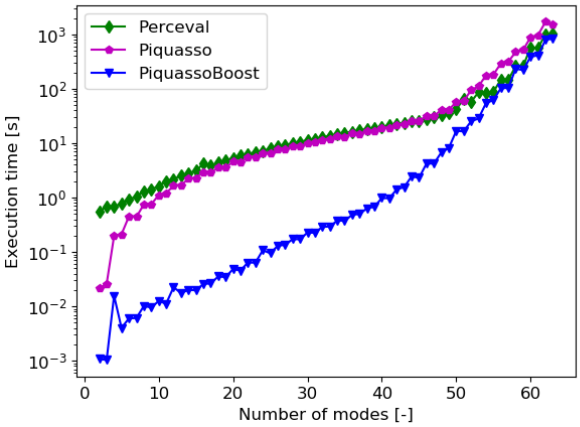


Figure 10: Performance benchmark of Boson Sampling with Piquasso (version 5.0.0), PiquassoBoost and Perceval (version 0.11.2). For the simulation, randomly generated Fock basis states were chosen as initial states, with number of particles half of the number of modes, and 100 samples were taken per datapoint under a runtime of 1s, otherwise 1 sample. The benchmarks were executed on a *AMD EPYC 7542 32-Core* processor platform. The script used for benchmarking is available at [54].

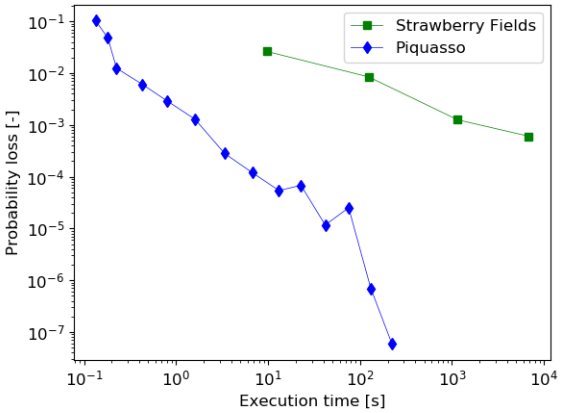


Figure 11: Comparison of execution time and probability loss for Piquasso (version 5.0.0) and Strawberry Fields (version 0.23.0) Fock simulation. The benchmark was executed on 10 modes for 4 CVQNN layers, with variable global or local cutoff values. The range of global cutoff in Piquasso simulations was $[3, 16]$ and the range of local cutoff in Strawberry Fields simulations was $[3, 6]$. For lower execution times, the probability loss and execution time values are averaged over 10 runs, while only 1 calculation is executed when the execution time surpasses $10s$. The Kerr gate angles and the interferometer angles in the CVQNN layers were generated by uniform random angles drawn from the normal distribution $\mathcal{N}(0, 1)$, and the squeezing and displacement parameters were drawn from $\mathcal{N}(0, 0.1)$. The benchmark was executed on a *Intel Xeon E5-2650 processor* platform. The script is available at [55].

bright side, one could argue that the contributions that are left out by using a global cutoff instead of a local one have small coefficients in most cases. Moreover, after the truncation using the global cutoff (for example on squeezed coherent states), the application of passive elements (e.g., Beamsplitter, Kerr) will not lead to further loss of probability during the simulation, while this is generally not true for the local cutoff. To illuminate this difference, the loss of probability has been calculated in two scenarios and illustrated in Fig. 11, where Piquasso implements a global cutoff, as opposed to the local cutoff implementation of Strawberry Fields. The benchmark was executed for 10 modes and 4 continuous-variable quantum neural network (CVQNN) layers, as described in [56]. This benchmark shows that for any probability loss, the Piquasso Fock simulator implementing the global cutoff has a lower execution time than Strawberry Fields with a local cutoff.

4.3.1 Differentiability

For machine learning purposes, a fast calculation of the gradient corresponding to a photonic circuit has the utmost importance. To illustrate the ability of Piquasso to perform such tasks swiftly, we have performed benchmarks assessing the execution time of the gradient of a certain cost function. More specifically, the benchmarks introduced in this section are based on one learning step in a state learning algorithm, i.e., training a quantum circuit to fit a certain target state [57]. The parameters of the quantum gates consisting of a single layer in the photonic circuit represent weights in the photonic neural network, and hence the gradient is calculated with respect to these parameters.

Considering a target state $|\psi_{\text{target}}\rangle$, we can define a cost function based on the distance of the output state and the desired target state induced by the 2-norm as

$$J(|\psi\rangle) = \||\psi\rangle - |\psi_{\text{target}}\rangle\|_2 \,, \tag{4}$$

where $|\psi\rangle$ is the resulting state after executing the circuit, depending on the weights of the neural network.

The execution times of the cost function gradients are shown for Piquasso and Strawberry Fields in Fig. 12. The benchmarks are executed

with a global cutoff of 6 for Piquasso and a local cutoff of 4 for Strawberry Fields, matching the probability loss of $\sim 10^{-2}$ according to Figure 11. Note, that the probability loss comparison has been executed for 10 modes, where the execution times match regardless of the framework. Surprisingly, the execution times approximately match until 8 modes, which can be explained by the framework overhead of TensorFlow. However, this difference diminishes as we increase the number of modes. For example, as indicated by the red line in Figure 11, when the number of modes is 10, a significant speedup of the global cutoff (Piquasso) implementation over the local cutoff implementation (Strawberry Fields) can be observed. In the Piquasso benchmark, the execution times increase exponentially for a given cutoff by increasing the number of modes, and the exponent appears to be linear. Meanwhile, in the Strawberry Fields benchmark, the execution times hit the exponential wall (in the logarithmic scale) by increasing the number of modes. In conclusion, the approaches using the local and global cutoff have a significant scaling difference in terms of the execution time. Moreover, due to the nature of the local cutoff (see Sec. 4.3) implemented in Strawberry Fields, its benchmark could not be executed for larger systems because of excessive memory usage.

For repetitive tasks, it can be beneficial to compile Piquasso code using `TensorflowConnector` into a callable TensorFlow graph, using `tf.function`. This feature, after the initial trace-compilation is executed, provides a significant speedup in `PureFockSimulator` by executing the optimized TensorFlow graph. A basic example is given by Code snippet 7. The graph execution via Piquasso is compared to Strawberry Fields with a benchmark on Figure 13, which shows a significant speedup of the compiled Piquasso code over Strawberry Fields.

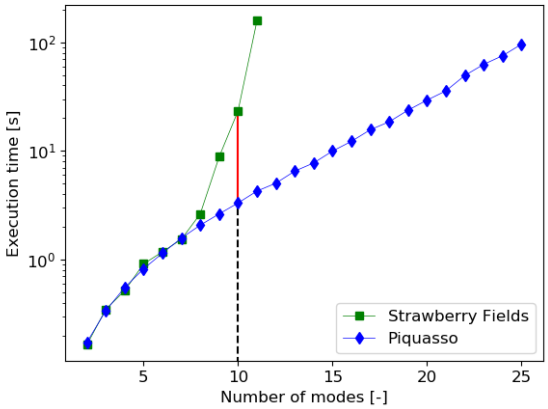


Figure 12: Comparison of gradient execution times of Piquasso (version 5.0.0) using a global cutoff with the execution times of Strawberry Fields (version 0.23.0) using a local cutoff. The Piquasso implementation used a global cutoff of 6, whereas the Strawberry Fields implementation uses a local cutoff of 4, matching the probability loss $\sim 10^{-2}$ according to Figure 11. The circuit layout is the same as for Figure 11, and the weights were also chosen from the same distribution. The benchmarks were performed using *AMD EPYC 7742P* architecture. The script is available at [58].

5 Web User Interface

Piquasso offers an intuitive web-based interface that allows users to create photonic circuits seamlessly using an interactive drag-and-drop circuit composer, while also enabling them to collaborate with others and share their results effortlessly. Authentication is supported through social media accounts and unlocks additional features, including project saving and publication, as well as a streamlined collaboration with other users.

5.1 Starting a new project

When creating a new project, the user must begin by selecting one of the available backend schemes, which were described in Sec. 3.2. Once a backend scheme has been selected, the Piquasso composer is launched. This tool provides an interactive drag-and-drop interface that enables the creation of photonic circuits with ease. The toolbar section of the interface displays the operations that are available for the selected scheme. These operations are grouped according to their respective types, such as 1-mode, 2-mode, multimode gates, measurements, and so on. To create a new circuit, the user simply needs to drag-and-drop the blocks that represent photonic operations onto the corresponding modes within the composer. The key components of the drag-and-drop interface are presented in Fig. 14.

Once the photonic circuit is finalized in the composer, the user can submit it through the control panel; the submitted job will be sent to the backend server for execution. The control panel offers several functionalities, such as starting or

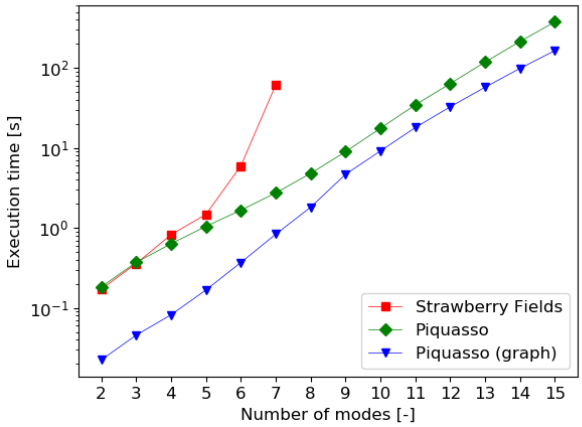


Figure 13: Comparison of the execution times between the Piquasso `PureFockSimulator` using `TensorflowConnector` (version 5.0.0) with and without graph compilation and the Strawberry Fields Fock backend (version 0.23.0). During the calculation, the setup described by Code snippet 7 was benchmarked with 4 CVQNN layers and cutoff 10 for both frameworks, the same setup as in Figure 11. When the execution times were under 1 second, the results were averaged over 100 runs, otherwise over 10 runs. We were forced to stop the calculation with Strawberry Fields due to excessive memory usage. The benchmarks were executed on a *Intel Xeon E5-2650 processor* platform using 12 × 16GB of RAM. The script is available at [59]

stopping a simulation and undoing or redoing previous actions. As soon as the submitted job is executed, the results are visualized on the dashboard, as shown in Fig. 15, and can be conveniently exported and downloaded in a variety of file formats, including `PNG` and `CSV`.

The project management menu provides users with an overview of their existing projects and of projects shared with them. Users can easily access previous projects and view detailed information about them including previously obtained results from past executions of the circuit. Furthermore, users can modify project settings such as the project name and collaborators directly from this menu.

5.2 Collaboration

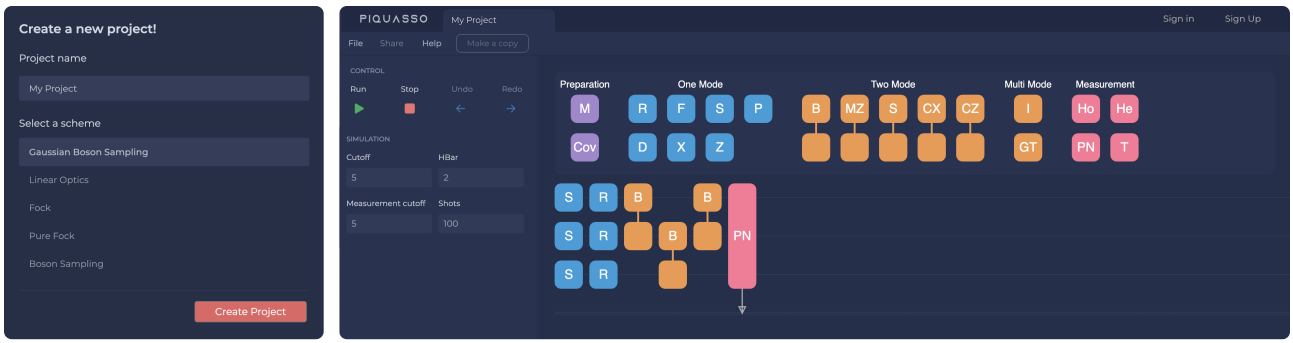
One of our primary focus has been on enabling users to collaborate and share simulation data. To this end, we have implemented several features such as real-time data sharing, version control, and commenting functionality.

The most prominent feature is the capability to add other Piquasso members directly to a project, enabling project members to collaborate in real-time on the same circuit and review the outcomes of submitted jobs. Depending on their role in the project, members can be granted different levels of access permissions, such as read-only access or full control over the circuit design. Write access allows users to freely modify the circuit, while read-only access allows users to view and execute the circuit yet forbidding any modifications. Project leaders have complete control over managing these permissions, as well as the ability to modify, rename, or share the project with others.

Another powerful example of collaboration and data sharing is the publishing feature. Users can publish their circuits and quantum algorithms to the Piquasso community for others to see, run, and build upon. This feature opens up opportunities for users to share their work with the wider scientific community. Furthermore, these published circuits can be shared via a generated URL online, e.g., in publications or on social media platforms.

6 Conclusion

We developed a photonic quantum computer simulator, named Piquasso, whose creation is driven by the increasing need for research in photonic quantum computing. The framework provides several specialized simulators tailored for specific computing schemes, along with a convenient and extendable Python interface, which have been explored in the current article. One such extension is the C/C++ plugin called PiquassoBoost which incorporates a high-performance engine into the Python framework, considerably aiding computational performance. Other extensions enable automatic differentiation of photonic circuits using popular machine learning frameworks such as TensorFlow and JAX. Finally, an ergonomic drag-and-drop web interface is made available at piquasso.com, which makes simulation data sharing and collaboration simple and accessible, empowering users to work together to push the boundaries of quantum computing and technological innovation.



a.) b.)

Figure 14: An overview of the Piquasso web interface. The user should (a) choose a supported backend scheme before creating a circuit, and then (b) use the interactive drag-and-drop composer to create the circuit from the relevant components of the chosen scheme. The depicted example is a Gaussian Boson Sampling circuit, the simulation results are visualized in Fig. 15.

7 Acknowledgements

We would like to thank Zsófia Kallus, Gábor Németh and the Ericsson Research team for the inspiring discussions. We also thank the Wigner Scientific Computing Laboratory for their support. This research was supported by the Ministry of Culture and Innovation and the National Research, Development and Innovation Office through the Quantum Information National Laboratory of Hungary (Grant No. 2022-2.1.1-NL-2022-00004), the UNKP-22-5 New National Excellence Program, Grants No. K134437 and FK135220. ZZ also acknowledges the QuantERA II project HQCC-101017733. RP was supported by the Hungarian Academy of Sciences through the Bolyai János Stipendium (BO/00571/22/11). MO and TR acknowledge financial support from the Foundation for Polish Science via the TEAM-NET project (contract no. POIR.04.04.00- 00-17C1/18-00).

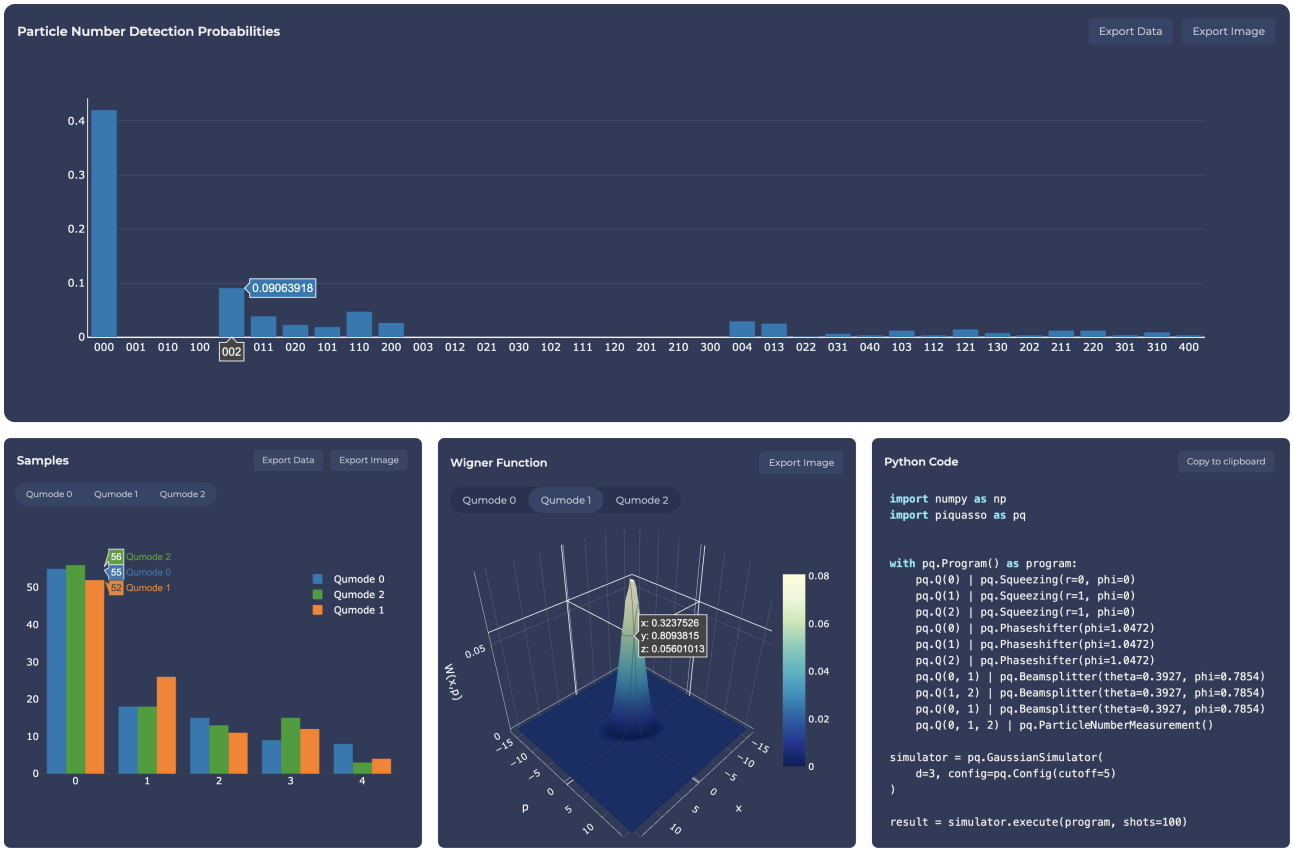


Figure 15: The simulated circuit results are visualized including the obtained measurement samples, the theoretical probability distribution, the Wigner function, and the corresponding Python code.

References

[1] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, Peng Hu, Xiao-Yan Yang, Wei-Jun Zhang, Hao Li, Yuxuan Li, Xiao Jiang, Lin Gan, Guangwen Yang, Lixing You, Zhen Wang, Li Li, Nai-Le Liu, Chao-Yang Lu, and Jian-Wei Pan. "Quantum computational advantage using photons". Science **370**, 1460–1463 (2020).

[2] Han-Sen Zhong, Yu-Hao Deng, Jian Qin, Hui Wang, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Dian Wu, Si-Qiu Gong, Hao Su, Yi Hu, Peng Hu, Xiao-Yan Yang, Wei-Jun Zhang, Hao Li, Yuxuan Li, Xiao Jiang, Lin Gan, Guangwen Yang, Lixing You, Zhen Wang, Li Li, Nai-Le Liu, Jelmer J. Renema, Chao-Yang Lu, and Jian-Wei Pan. "Phase-Programmable Gaussian Boson Sampling Using Stimulated Squeezed Light". Phys. Rev. Lett.**127** (2021).

[3] Lars S Madsen, Fabian Laudenbach, Mohsen Falamarzi Askarani, Fabien Rortais, Trevor Vincent, Jacob FF Bulmer, Filippo M Miatto, Leonhard Neuhaus, Lukas G Helt, Matthew J Collins, et al. "Quantum computational advantage with a programmable photonic processor". Nature **606**, 75–81 (2022).

[4] Sara Bartolucci, Patrick Birchall, Hector Bombin, Hugo Cable, Chris Dawson, Mercedes Gimeno-Segovia, Eric Johnston, Konrad Kieling, Naomi Nickerson, Mihir Pant, et al. "Fusion-based quantum computation". Nature Comm. **14**, 912 (2023).

[5] Hector Bombin, Chris Dawson, Ryan V Mishmash, Naomi Nickerson, Fernando Pastawski, and Sam Roberts. "Logical blocks for fault-tolerant topological quantum computation". PRX Quantum **4**, 020303 (2023).

[6] J Eli Bourassa, Rafael N Alexander, Michael Vasmer, Ashlesha Patil, Ilan Tzitrin, Takaya Matsuura, Daiqin Su, Ben Q Baragiola, Saikat Guha, Guillaume Dauphinais, et al.

"Blueprint for a scalable photonic fault-tolerant quantum computer". Quantum **5**, 392 (2021).

[7] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. "Open source software in quantum computing". PLOS ONE **13**, e0208561 (2018).

[8] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. "Strawberry Fields: A Software Platform for Photonic Quantum Computing". Quantum **3**, 129 (2019).

[9] Guido Van Rossum and Fred L. Drake. "Python 3 Reference Manual". CreateSpace. Scotts Valley, CA (2009). url: https://www.python.org/.

[10] Nicolas Heurtel, Andreas Fyrillas, Grégoire de Gliniasty, Raphaël Le Bihan, Sébastien Malherbe, Marceau Pailhas, Eric Bertasi, Boris Bourdoncle, Pierre-Emmanuel Emeriau, Rawad Mezher, Luka Music, Nadia Belabas, Benoît Valiron, Pascale Senellart, Shane Mansfield, and Jean Senellart. "Perceval: A Software Platform for Discrete Variable Photonic Quantum Computing". Quantum **7**, 931 (2023).

[11] Timothy J Stavenger, Eleanor Crane, Kevin Smith, Christopher T Kang, Steven M Girvin, and Nathan Wiebe. "Bosonic Qiskit" (2022). arXiv:2209.11153.

[12] Jie Lin, Benjamin MacLellan, Sobhan Ghanbari, Julie Belleville, Khuong Tran, Luc Robichaud, Roger G. Melko, Hoi-Kwong Lo, and Piotr Roztocki. "GraphiQ: Quantum circuit design for photonic graph states". Quantum **8**, 1453 (2024).

[13] Yuan Yao, Filippo Miatto, and Nicolás Quesada. "Riemannian optimization of photonic quantum circuits in phase and Fock space". SciPost Phys. (2024).

[14] Benoit Seron and Antoine Restivo. "BosonSampling.jl: A Julia package for quantum multi-photon interferometry". Quantum **8**, 1378 (2024).

[15] Ágoston Kaposi, Zoltán Kolarovszki, Tamás Kozsik, Zoltán Zimborás, and Péter Rakyta. "Polynomial speedup in Torontonian calculation by a scalable recursive algorithm" (2022). arXiv:2109.04528.

[16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-scale Machine Learning on Heterogeneous Systems". https://www.tensorflow.org/ (2015).

[17] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. "JAX: composable transformations of Python+NumPy programs". http://github.com/google/jax (2018).

[18] Csaba Czabán, Zoltán Kolarovszki, Márton Karácsony, and Zoltán Zimborás. "Suppressing photon detection errors in nondeterministic state preparation". In Proceedings of Recent Advances in Quantum Computing and Technology. ReAQCT '24 (2024).

[19] Zoltán Zimborás Zoltán Kolarovszki, Dániel T. R. Nagy. "On the learning abilities of photonic continuous-variable Born machines". In In proceedings of 2024 IEEE International Conference on Quantum Computing and Engineering. 2024 IEEE International Conference on Quantum Computing and Engineering (QCE) (2024).

[20] Dániel T. R. Nagy, Csaba Czabán, Bence Bakó, Péter Hága, Zsófia Kallus, and Zoltán Zimborás. "Hybrid Quantum-Classical Reinforcement Learning in Latent Observation Spaces" (2024). arXiv:2410.18284.

[21] https://github.com/Budapest-Quantum-Computing-Group/piquasso.

[22] https://piquasso.readthedocs.io/en/v5.0.0/.

[23] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser,

Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. "Array programming with NumPy". Nature **585**, 357–362 (2020).

[24] Filippo M. Miatto and Nicolás Quesada. "Fast optimization of parametrized quantum optical circuits". Quantum **4**, 366 (2020).

[25] Scott Aaronson and Alex Arkhipov. "The computational complexity of linear optics". In Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing. STOC '11 (2011).

[26] Adam Bouland, Daniel Brod, Ishaun Datta, Bill Fefferman, Daniel Grier, Felipe Hernandez, and Michal Oszmaniec. "Complexity-theoretic foundations of BosonSampling with a linear number of modes" (2023). arXiv:2312.00286.

[27] Michał Oszmaniec and Daniel J. Brod. "Classical simulation of photonic linear optics with lost particles". New J. Phys. **20**, 092002 (2018).

[28] Daniel J. Brod and Michał Oszmaniec. "Classical simulation of linear optics subject to nonuniform losses". Quantum **4**, 267 (2020).

[29] Raúl García-Patrón, Jelmer J. Renema, and Valery Shchesnovich. "Simulating boson sampling in lossy architectures". Quantum **3**, 169 (2019).

[30] Gregory Morse, Tomasz Rybotycki, Ágoston Kaposi, Zoltán Kolarovszki, Uroš Stojčić, Tamás Kozsik, Oskar Mencer, Michał Oszmaniec, Zoltán Zimborás, and Péter Rakyta. "High performance Boson sampling simulation via data-flow engines". New J. Phys. **26**, 033033 (2024).

[31] https://github.com/Budapest-Quantum-Computing-Group/piquassoboost.

[32] Craig S. Hamilton, Regina Kruse, Linda Sansoni, Sonja Barkhofen, Christine Silberhorn, and Igor Jex. "Gaussian Boson Sampling". Phys. Rev. Lett. **119**, 170501 (2017).

[33] Regina Kruse, Craig S. Hamilton, Linda Sansoni, Sonja Barkhofen, Christine Silberhorn, and Igor Jex. "Detailed study of Gaussian boson sampling". Phys. Rev. A **100**, 032326 (2019).

[34] Nicolás Quesada and Juan Miguel Arrazola. "Exact simulation of Gaussian boson sampling in polynomial space and exponential time". Phys. Rev. Research **2**, 023005 (2020).

[35] Nicolás Quesada, Rachel S. Chadwick, Bryn A. Bell, Juan Miguel Arrazola, Trevor Vincent, Haoyu Qi, and Raúl García−Patrón. "Quadratic speed-up for simulating gaussian boson sampling". PRX Quantum **3**, 010306 (2022).

[36] Eduardo R. Caianiello. "Combinatorics & Renormalization in Quantum Field Theory". Volume 38. Benjamin. Reading (1973). url: https://www.osti.gov/biblio/4338754.

[37] Andreas Björklund, Brajesh Gupt, and Nicolás Quesada. "A faster hafnian formula for complex matrices and its benchmarking on a supercomputer". ACM J. Exp. Algorithmics (2019).

[38] Jacob F. F. Bulmer, Bryn A. Bell, Rachel S. Chadwick, Alex E. Jones, Diana Moise, Alessandro Rigazzi, Jan Thorbecke, Utz-Uwe Haus, Thomas Van Vaerenbergh, Raj B. Patel, Ian A. Walmsley, and Anthony Laing. "The Boundary for Quantum Advantage in Gaussian Boson Sampling". Sci. Adv. **8**, eabl9236 (2022).

[39] Herbert J. Ryser. "Combinatorial mathematics". The Carus Mathematical Monographs. Mathematical Association of America. (1963). url: https://www.cambridge.org/core/books/combinatorial-mathematics/8AB6985C13895FAA27999FC5EDABA7AD.

[40] David G. Glynn. "Permanent formulae from the Veronesean". Des. Codes Cryptogr. **68**, 39–47 (2013).

[41] https://github.com/Budapest-Quantum-Computing-Group/piquasso/blob/v5.0.0/scripts/gaussian_boson_sampling_benchmark.py.

[42] https://github.com/Budapest-Quantum-Computing-Group/piquasso/blob/v5.0.0/scripts/hafnian_benchmark.py.

[43] https://github.com/Budapest-Quantum-Computing-Group/piquasso/blob/v5.0.0/scripts/loop_hafnian_benchmark.py.

[44] Nicolás Quesada, Juan Miguel Arrazola, and Nathan Killoran. "Gaussian boson sampling using threshold detectors". Phys. Rev. A **98**, 062322 (2018).

[45] Yuxuan Li, Lin Gan, Mingcheng Chen, Yaojian Chen, Haitian Lu, Chaoyang Lu, Jianwei Pan, Haohuan Fu, and Guangwen Yang. "Benchmarking 50-Photon Gaussian Boson Sampling on the Sunway TaihuLight". IEEE Trans. Parallel Distrib. Syst. **33**, 1357–1372 (2022).

[46] J. F. F. Bulmer, S. Paesani, R. S. Chadwick, and N. Quesada. "Threshold detection statistics of bosonic states". Phys. Rev. A **106**, 043712 (2022).

[47] https://github.com/Budapest-Quantum-Computing-Group/piquasso/blob/v5.0.0/scripts/torontonian_benchmark.py.

[48] https://github.com/Budapest-Quantum-Computing-Group/piquasso/blob/v5.0.0/scripts/loop_torontonian_benchmark.py.

[49] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding". ACM Trans. Math. Softw. **33**, 13–es (2007).

[50] Seungbeom Chin and Joonsuk Huh. "Generalized concurrence in boson sampling". Sci. Rep. **8**, 6101 (2018).

[51] P.H. Lundow and K. Markström. "Efficient computation of permanents, with applications to Boson sampling and random matrices". J. Comput. Phys. **455**, 110990 (2022).

[52] Peter Clifford and Raphaël Clifford. "Faster classical Boson Sampling". Phys. Scr. **99**, 065121 (2024).

[53] https://github.com/Budapest-Quantum-Computing-Group/piquassoboost/blob/v0.3.0/boostbenchmarks/permanent_benchmark.py.

[54] https://github.com/Budapest-Quantum-Computing-Group/piquassoboost/blob/v0.3.0/boostbenchmarks/boson_sampling_benchmark.py.

[55] https://github.com/Budapest-Quantum-Computing-Group/piquasso/blob/v5.0.0/scripts/probability_loss_comparison.py.

[56] Nathan Killoran, Thomas R. Bromley, Juan Miguel Arrazola, Maria Schuld, Nicolás Quesada, and Seth Lloyd. "Continuous-variable quantum neural networks". Phys. Rev. Res. **1**, 033063 (2019).

[57] Juan Miguel Arrazola, Thomas R Bromley, Josh Izaac, Casey R Myers, Kamil Brádler, and Nathan Killoran. "Machine learning method for state preparation and gate synthesis on photonic quantum computers". Quantum Sci. Technol. **4**, 024004 (2019).

[58] https://github.com/Budapest-Quantum-Computing-Group/piquasso/blob/v5.0.0/scripts/cvqnn_state_learning_benchmark.py.

[59] https://github.com/Budapest-Quantum-Computing-Group/piquasso/blob/v5.0.0/scripts/cvqnn_tensorflow_comparison_benchmark.py.

[60] N. Quesada, L. G. Helt, J. Izaac, J. M. Arrazola, R. Shahrokhshahi, C. R. Myers, and K. K. Sabapathy. "Simulating realistic non-Gaussian state preparation". Phys. Rev. A **100**, 022341 (2019).

A Basics and notations

A.1 Bosonic Fock space

The Hilbert space of a d-mode bosonic system of n particles is given by

$$\left(\mathbb{C}^d\right)^{\vee n} = \underbrace{\mathbb{C}^d \vee \mathbb{C}^d \vee \cdots \vee \mathbb{C}^d}_{n \text{ times}}, \tag{A.5}$$

where $\vee$ denotes the symmetrized tensor product. We also formally define the zero-particle Hilbert space $(\mathbb{C}^d)^{\vee 0} := \mathbb{C}$, which corresponds to the vacuum. Given an orthonormal basis $\{e_i\}_{i=1}^d$ on $\mathbb{C}^d$, we can directly construct an orthonormal basis on $\left(\mathbb{C}^d\right)^{\vee n}$ as

$$\left\{ \left(\tfrac{1}{n_1!\ldots n_d!}\right)^{\frac{1}{2}} e_{i_1} \vee \ldots \vee e_{i_n} : i_1 \leq \ldots \leq i_n \right\}, \tag{A.6}$$

where n_k denotes the repetition of the index k in the index sequence $i_1, \ldots, i_n$ and $n_1 + \cdots + n_d = n$. One could also denote the basis vectors more succinctly as

$$|n_1, \ldots, n_d\rangle \coloneqq \left(\tfrac{1}{n_1!\ldots n_d!}\right)^{\frac{1}{2}} e_{i_1} \vee \ldots \vee e_{i_n}. \tag{A.7}$$

Allowing for an arbitrary number of particles, the Hilbert space of a d-mode bosonic system is the bosonic Fock space given by the direct sum of all fixed particle Hilbert spaces

$$\mathcal{F}_B(\mathbb{C}_d) = \mathbb{C} \oplus \mathbb{C}^d \oplus \mathbb{C}^d \vee \mathbb{C}^d \oplus \mathbb{C}^d \vee \mathbb{C}^d \vee \mathbb{C}^d \oplus \ldots = \bigoplus_{\lambda=0}^{\infty} \left(\mathbb{C}^d\right)^{\vee \lambda}. \tag{A.8}$$

An orthonormal basis of $\mathcal{F}_B(\mathbb{C}_d)$ is given by the vectors $|n_1, \ldots, n_d\rangle$ with no restriction on the total particle number $n = n_1 + n_2 \cdots + n_d$, this is called the *occupation number basis* of the Fock space. The annihilation and creation operators (jointly called ladder operators) are defined as

$$\begin{aligned} \hat{a}_j |n_1, ..., n_j, ..., n_d\rangle &= \sqrt{n_j}\, |n_1, ..., n_j - 1, ..., n_d\rangle, \\ \hat{a}_j^\dagger |n_1, ..., n_j, ..., n_d\rangle &= \sqrt{n_j + 1}\, |n_1, ..., n_j + 1, ..., n_d\rangle. \end{aligned}$$

These operators satisfy the canonical commutation relations

$$[\hat{a}_j, \hat{a}_k] = [\hat{a}_j^\dagger, \hat{a}_k^\dagger] = 0, \quad [\hat{a}_j, \hat{a}_k^\dagger] = \delta_{jk} \mathbb{1}. \tag{A.9}$$

A.2 Typical gates in photonic systems

Piquasso supports numerous photonic quantum gates, which we will review in this section. Piquasso can also export any `Program` instance to Blackbird Quantum Assembly Language script [8]. To make the transition easier, we used the same parametrization of the gates as in Blackbird. Some of the most important ones are presented in this section.

Except for Sec. A.2.6, the gates presented here are one- and two-mode gates, with their indices labeling on which modes they act non-trivially. The phase shift, the beamsplitter, the squeezing, and the displacement gates are Gaussian gates. More precisely, they map ladder operators to a linear combination of ladder operators (and also the identity in the case of the displacement gate), which we will also provide. The Kerr gate is a non-linear gate, however, it preserves the particle number similarly to the phaseshift and beamsplitter gates.

A.2.1 Phaseshift gate

The phaseshift (or rotation) gate models the actual phase shifter optical element, which rotates the phase of a traveling electromagnetic wave. As a quantum gate, it acts on a state by rotating it in the canonical phase space. Additionally, the phaseshift gate is the only single-mode passive linear gate. The unitary operator corresponding to the phaseshift gate is

$$R_j(\phi) = \exp\left(i\phi\hat{n}_j\right), \tag{A.10}$$

where $\hat{n}_j \coloneqq \hat{a}_j^\dagger \hat{a}_j$ and $\phi \in [0, 2\pi)$. The phaseshift gate is a passive linear optical element that transforms the ladder operators in the Heisenberg picture as follows:

$$R_j^\dagger(\phi) \begin{bmatrix} \hat{a}_j \\ \hat{a}_j^\dagger \end{bmatrix} R_j(\phi) = \begin{bmatrix} e^{i\phi} & 0 \\ 0 & e^{-i\phi} \end{bmatrix} \begin{bmatrix} \hat{a}_j \\ \hat{a}_j^\dagger \end{bmatrix}. \tag{A.11}$$

In Piquasso, a phaseshift gate can be implemented with `pq.Phaseshifter`.

A.2.2 Beamsplitter gate

A beamsplitter is an optical device that splits light into two parts and is an essential part of many optical experiments. The unitary operator corresponding to the beamsplitter gate is

$$B_{jk}(\theta, \phi) = \exp\left(\theta e^{i\phi} \hat{a}_j^\dagger \hat{a}_k - \theta e^{-i\phi} \hat{a}_k^\dagger \hat{a}_j\right), \tag{A.12}$$

where $\theta, \phi \in [0, 2\pi)$. The beamsplitter gate transforms the ladder operators as

$$B_{jk}^\dagger(\theta, \phi) \begin{bmatrix} \hat{a}_j \\ \hat{a}_k \\ \hat{a}_j^\dagger \\ \hat{a}_k^\dagger \end{bmatrix} B_{jk}(\theta, \phi) = \begin{bmatrix} t & -r^* & & \\ r & t & & \\ & & t & -r \\ & & r^* & t \end{bmatrix} \begin{bmatrix} \hat{a}_j \\ \hat{a}_k \\ \hat{a}_j^\dagger \\ \hat{a}_k^\dagger \end{bmatrix}, \tag{A.13}$$

where $t = \cos(\theta)$ and $r = e^{i\phi}\sin(\theta)$. In Piquasso, a beamsplitter gate can be implemented with `pq.Beamsplitter`.

A.2.3 Squeezing gate

Considering a vacuum state as the initial state, the squeezing gate produces a state that is "squeezed" in the phase space along a certain direction. The squeezing gate is the prototypical example of an active linear gate. The unitary operator corresponding to the squeezing gate is

$$S_j(z) = \exp\left(\frac{1}{2}\left(z^* \hat{a}_j^2 - z\hat{a}_j^{\dagger 2}\right)\right), \tag{A.14}$$

where $z \in \mathbb{C}$. The squeezing gate transforms the ladder operators as

$$S_j^\dagger(z) \begin{bmatrix} \hat{a}_j \\ \hat{a}_j^\dagger \end{bmatrix} S_j(z) = \begin{bmatrix} \cosh r & -e^{i\phi}\sinh r \\ -e^{-i\phi}\sinh r & \cosh r \end{bmatrix} \begin{bmatrix} \hat{a}_j \\ \hat{a}_j^\dagger \end{bmatrix}, \tag{A.15}$$

where $z = re^{i\phi}$. In Piquasso, a squeezing gate can be implemented with `pq.Squeezing`.

A.2.4 Displacement gate

A displacement gate "displaces" the quantum state in the phase space. Starting from vacuum as an initial state, the resulting states after applying a displacement gate are called coherent states. The unitary operator corresponding to the displacement gate is

$$D_j(\alpha) = \exp\left(\alpha\hat{a}_j^\dagger - \alpha^*\hat{a}_j\right), \tag{A.16}$$

where $\alpha \in \mathbb{C}$. The displacement gate transforms the ladder operators as

$$D_j^\dagger(\alpha) \begin{bmatrix} \hat{a}_j \\ \hat{a}_j^\dagger \end{bmatrix} D_j(\alpha) = \begin{bmatrix} \hat{a}_j + \alpha \mathbb{1} \\ \hat{a}_j^\dagger + \alpha^* \mathbb{1} \end{bmatrix}. \tag{A.17}$$

In Piquasso, a displacement gate can be implemented with `pq.Displacement`.

A.2.5 Kerr gate

The Kerr gate is the most trivial example of a passive non-linear gate. The definition of the Kerr gate is

$$K_j(\kappa) = \exp\left(i\kappa \hat{n}_j^2\right), \tag{A.18}$$

where $\kappa \in [0, 2\pi)$. The Kerr gate transforms the ladder operators as

$$\begin{aligned} K_j^\dagger(\kappa) \hat{a}_j K_j(\kappa) &= \hat{a}_j \exp\left(i\kappa\left(2\hat{n}_j - \mathbb{1}\right)\right), \\ K_j^\dagger(\kappa) \hat{a}_j^\dagger K_j(\kappa) &= \hat{a}_j^\dagger \exp\left(-i\kappa\left(2\hat{n}_j - \mathbb{1}\right)\right). \end{aligned} \tag{A.19}$$

In Piquasso, a Kerr gate can be implemented with `pq.Kerr`.

A.2.6 General Gaussian gates

Gaussian unitaries can be characterized by Hamiltonian operators which contain only quadratic and linear terms. Concretely, one can write the Hamiltonian as

$$\begin{aligned} H &= \sum_{j,k=1}^{d} A_{jk} \hat{a}_j^\dagger \hat{a}_k + B_{jk} \hat{a}_j^\dagger \hat{a}_k^\dagger + \sum_{j=1}^{d} \beta_j \hat{a}_j + h.c. \\ &= \xi^\dagger \mathbf{H} \xi + \alpha \xi, \end{aligned} \tag{A.20}$$

with

$$\mathbf{H} = \begin{bmatrix} A & B \\ B^* & A^* \end{bmatrix}, \tag{A.21}$$

$$\xi = \left[\hat{a}_1, \ldots, \hat{a}_d, \hat{a}_1^\dagger, \ldots, \hat{a}_d^\dagger\right]^T, \tag{A.22}$$

where $A = A^\dagger$, $B = B^T$, $\alpha = [\beta, \beta^*]$, and $\beta \in \mathbb{C}^d$. When $\beta = 0_d$ and $B = 0_{d \times d}$, the Hamiltonian H corresponds to passive Gaussian gates, e.g., beamsplitters and phaseshifters. These are Gaussian gates that preserve the particle number. Generally, the quadratic part of the Hamiltonian can be split into passive and active parts, while the linear terms correspond to displacements. In particular, for $A = B = 0_{d \times d}$ and general $\beta \in \mathbb{C}^d$ the Hamiltonian H describes pure displacement gates.

Under a Gaussian transformation, Gaussian states are mapped to Gaussian states. The symplectic representation of unitary evolution can be described by

$$U^\dagger \xi U = S_{(c)} \xi + \alpha, \tag{A.23}$$

where $\alpha \in \mathbb{C}^{2d}$, $U = \exp\left(-i\left(\xi^\dagger \mathbf{H} \xi + \alpha \xi\right)\right)$ and $S_{(c)} \in \mathrm{Sp}(2d)$ is a symplectic matrix. Then $S_{(c)}$ can be calculated by

$$S_{(c)} = e^{-iK\mathbf{H}}, \quad \text{where} \quad K = \begin{bmatrix} \mathbb{1} & \\ & -\mathbb{1} \end{bmatrix}. \tag{A.24}$$

In Piquasso, a general Gaussian tranformation can be implemented with `pq.GaussianTransform`.

A.3 States and their evolution

In the simulation of photonic quantum computation, states can be represented in multiple ways. In this subsection, we will provide a short overview of the most important representations. We also describe the action of the most important gates in the different representations.

A.3.1 Generic states in the occupation number representation

A pure bosonic state is an element of the multimode bosonic Fock space defined in Eq. (A.8). Generally, it can be expanded in the canonical basis of this vector space, i.e., in the occupation number basis as

$$|\psi\rangle = \sum_{\mathbf{n}\in\mathbb{N}^d} c_{\mathbf{n}}|\mathbf{n}\rangle, \tag{A.25}$$

where $\langle\psi|\psi\rangle = \sum_{n\in\mathbb{N}^d} |c_n|^2 = 1$. Similarly, a general mixed state can be written as a density operator

$$\rho = \sum_{\mathbf{n},\mathbf{m}\in\mathbb{N}^d} c_{\mathbf{n},\mathbf{m}}|\mathbf{n}\rangle\langle\mathbf{m}|, \tag{A.26}$$

where $\operatorname{Tr}\rho = 1$ and ρ is positive semidefinite.

Given a quantum gate from Sec. A.2, the transformation of the ladder operators in the Heisenberg picture relates to the evolution of the quantum states in the Fock representation by Eq. (A.23). As an example, by applying a rotation gate from Sec. A.2.1 to an occupation number state, one gets

$$R_j(\phi)|n_1\dots n_j\dots n_d\rangle = e^{in_j\phi}|n_1\dots n_j\dots n_d\rangle. \tag{A.27}$$

Similarly, for a beamsplitter gate, one can write

$$B_{jk}(\theta,\phi)|n,m\rangle = \sum_{k=0}^{n}\sum_{l=0}^{m} c_{n,m}^{k,l}(r,t)|n-k+l, m+k-l\rangle, \tag{A.28}$$

where $c_{n,m}^{k,l}(r,t) = \binom{n}{k}\binom{m}{l}t^{n+m-k-l}(-r^*)^l r^k$.

When an active gate is applied to a fixed particle-number state, the resulting state will have terms in different particle-number sectors. As an illustration, the squeezing gate is applied to the vacuum as

$$S_j(z)|0_1\dots 0_j\dots 0_d\rangle = \frac{1}{\sqrt{\cosh r}}\sum_{k_j=0}^{\infty}\left(-e^{i\phi}\tanh r\right)^{k_j}\frac{\sqrt{(2k_j)!}}{2^{k_j}k_j!}|0_1\dots 2k_j\dots 0_d\rangle, \tag{A.29}$$

where $r = |z|$ and $\phi = \arg(z)$.

Finally, the action of the non-linear Kerr gate applied to an occupation number basis state is given as

$$K_j(\phi)|n_1\dots n_j\dots n_d\rangle = \exp\left(i\xi n_j^2\right)|n_1\dots n_j\dots n_d\rangle. \tag{A.30}$$

Similar results can be obtained when applying the above considerations for density operators. For such calculations, one could use `pq.PureFockSimulator` or `pq.FockSimulator` in Piquasso.

A.3.2 Gaussian states

Gaussian states on d modes can be characterized by their mean vector and covariance matrix (μ,σ) defined by

$$\begin{aligned}\mu_i &= \operatorname{Tr}\left[\rho R_i\right],\\ \sigma_{ij} &= \operatorname{Tr}\left[\rho\{R_i-\mu_i, R_j-\mu_j\}\right], \qquad (i,j=1,\dots,2d)\end{aligned} \tag{A.31}$$

where $\{A,B\} = AB+BA$ is the anticommutator and R is a vector of operators

$$R = (\hat{x}_1,\dots,\hat{x}_d,\hat{p}_1,\dots,\hat{p}_d)^T. \tag{A.32}$$

The evolution of the quantum state in terms of the mean vector and the covariance matrix can be given in terms of the evolution of the ladder operators in the Heisenberg picture as

$$\begin{aligned} \mu &\mapsto S\mu, \\ \sigma &\mapsto S\sigma S^T, \end{aligned} \tag{A.33}$$

where S is the symplectic matrix corresponding to the quantum gate in the canonical coordinate representation according to Eq. (A.32). For simulating Gaussian states, one could use `pq.GaussianSimulator` in Piquasso.

A.4 Typical measurements

A.4.1 Homodyne and heterodyne measurements

Homodyne and heterodyne measurements correspond to the usual detection schemes from optical setups. These measurements have the common property of preserving the Gaussian character of the state after measurement. Homodyne measurement corresponds to the measurement of the operator

$$\hat{x}_\phi = \cos(\phi)\hat{x} + \sin(\phi)\hat{p} \tag{A.34}$$

with outcome probability density given by

$$p(x_\phi) = \langle x_\phi | \rho | x_\phi \rangle, \tag{A.35}$$

where x_ϕ correspond to the eigenvalues of $\hat{x}_\phi$. On the other hand, using heterodyne measurement, the probability density is given by

$$p(x_\phi) = \frac{1}{\pi} \operatorname{Tr}\left[\rho |\alpha\rangle\langle\alpha|\right]. \tag{A.36}$$

In optical setups, the heterodyne measurement is performed by mixing the state ρ with a vacuum state $|0\rangle$, then subtracting the detected intensities of the two outputs. The mixing is performed with a 50:50 beamsplitter. Homodyne and heterodyne measurements can be implemented using `pq.HomodyneMeasurement` and `pq.HeterodyneMeasurement` in Piquasso.

A.4.2 Particle number measurement

Particle number measurement or photon detection is a non-Gaussian projective measurement that is performed via number-resolving detectors. The probability of detecting particle numbers $\mathbf{n} = (n_1, n_2, \ldots, n_d)$ is given by

$$p(\mathbf{n}) = \operatorname{Tr}\left[\rho |\mathbf{n}\rangle\langle\mathbf{n}|\right]. \tag{A.37}$$

The samples are non-negative integer values corresponding to the detected photon number.

When the state is a non-displaced Gaussian state, the probability of an output $\mathbf{t} = (t_1, \ldots, t_d)$ during particle number measurement is given by

$$p(\mathbf{t}) = \frac{1}{\sqrt{\det Q}} \frac{\operatorname{haf}\left(A_{\mathbf{t}}\right)}{t_1! \ldots t_d!}, \tag{A.38}$$

where $A_{\mathbf{t}}$ is the block reduction of the matrix A by the vector $\mathbf{t}$ and

$$\begin{aligned} A &= \begin{bmatrix} & \mathbb{1} \\ \mathbb{1} & \end{bmatrix} \left(\mathbb{1} - Q^{-1}\right), \\ Q &= \Sigma + \frac{1}{2}\mathbb{1}, \\ \Sigma &= W\sigma W^\dagger, \\ W &= \frac{1}{\sqrt{2}} \begin{bmatrix} \mathbb{1} & i\mathbb{1} \\ \mathbb{1} & -i\mathbb{1} \end{bmatrix} \end{aligned} \tag{A.39}$$

following Hamilton et. al. [32].

Furthermore, the hafnian of a matrix is defined by

$$\mathrm{haf}(A) = \sum_{M \in \mathrm{PMP}(n)} \prod_{(i,j)\in M} A_{i,j}, \tag{A.40}$$

where $\mathrm{PMP}(n)$ is the set of perfect matching permutations of n even elements, such that $\sigma(2i-1) < \sigma(2i)$ and $\sigma(2i-1) < \sigma(2i+1)$, where $\sigma : [n] \to [n]$ is some permutation.

When the Gaussian state is displaced with complex displacement vector $\boldsymbol{\alpha}$, the detection probability of an output $\mathbf{t}$ is given by

$$p(\mathbf{t}) = \frac{\exp\left(-\frac{1}{2}\boldsymbol{\alpha}^\dagger \Sigma^{-1} \boldsymbol{\alpha}\right)}{\sqrt{\det Q}} \frac{\mathrm{lhaf}\left(\mathrm{filldiag}(A_{\mathbf{t}}, \boldsymbol{\gamma}_{\mathbf{t}})\right)}{t_1! \dots t_d!}, \tag{A.41}$$

where $\boldsymbol{\gamma} = \boldsymbol{\alpha}^\dagger \Sigma^{-1}$ and filldiag puts the repeated vector $\boldsymbol{\gamma}_{\mathbf{t}}$ into the diagonals of $A_{\mathbf{t}}$. Moreover, the lhaf is the loop hafnian of a matrix $A \in \mathbb{C}^{n\times n}$ is defined by [60]

$$\mathrm{lhaf}(A) = \sum_{M \in \mathrm{SPM}(n)} \prod_{(i,j)\in M} A_{i,j}, \tag{A.42}$$

where $\mathrm{SPM}(n)$ is the set of single-pair matchings of n elements, which is analoguous to $\mathrm{PMP}(n)$, but loops are also permitted. Particle number measurement using Gaussian states is also called Gaussian Boson Sampling.

When the state is an occupation number state with $\mathbf{s} = (s_1, \dots, s_d)$ initial particles, the probability of resulting in $\mathbf{t} = (t_1, \dots, t_d)$ particles after unitary circuit is [25]

$$p(\mathbf{s}, \mathbf{t}) = \frac{|\,\mathrm{per}\,(U_{\mathbf{s},\mathbf{t}})\,|^2}{t_1! \dots t_d! s_1! \dots s_d!}, \tag{A.43}$$

where $U_{\mathbf{s},\mathbf{t}}$ represents the set of unitary gates U corresponding to the circuit, and the permanent per of a matrix is defined via

$$\mathrm{per}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{\sigma(i),i}, \tag{A.44}$$

where $A \in \mathbb{C}^{n\times n}$. Particle number measurements can be implemented using `pq.ParticleNumberMeasurement` in Piquasso.

A.4.3 Threshold measurement

Threshold measurement or threshold detection is very similar to particle number measurement, but it only results in samples containing 0 or 1, where 0 corresponds to no photon being detected, and 1 corresponds to the detection of at least one photon.

Note that threshold measurement is only supported in the Gaussian simulator. When the state is a non-displaced Gaussian state, the probability of an output $\mathbf{t} = (t_1, \dots, t_d) \in \{0,1\}^d$ during threshold detection is given by [44]

$$p(\mathbf{t}) = \frac{1}{\sqrt{\det Q}} \frac{\mathrm{tor}\,(O_{\mathbf{t}})}{t_1! \dots t_d!}, \tag{A.45}$$

where

$$O = \mathbb{1} - Q^{-1}, \tag{A.46}$$

Q is given by Eq. (A.39) and the torontonian tor is defined as

$$\mathrm{tor}(A) = \sum_{\mathbf{z} \in P_n} \frac{(-1)^{n/2-|\mathbf{z}|}}{\sqrt{|\det(\mathbb{1} - A_{\mathbf{z}})|}}, \tag{A.47}$$

where P_n is the power set of $1, 2, \ldots, n/2$. When the Gaussian state is displaced with complex displacement vector $\boldsymbol{\alpha}$, the probability distribution is modified as [46]

$$p(\mathbf{t}) = \frac{\exp\left(-\frac{1}{2}\boldsymbol{\alpha}^\dagger \Sigma^{-1} \boldsymbol{\alpha}\right)}{\sqrt{\det Q}} \operatorname{ltor}(O_{\mathbf{t}}, \boldsymbol{\gamma}_{\mathbf{t}}), \tag{A.48}$$

where $\boldsymbol{\gamma} = (\Sigma^{-1}\boldsymbol{\alpha})^*$ and ltor is the loop torontonian defined by

$$\operatorname{ltor}(A, \boldsymbol{\gamma}) = \sum_{\mathbf{z} \in P_n} \frac{(-1)^{n/2-|\mathbf{z}|}}{\sqrt{|\det(\mathbb{1} - A_{\mathbf{z}})|}} \exp\left(\frac{1}{2}\boldsymbol{\gamma}^T (\mathbb{1} - A_{\mathbf{z}})^{-1} \boldsymbol{\gamma}^*\right). \tag{A.49}$$

Threshold measurements can be implemented using `pq.ThresholdMeasurement` in Piquasso.

B Code snippets

The code examples in this Appendix are written for Piquasso version 5.0.0.

```
import numpy as np
import piquasso as pq

with pq.Program() as program:
    pq.Q(all) | pq.StateVector((1, 1, 1)) / np.sqrt(2)
    pq.Q(all) | pq.StateVector((0, 1, 1)) / np.sqrt(2)

    pq.Q(1,2) | pq.CrossKerr(xi=np.pi / 4)

    pq.Q(all) | pq.ParticleNumberMeasurement()

config = pq.Config(cutoff=4)
simulator = pq.PureFockSimulator(d=3, config=config)
result = simulator.execute(program, shots=1000)
print(result.samples)  # [(0, 1, 1), (0, 1, 1), ...
```

Code snippet 2: Example usage of `PureFockSimulator`, where the initial state is specified as $\frac{1}{\sqrt{2}}(|111\rangle + |011\rangle)$. For simulations where the states are represented in the Fock space, a Fock space *cutoff* needs to be specified in a `pq.Config` instance, which is then passed to `PureFockSimulator` as a constructor parameter. The `FockSimulator` can be parametrized similarly.

```python
import numpy as np
import piquasso as pq

simulator = pq.GaussianSimulator(d=5)

with pq.Program() as program:
    pq.Q(all) | pq.Vacuum()

    for i in range(5):
        pq.Q(i) | pq.Squeezing(r=0.1) | pq.Displacement(r=1.0)

    pq.Q(0,1) | pq.Beamsplitter(theta=np.pi / 3)
    pq.Q(2,3) | pq.Beamsplitter(theta=np.pi / 4)
    pq.Q(3,4) | pq.Beamsplitter(theta=np.pi / 5)

    pq.Q(all) | pq.ParticleNumberMeasurement()

result = simulator.execute(program, shots=1000)
print(result.samples)
# [(0, 1, 1, 2, 2), (1, 3, 0, 0, 1), (0, 1, 0, 0, 3),...
```

Code snippet 3: Example usage of `GaussianSimulator`. This simulator enables fast simulation of Gaussian states, which are states that can be constructed via linear optical gates from the vacuum (or from single-mode thermal states in the mixed case). In this example, these linear gates are the squeezing, the displacement, and the beamsplitter gates. The program definition is concluded with a photon detection on all modes. Finally, the program is executed via the simulator with 100 shots.

```python
import numpy as np
import piquasso as pq

with pq.Program() as program:
    pq.Q(all) | pq.StateVector((1, 2, 0, 2, 1))

    pq.Q(1,2) | pq.Beamsplitter(theta=np.pi / 3)
    pq.Q(2,3) | pq.Beamsplitter(theta=np.pi / 4)
    pq.Q(0,1) | pq.Beamsplitter(theta=np.pi / 5)
    pq.Q(3,4) | pq.Beamsplitter(theta=np.pi / 6)

    pq.Q(all) | pq.ParticleNumberMeasurement()

simulator = pq.SamplingSimulator(d=5)
result = simulator.execute(program, shots=1000)
print(result.samples)  # [(0, 1, 2, 1, 2), (2, 0, 3, 0, 1), ...
```

Code snippet 4: Example usage of `SamplingSimulator`, which is specifically tailored for the Boson Sampling algorithm. Compared to `PureFockSimulator`, this simulator is restricted in a way that only `StateVector` can be used as preparation, only `ParticleNumberMeasurement` as measurement, and only passive linear elements as gates.

```python
import numpy as np
import piquasso as pq
import tensorflow as tf

cutoff = 7  # Fock space truncation
d = 1  # Number of modes

target_state_vector = tf.constant(
    [0., 1./np.sqrt(2), 1./np.sqrt(2), 0., 0., 0., 0.],
    tf.float64,
)

# The neural network weights, initialized with 0
weights = tf.Variable([0., 0., 0., 0.], dtype=tf.float64)

with tf.GradientTape() as tape:
    # Program definition with single CV neural
    # network layer
    with pq.Program() as program:
        pq.Q() | pq.Vacuum()
        pq.Q() | pq.Squeezing(weights[0])
        pq.Q() | pq.Phaseshifter(weights[1])
        pq.Q() | pq.Displacement(weights[2])
        pq.Q() | pq.Kerr(weights[3])

    # Enabling automatic differentiation via `TensorflowConnector`
    simulator = pq.PureFockSimulator(
        d=d,
        config=pq.Config(cutoff=cutoff),
        connector=pq.TensorflowConnector()
    )

    # Simulating `program` using `simulator`
    state = simulator.execute(program).state
    state_vector = state.state_vector

    # Calculating the cost function `J`
    J = tf.math.sqrt(tf.reduce_sum(tf.abs(target_state_vector - state_vector)**2))

# Automatically differentiating the cost
# function by the CV neural network weights
gradients = tape.gradient(J, weights)
```

Code snippet 5: The definition of a single CVQNN layer using Piquasso on a single mode. By running the program using the `simulator` inside a `GradientTape` context, one can extract a quantity from the circuit defined in `program` which can be automatically differentiated.

```python
import numpy as np
import piquasso as pq

with pq.Program() as program:
    pq.Q(all) | pq.StateVector([1, 2, 0, 2, 1])

    pq.Q(1,2) | pq.Beamsplitter(theta=np.pi / 3)
    pq.Q(2,3) | pq.Beamsplitter(theta=np.pi / 4)
    pq.Q(0,1) | pq.Beamsplitter(theta=np.pi / 5)
    pq.Q(3,4) | pq.Beamsplitter(theta=np.pi / 6)

    for i in range(5):
        pq.Q(i) | pq.Loss(transmissivity=0.95)

    pq.Q(all) | pq.ParticleNumberMeasurement()

simulator = pq.SamplingSimulator(d=5)
result = simulator.execute(program, shots=1000)
print(result.samples)  # [(0, 1, 2, 1, 2), (2, 0, 3, 0, 1), ...
```

Code snippet 6: Example usage of `SamplingSimulator` using losses with 95% transmissivity modeled by the `Loss` channel.

```python
import scipy
import numpy as np
import piquasso as pq
import tensorflow as tf


@tf.function
def calculate_cost(psi_target, w, cutoff):
    d = pq.cvqnn.get_number_of_modes(w.shape[1])

    simulator = pq.PureFockSimulator(
        d, pq.Config(cutoff=cutoff),
        connector=pq.TensorflowConnector(decorate_with=tf.function),
    )

    with tf.GradientTape() as tape:
        cvqnn_layers = pq.cvqnn.create_layers(w)

        program = pq.Program(instructions=[pq.Vacuum()] + cvqnn_layers.instructions)

        simulator.execute(program)

        psi = simulator.execute(program).state.state_vector

        cost = tf.math.sqrt(tf.math.reduce_sum(tf.math.abs(psi - psi_target)**2))

    return cost, tape.gradient(cost, w)

number_of_layers = 3
d = 4
cutoff = 7

w = tf.Variable(pq.cvqnn.generate_random_cvqnn_weights(number_of_layers, d))

# Size of the state vector in the truncated Fock space
state_vector_size = scipy.special.comb(d + cutoff - 1, cutoff - 1, exact=True)

psi_target = (
    np.random.rand(state_vector_size)
    + 1j * np.random.rand(state_vector_size)
)

# Normalization
psi_target /= np.sum(np.abs(psi_target))

cost, cost_grad = calculate_cost(psi_target, w, cutoff)
```

Code snippet 7: Basic example of using Piquasso `PureFockSimulator` with Tensorflow `tf.function`. The example uses the `pq.cvqnn` module, which implements CVQNN layers. The `calculate_cost` function calculates the cost described by Eq. (4) using a randomly generated target state vector. Note, that `TensorflowConnector` is supplied with the constructor argument `decorate_with=tf.function` in order to decrease the compilation time of the tracing step.

```
class CustomSimulator(pq.Simulator):
    _state_class: Type[State] = CustomState

    _config_class: Type[Config] = CustomConfig

    _connector_class: Type[BaseConnector] = CustomConnector

    _instruction_map: Dict[Type[Instruction], Callable] = {
        CustomPreparation: calculate_custom_preparation,
        CustomGate: calculate_custom_gate,
        CustomMeasurement: calculate_custom_measurement,
    }
```

Code snippet 8: Defining a custom simulator class with customized functions for simulation. The `_state_class` attribute serves to specify a custom state class that overrides `State`, and it is strictly required to specify one. In addition, one can optionally specify a `Config` class with the `_config_class` class attribute. Some calculations (like hafnian calculation) can be replaced by overriding, e.g., `NumpyConnector` and setting the `_connector_class` attribute. Finally, one can specify calculation functions for custom instructions to be executed with the `_instruction_map` attribute. This code does not serve as a working example, only as a schematic guide.

```
import piquasso as pq

def custom_loop_hafnian_function(matrix, reduce_on):
    """Custom loop hafnian implementation goes here"""

# Creating a Connector with the newly created function for computing loop_hafnians
class CustomConnector(pq.NumpyConnector):
    def loop_hafnian(self, matrix, reduce_on):
        return custom_loop_hafnian_function(matrix, reduce_on)

# Creating the custom simulator instance
simulator = pq.GaussianSimulator(d=5, connector=CustomConnector())
```

Code snippet 9: Specifying the custom loop hafnian function that is injected into Piquasso. During simulations, the newly created `CustomConnector.loop_hafnian` function will be executed instead of the builtin loop hafnian function.

```
import piquasso as pq
import piquassoboost as pqb
from scipy.stats import unitary_group

d = 20
interferometer_matrix = unitary_group.rvs(d)

# Regular Piquasso program
with pq.Program() as program:
    # Apply squeezing gates to all modes
    for i in range(d):
        pq.Q(i) | pq.Squeezing(r=0.9)

    # Apply a random interferometer
    pq.Q(all) | pq.Interferometer(interferometer_matrix)

    # Measure on all modes
    pq.Q(all) | pq.ThresholdMeasurement()

# PiquassoBoost simulator
boosted_simulator = pqb.BoostedGaussianSimulator(d)
result = boosted_simulator.execute(program)
```

Code snippet 10: A schematic usage example of the PiquassoBoost plugin.