Strawberry Fields:
A Software Platform for Photonic Quantum Computing

Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook

Xanadu, 372 Richmond St W, Toronto, M5V 1X6, Canada

We introduce Strawberry Fields, an open-source quantum programming architecture for light-based quantum computers, and detail its key features. Built in Python, Strawberry Fields is a full-stack library for design, simulation, optimization, and quantum machine learning of continuous-variable circuits. The platform consists of three main components: (i) an API for quantum programming based on an easy-to-use language named Blackbird; (ii) a suite of three virtual quantum computer backends, built in NumPy and TensorFlow, each targeting specialized uses; and (iii) an engine which can compile Blackbird programs on various backends, including the three built-in simulators, and – in the near future – photonic quantum information processors. The library also contains examples of several paradigmatic algorithms, including teleportation, (Gaussian) boson sampling, instantaneous quantum polynomial, Hamiltonian simulation, and variational quantum circuit optimization.

Introduction

The decades-long worldwide quest to build practical quantum computers is currently undergoing a critical period. During the next few years, a number of different quantum devices will become available to the public. While fault-tolerant quantum computers will one day provide significant computational speedups for problems like factoring [1], search [2], or linear algebra [3], near-term quantum devices will be noisy, approximate, and sub-universal [4]. Nevertheless, these emerging quantum processors are expected to be strong enough to show a computational advantage over classical computers for certain problems, an achievement known as quantum computational supremacy.

As we approach this milestone, work is already underway exploring how such quantum processors might best be leveraged. Popular techniques include variational quantum eigensolvers [5, 6], quantum approximate optimization algorithms [7, 8], sampling from computationally hard probability distributions [9–17], and quantum annealing [18, 19]. Notably, these methodologies can be applied to tackle important practical problems in chemistry [20–23], finance [24], optimization [25, 26], and machine learning [27–34]. These known applications are very promising, yet it is perhaps the *unknown* future applications of quantum computers that are most tantalizing. We may not know the best applications of quantum computers until these devices become available more widely to researchers, students, entrepreneurs, and programmers worldwide.

To this end, a nascent quantum software ecosystem has recently begun to develop [35–47]. However, a prevailing theme for these software efforts is to target qubit-based quantum devices. In reality, there are several competing models of quantum computing which are equivalent in computational terms, but which are conceptually quite distinct. One prominent approach is the continuous variable (CV) model of quantum computing [48–50]. In the CV model, the basic information-processing unit is an infinite-dimensional bosonic mode, making it particularly well-suited for implementations and applications based on light. The CV model retains the computational power of the qubit model (cf. Chap. 4 of Ref. [51]), while offering a number of unique features. For instance, the CV model is a natural fit for simulating bosonic systems (electromagnetic fields, trapped atoms, harmonic oscillators, Bose-Einstein condensates, phonons, or optomechanical resonators) and for settings where continuous quantum operators – such as position & momentum – are present. Even in classical computing, recent advances from deep learning have demonstrated the power and flexibility of a continuous representation of computation [52, 53] in comparison to the discrete computational model which has historically dominated.

Here we introduce Strawberry Fields[1], an open-source software architecture for photonic quantum computing. Strawberry Fields is a full-stack quantum software platform, implemented in Python, specifically targeted to the CV model. Its main element is a new quantum programming language named Blackbird. To lay the groundwork for future photonic quantum computing hardware, Straw-

[1] This document refers to Strawberry Fields version 0.9. Full documentation is available online at strawberryfields.readthedocs.io.

berry Fields also includes a suite of three CV quantum simulator backends implemented using NumPy [54] and TensorFlow [55]. Strawberry Fields comes with a built-in engine to convert Blackbird programs to run on any of the simulator backends or, when they are available, on photonic quantum computers. To accompany the library, an online service for interactive exploration and simulation of CV circuits is available at strawberryfields.ai.

Aside from being the first quantum software framework to support photonic quantum computation with continuous-variables, Strawberry Fields provides additional computational features not presently available in the quantum software ecosystem:

1. We provide two numeric simulators; a Gaussian backend, and a Fock-basis backend. These two formulations are unique to the CV model of quantum computation due to the use of an infinite Hilbert space, and came with their own technical challenges.

   (a) The Gaussian backend provides state-of-the-art methods and functions for calculating the fidelity and Fock state probabilities, involving calculations of the classically intractable hafnian [56].

   (b) The Fock backend allows operations such as squeezing and beamsplitters to be performed in the Fock-basis, a computationally intensive calculation that has been highly vectorized and benchmarked for performance.

2. We provide a suite of important circuit decompositions appearing in quantum photonics – such as the Williamson, Bloch-Messiah, and Clements decompositions.

3. The Fock-basis backend written using the TensorFlow machine learning library allows for symbolic calculations, automatic differentiation, and backpropagation through CV quantum simulations. As far as we are aware, this is the first quantum simulation library written using a high-level machine learning library, with support for dataflow programming and automatic differentiation.

The remainder of this white paper is structured as follows. Before presenting Strawberry Fields, we first provide a brief overview of the key ingredients for CV quantum computation, specifically the most important states, gates, and measurements. We then introduce the Strawberry Fields architecture in full, presenting the Blackbird quantum assembly language, outlining how to use the library for numerical simulation, optimization, and quantum machine learning. Finally, we discuss the three built-in simulators and the internal representations that they employ. In the Appendices, we give further mathematical and software details and provide full example code for a number of important CV quantum computing tasks.

Quantum Computation with Continuous Variables

Many physical systems in nature are intrinsically continuous, with light being the prototypical example. Such systems reside in an infinite-dimensional Hilbert space, offering a paradigm for quantum computation which is distinct from the discrete qubit model. This *continuous-variable model* takes its name from the fact that the quantum operators underlying the model have continuous spectra. It is possible to embed qubit-based computations into the CV picture [57], so the CV model is as powerful as its qubit counterparts.

From Qubits to Qumodes

A high-level comparison of CV quantum computation with the qubit model is depicted in Table I. In the remainder of this section, we will provide a basic presentation of the key elements of the CV model. A more detailed technical overview can be found in Appendix A. Readers experienced with CV quantum computing can safely skip to the next section.

	CV	**Qubit**
Basic element	Qumodes	Qubits
Relevant operators	Quadratures $\hat{x}, \hat{p}$ Mode operators $\hat{a}, \hat{a}^\dagger$	Pauli operators $\hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z$
Common states	Coherent states $\|\alpha\rangle$ Squeezed states $\|z\rangle$ Number states $\|n\rangle$	Pauli eigenstates $\|0/1\rangle, \|\pm\rangle, \|\pm i\rangle$
Common gates	Rotation, Displacement, Squeezing, Beamsplitter, Cubic Phase	Phase shift, Hadamard, CNOT, T-Gate
Common measurements	Homodyne $\|x_\phi\rangle\langle x_\phi\|$, Heterodyne $\frac{1}{\pi}\|\alpha\rangle\langle\alpha\|$, Photon-counting $\|n\rangle\langle n\|$	Pauli eigenstates $\|0/1\rangle\langle 0/1\|, \|\pm\rangle\langle\pm\|,$ $\|\pm i\rangle\langle\pm i\|$

Table I: *Basic comparison of the CV and qubit settings.*

The most elementary CV system is the bosonic harmonic oscillator, defined via the canonical mode operators $\hat{a}$ and $\hat{a}^\dagger$. These satisfy the well-known commutation relation $[\hat{a}, \hat{a}^\dagger] = \mathbb{I}$. It is also common to work with the *quadrature operators* (also called the position & momentum oper-

ators)[2],

$$\hat{x} := \sqrt{\frac{\hbar}{2}}(\hat{a} + \hat{a}^\dagger), \tag{1}$$

$$\hat{p} := -i\sqrt{\frac{\hbar}{2}}(\hat{a} - \hat{a}^\dagger), \tag{2}$$

where $[\hat{x}, \hat{p}] = i\hbar\hat{\mathbb{I}}$. We can picture a fixed harmonic oscillator mode (say, within an optical fibre or a waveguide on a photonic chip) as a single 'wire' in a quantum circuit. These *qumodes* are the fundamental information-carrying units of CV quantum computers. By combining multiple qumodes – each with corresponding operators $\hat{a}_i$ and $\hat{a}_i^\dagger$ – and interacting them via sequences of suitable quantum gates, we can implement a general CV quantum computation.

CV States

The dichotomy between qubit and CV systems is perhaps most evident in the basis expansions of quantum states:

$$\text{Qubit} \qquad |\phi\rangle = \phi_0 |0\rangle + \phi_1 |1\rangle, \tag{3}$$

$$\text{Qumode} \qquad |\psi\rangle = \int dx\ \psi(x)|x\rangle. \tag{4}$$

For qubits, we use a discrete set of coefficients; for CV systems, we can have a *continuum*. The states $|x\rangle$ are the eigenstates of the $\hat{x}$ quadrature, $\hat{x}|x\rangle = x|x\rangle$, with $x \in \mathbb{R}$. These quadrature states are special cases of a more general family of CV states, the *Gaussian states*, which we now introduce.

Gaussian states

Our starting point is the vacuum state $|0\rangle$. Other states can be created by evolving the vacuum state according to

$$|\psi\rangle = \exp(-itH)|0\rangle, \tag{5}$$

where H is a bosonic Hamiltonian (i.e., a function of the operators $\hat{a}_i$ and $\hat{a}_i^\dagger$) and t is the evolution time. States where the Hamiltonian H is at most quadratic in the operators $\hat{a}_i$ and $\hat{a}_i^\dagger$ (equivalently, in $\hat{x}_i$ and $\hat{p}_i$) are called *Gaussian*. For a single qumode, Gaussian states are parameterized by two continuous complex variables: a displacement parameter $\alpha \in \mathbb{C}$ and a squeezing parameter $z \in \mathbb{C}$ (often expressed as $z = r\exp(i\phi)$, with $r \geq 0$). Gaussian states are so-named because we can identify each Gaussian state with a corresponding Gaussian distribution. For single qumodes, the identification proceeds through its displacement and squeezing parameters. The displacement gives the centre of the distribution, while the squeezing determines the variance and rotation of the distribution (see Fig. 1). Multimode Gaussian states, on the other hand, are parameterized by a vector of displacements $\bar{\mathbf{r}}$ and a covariance matrix $\mathbf{V}$. Many important pure states in the CV model are special cases of the pure Gaussian states; see Table II for a summary.

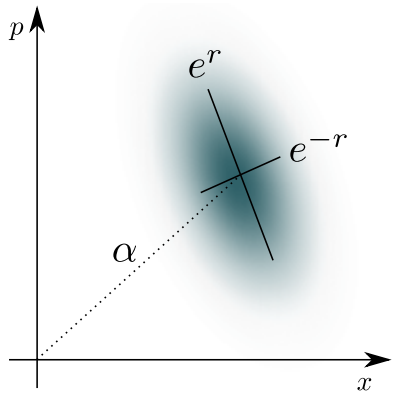

FIG. 1: *Schematic representation of a Gaussian state for a single mode. The shape and orientation are parameterized by the displacement α and squeezing $z = r\exp(i\phi)$.*

State family	Displacement	Squeezing
Vacuum state $\|0\rangle$	$\alpha = 0$	$z = 0$
Coherent states $\|\alpha\rangle$	$\alpha \in \mathbb{C}$	$z = 0$
Squeezed states $\|z\rangle$	$\alpha = 0$	$z \in \mathbb{C}$
Displaced squeezed states $\|\alpha, z\rangle$	$\alpha \in \mathbb{C}$	$z \in \mathbb{C}$
$\hat{x}$ eigenstates $\|x\rangle$	$\alpha \in \mathbb{C}$, $x = 2\sqrt{\frac{\hbar}{2}}\mathrm{Re}(\alpha)$	$\phi = 0$, $r \to \infty$
$\hat{p}$ eigenstates $\|p\rangle$	$\alpha \in \mathbb{C}$, $p = 2\sqrt{\frac{\hbar}{2}}\mathrm{Im}(\alpha)$	$\phi = \pi$, $r \to \infty$
Fock states $\|n\rangle$	N.A.	N.A.

Table II: *Common single-mode pure states and their relation to the displacement and squeezing parameters. All listed families are Gaussian, except for the Fock states. The $n = 0$ Fock state is also the vacuum state.*

Fock states

Complementary to the continuous Gaussian states are the discrete *Fock states* (or *number states*) $|n\rangle$, where n are non-negative integers. These are the eigenstates of the number operator $\hat{n} = \hat{a}^\dagger\hat{a}$. The Fock states form a discrete (countable) basis for qumode systems. Thus, each of the Gaussian states considered in the previous section can be expanded in the Fock-state basis. For example, coherent states have

[2] It is common to picture $\hbar$ as a (dimensionless) scaling parameter for the $\hat{x}$ and $\hat{p}$ operators rather than a physical constant. However, there are several conventions for the scaling value in common use [58]. These self-adjoint operators are proportional to the Hermitian and anti-Hermitian parts of the operator $\hat{a}$. Strawberry Fields allows the user to specify this value, with the default $\hbar = 2$.

the form

$$|\alpha\rangle = \exp\left(-\tfrac{|\alpha|^2}{2}\right)\sum_{n=0}^{\infty}\frac{\alpha^n}{\sqrt{n!}}|n\rangle\,, \tag{6}$$

while (undisplaced) squeezed states only have even number states in their expansion:

$$|z\rangle = \frac{1}{\sqrt{\cosh r}}\sum_{n=0}^{\infty}\frac{\sqrt{(2n)!}}{2^n n!}[-e^{i\phi}\tanh(r)]^n\,|2n\rangle\,. \tag{7}$$

Mixed states

Mixed Gaussian states are also important in the CV picture, for instance, the *thermal state*

$$\rho(\overline{n}) := \sum_{n=0}^{\infty}\frac{\overline{n}^n}{(1+\overline{n})^{n+1}}|n\rangle\langle n|, \tag{8}$$

which is parameterized via the mean photon number $\overline{n} := \mathrm{Tr}(\rho(\overline{n})\hat{n})$. Starting from this state, we can consider a mixed-state-creation process similar to Eq. (5), namely

$$\rho = \exp(-itH)\rho(\overline{n})\exp(itH). \tag{9}$$

Analogously to pure states, by applying Hamiltonians of second-order (or lower) to thermal states, we generate the family of Gaussian mixed states.

CV Gates

Unitary operations can be associated with a generating Hamiltonian H via the recipe (cf. Eqs. (5) & (9))

$$U := \exp(-itH). \tag{10}$$

For convenience, we classify unitaries by the degree of their generator. A CV quantum computer is said to be universal if it can implement, to arbitrary precision and with a finite number of steps, any unitary which is polynomial in the mode operators [48]. We can build a multimode unitary by applying a sequence of gates from a *universal gate set*, each of which acts only on one or two modes. We focus on a universal set made from the following two subsets:

Gaussian gates: Single and two-mode gates which are at most quadratic in the mode operators, e.g., *Displacement, Rotation, Squeezing, and Beamsplitter* gates.

Non-Gaussian gate: A single-mode gate which is degree 3 or higher, e.g., the *Cubic phase* gate.

A number of fundamental CV gates are presented in Table III. Many of the Gaussian states from the previous section are connected to a corresponding Gaussian gate. Any multimode Gaussian gate can be implemented through a suitable combination of Displacement, Rotation, Squeezing, and Beamsplitter Gates [50], making these gates sufficient for constructing all quadratic unitaries. The cubic phase gate is presented as an exemplary non-Gaussian gate, but any other non-Gaussian gate could also be used to achieve universality. A number of other useful CV gates are listed in Appendix B.

Gate	Unitary	Symbol
Displacement	$D_i(\alpha) = \exp(\alpha\hat{a}_i^\dagger - \alpha^*\hat{a}_i)$	D
Rotation	$R_i(\phi) = \exp(i\phi\hat{n}_i)$	R
Squeezing	$S_i(z) = \exp(\frac{1}{2}(z^*\hat{a}_i^2 - z\hat{a}_i^{\dagger 2}))$	S
Beamsplitter	$BS_{ij}(\theta,\phi) = \exp(\theta(e^{i\phi}\hat{a}_i\hat{a}_j^\dagger - e^{-i\phi}\hat{a}_i^\dagger\hat{a}_j))$	BS
Cubic phase	$V_i(\gamma) = \exp\left(i\frac{\gamma}{3\hbar}\hat{x}_i^3\right)$	V

Table III: *Some important CV model gates. All listed gates except the cubic phase gate are Gaussian.*

CV Measurements

As with CV states and gates, we can distinguish between Gaussian and non-Gaussian measurements. The Gaussian class consists of two (continuous) types: homodyne and heterodyne measurements, while the key non-Gaussian measurement is photon counting. These are summarized in Table IV.

Homodyne measurements

Ideal homodyne detection is a projective measurement onto the eigenstates of the quadrature operator $\hat{x}$. These states form a continuum, so homodyne measurements are inherently continuous, returning values $x \in \mathbb{R}$. More generally, we can consider projective measurement onto the eigenstates $|x_\phi\rangle$ of the Hermitian operator

$$\hat{x}_\phi := \cos\phi\ \hat{x} + \sin\phi\ \hat{p}. \tag{11}$$

This is equivalent to rotating the state clockwise by ϕ and performing an $\hat{x}$-homodyne measurement. If we have a multimode Gaussian state and we perform homodyne measurement on one of the modes, the conditional state of the unmeasured modes remains Gaussian.

Heterodyne measurements

Whereas homodyne detection is a measurement of $\hat{x}$, heterodyne detection can be seen as a simultaneous measurement of both $\hat{x}$ and $\hat{p}$. Because these operators do not commute, they cannot be simultaneously measured without some degree of uncertainty. Equivalently, we can picture heterodyne measurement as projection onto the coherent states, with measurement operators $\frac{1}{\pi}|\alpha\rangle\langle\alpha|$. Because the coherent states are not orthogonal, there is a corresponding lack of sharpness in the measurements. If we perform heterodyne measurement on one mode of a multimode state, the conditional state on the remaining modes stays Gaussian.

Measurement	Measurement Operators	Measurement values
Homodyne	$\|x_\phi\rangle\langle x_\phi\|$	$x \in \mathbb{R}$
Heterodyne	$\frac{1}{\pi}\|\alpha\rangle\langle\alpha\|$	$\alpha \in \mathbb{C}$
Photon counting	$\|n\rangle\langle n\|$	$n \in \mathbb{N}$

Table IV: *Key measurement types for the CV model. The '-dyne' measurements are Gaussian, while photon-counting is non-Gaussian.*

Photon Counting

Photon counting (also known as as *photon-number resolving measurement*), is a complementary measurement method to the '-dyne' measurements, revealing the particle-like, rather than the wave-like, nature of qumodes. This measurement projects onto the number eigenstates $|n\rangle$, returning non-negative integer values $n \in \mathbb{N}$. Except for the outcome $n = 0$, a photon-counting measurement on a single mode of a multimode Gaussian state will cause the remaining modes to become non-Gaussian. Thus, photon-counting can be used as an ingredient for implementing non-Gaussian gates. A related process is *photodetection*, where a detector only resolves the vacuum state from non-vacuum states. This process has only two measurement operators, namely $|0\rangle\langle 0|$ and $\mathbb{I} - |0\rangle\langle 0|$.

The Strawberry Fields Software Platform

The Strawberry Fields library has been designed with several key goals in mind. Foremost, it is a standard-bearer for the CV model, laying the groundwork for future photonic quantum computers. As well, Strawberry Fields is designed to be simple to use, giving entry points for as many users as possible. Finally, since the potential applications of near-term quantum computers are still being worked out, it is important that Strawberry Fields provides powerful tools to easily explore many different use-cases and applications.

Strawberry Fields has been implemented in Python, a modern language with a gentle learning curve which is already familiar to many programmers and scientific practitioners. The accompanying quantum simulator backends are built upon the widely used Python packages NumPy and TensorFlow. All Strawberry Fields code is open source. Strawberry Fields can be accessed programmatically as a Python package, or via a browser-based interface for designing quantum circuits.

A pictorial outline of Strawberry Fields' key elements and their interdependencies is presented in Fig. 2. Conceptually, the software stack is separated into two main pieces: a user-facing frontend layer and a lower-level backends component. The frontend encompasses the Strawberry Fields Python API and the Blackbird quantum assembly language. These elements provide access points for users to design quantum circuits. These circuits are then linked to a backend via a quantum compiler engine. For a backend, the engine can currently target one of three included quantum computer simulators. When CV quantum processors become available in the near future, the engine will also build and run circuits on those devices. Further, high-level quantum computing applications can be built by leverging the Strawberry Fields frontend API. Existing examples include the Strawberry Fields Interactive website, the Quantum Machine Learning Toolbox (for streamlining the training of variational quantum circuits), and SFOpenBoson (an interface between the electronic structure library OpenFermion [45] and Strawberry Fields).

In the remainder of this section, the key elements of Strawberry Fields will be presented in more detail. Proceeding through a series of examples, we show how CV quantum computations can be defined using the Blackbird language, then compiled and run on a quantum computer backend. We also outline how to use Strawberry Fields for optimization and machine learning on quantum circuits. Finally, we discuss the suite of quantum computer simulators included within Strawberry Fields.

Blackbird: A Quantum Programming Language

As classical computers have become progressively more powerful, the languages used to program them have also undergone considerable paradigmatic changes. Machine code gave way to human-readable assembly languages, followed by higher-level procedural and object-oriented languages. With each generation, the trend has been towards higher levels of abstraction, separating the programmer more and more from details of the actual computer hardware. Quantum computers are still at an early stage of development, so while we can imagine what higher-level quantum programming might look like, in the near term we first need to build languages which are conceptually closer to the quantum hardware.

Blackbird is a standalone domain specific language (DSL) for continuous-variable quantum computation. With a well-defined grammar in extended Backus-Naur form, and both Python and C++ parsers available, Blackbird provides operations that match the basic CV states, gates, and measurements, and maps directly to low-level hardware instructions. The abstract syntax keeps a close connection between code and the quantum operations that they implement; this syntax is modeled after that of ProjectQ [41], but specialized to the CV setting. Blackbird can be used as part of the Strawberry Fields stack, but also directly with photonic quantum computing hardware systems.

Within the Strawberry Fields framework, we have built an implementation of Blackbird using Python 3 as the embedding language — an 'embedded' DSL. This 'Python-enhanced' Blackbird language provides the same core operations and follows the same grammar and syntactical rules as the standalone DSL, but, by nature, may also contain valid Python constructs. Furthermore, Strawberry Fields' 'Python-enhanced' Blackbird provides users with additional

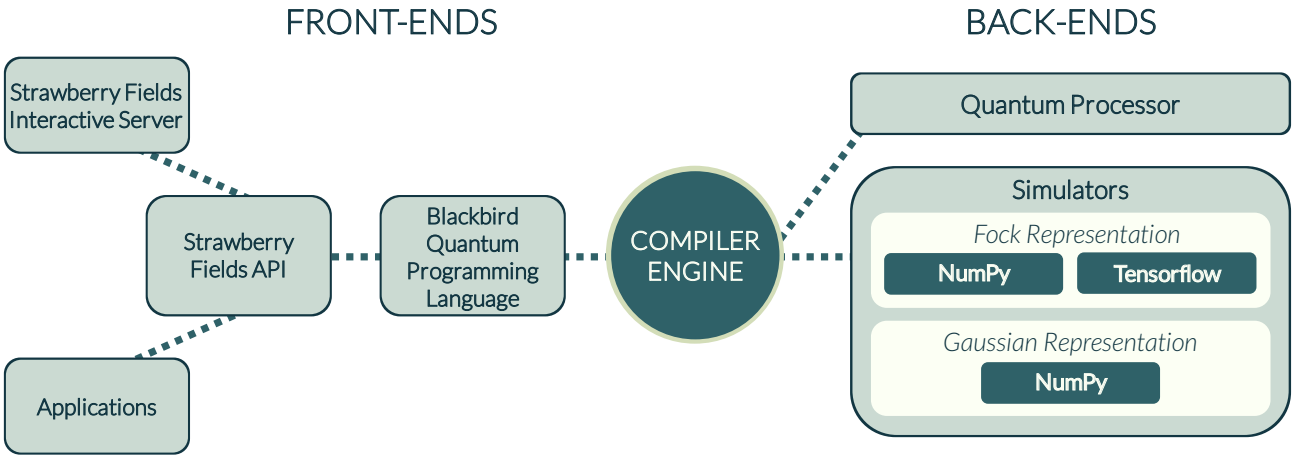

FIG. 2: *Outline of the Strawberry Fields software stack. The Strawberry Fields Interactive server is available online at strawberryfields.ai.*

quantum operations that are decomposed into lower-level Blackbird assembly commands. We will introduce the elements of Blackbird through a series of basic examples, discussing more technical aspects as they arise.

Operations

Quantum computations consist of four main ingredients: state preparations, application of gates, performing measurements, and adding/removing subsystems. In Blackbird, these are all considered as *Operations*, and share the same basic syntax. In the following code examples, we use the variable `q` for a set of qumodes (more specifically, a *quantum register*), the details of which are deferred until the next section.

Our first considered Operation is state preparation. By default, qumodes are initialized in the vacuum state. Various other important CV states can be created with simple Blackbird commands.

```
# Create the vacuum state in qumode 0
Vac | q[0]

# Create a coherent state in qumode 1
alpha = 2.0 + 1j
Coherent(alpha) | q[1]

# Create squeezed states in qumodes 0 & 1
S = Squeezed(2.0)
S | q[0]
S | q[1]

# Create a Fock state in qumode 1
Fock(4) | q[1]
```

Codeblock 1: *Blackbird code for creating various CV quantum states.*

Blackbird state preparations such as those used in Codeblock 1 implicitly reset the existing state of the qumodes. Conceptually, the vertical bar symbol '|' separates Operations – like state preparation – from the registers that they act upon. Notice that we can use Operations inline, or construct them separately and reuse them several times.

After creating states, we will want to transform these using quantum gates.

```
# Apply the Displacement gate to qumode 0
alpha = 2.0 + 1j
Dgate(alpha) | q[0]

# Apply the Rotation gate
phi = 1.157
Rgate(phi) | q[0]

# Apply the Squeezing gate
Sgate(2.0, 0.17) | q[0]

# Apply the Beamsplitter gate to qumodes 0 & 1
BSgate(0.314, 0.223) | (q[0], q[1])

# Apply Cubic phase gate (VGate) to qumode 0
gamma = 0.1
Vgate(gamma) | q[0]

# Apply Hermitian conjugate of a gate
V = Vgate(gamma)
V.H | q[0]
```

Codeblock 2: *Blackbird code for applying various CV gates.*

Blackbird supports all of the gates listed in the previous section as well as a number of composite gates, each of which can be decomposed using the universal gates. The supported composite gates are: controlled X (`CXgate`), controlled Z (`CZgate`), quadratic phase (`Pgate`), and two-mode squeezing (`S2gate`). A full list of gates currently supported in Blackbird can be found in Appendix B.

Finally, we can specify measurement Operations using Blackbird.

```
# Homodyne measurement at angle phi
phi = 0.785
MeasureHomodyne(phi) | q[0]

# Special homodyne measurements
MeasureX | q[0]
MeasureP | q[1]

# Heterodyne measurement
MeasureHeterodyne() | q[0]
MeasureHD           | q[1]  # shorthand

# Number state measurements of various qumodes
MeasureFock() | q[0]
MeasureFock() | (q[1], q[2]) # multiple modes
Measure       | q[3] # shorthand
```

Codeblock 3: *Blackbird code for carrying out CV measurements.*

Measurements have several effects. For one, the numerical result of the measurement is placed in a classical register. As well, the state of all remaining qumodes is projected to the (normalized) conditional state for that measurement value. Finally, the state of the measured qumode is reset to the vacuum state. This is the typical behaviour of photonic hardware, where measurements absorb all the energy of the measured qumode.

Running Blackbird Programs in Python

Within Python, Blackbird programs are managed by an *Engine*. The function `Engine` in the Strawberry Fields API will instantiate an Engine, returning both the Engine and its corresponding quantum register. The Engine is used as a Python context manager, providing a convenient way to encapsulate Blackbird programs.

```
# Create Engine and quantum register
import strawberryfields as sf
eng, q = sf.Engine(num_subsystems=2)

# The register is also available via
 ↪  eng.register
assert q == eng.register

# Put Blackbird Operations in namespace
from strawberryfields.ops import *

# Declare a Blackbird program
from math import pi
z = 4.
S = Sgate(z)
B = BSgate(pi / 4, 0)
with eng:
  S   | q[0]
  S.H | q[1]
  B   | q
  MeasureP | q[0]
  MeasureP | q[1]

# Execute Blackbird program and extract values
eng.run(backend="gaussian")
vals = [reg.val for reg in q]
```

Codeblock 4: *Code for declaring and running Blackbird programs using the Strawberry Fields library.*

The above code example is runnable and carries out a complete quantum computation, namely the preparation and measurement of an EPR entangled state. Note that Operations can be declared outside of the Engine, but their action on the quantum registers must come within the Engine context. Also notice that our register has a length of 2, so any single-mode Operations must act on specific elements, i.e., `q[i]`, while two-mode Operations can act on `q` directly. Finally, the user must specify a backend – as well as any backend-dependent settings – when calling `eng.run()`. We will discuss the Strawberry Fields backends further in a later section.

Quantum and Classical Registers

When a Strawberry Fields Engine is constructed, the user must specify the number of qumode subsystems to begin with. This number is required for the initialization of the Engine, but may change within a computation (e.g., when temporary ancilla modes are used). Qumodes can be added/deleted by using the `New` and `Del` Operations.

```
# A Blackbird circuit where gates
# are added and deleted
alice = q[0]
with eng:
  Sgate(1)    | alice
  bob, charlie = New(2)
  BSgate(0.5) | (alice, bob)
  CXgate(1)   | (alice, charlie)
  Del         | alice
  S2gate(0.4) | (charlie, bob)

# Attempting to act on registers which have
# been removed will raise an IndexError
try:
  with eng:
    Dgate(0.1) | alice
except Exception as e:
  assert isinstance(e, IndexError)
```

Codeblock 5: *Adding and deleting qumode subsystems.*

An Engine maintains a unique numeric indexing for the quantum registers based on the order they were added. When a subsystem is deleted from the circuit, no further gates can act on that register.

As stated earlier, measurement Operations produce classical information (the measurement result) and manipulate the corresponding register. Compared to previously released quantum programming frameworks, such as ProjectQ, PyQuil, and Qiskit, Strawberry Fields has been designed so that the notation `q[i]`, while principally denoting the quantum register, may also encapsulate classical information or a classical register. This behaviour is contextual,

and unique to Strawberry Fields. For example, prior to measurement, `q[i]` simply references a quantum register. Once a measurement operation is performed, `q[i]` continues to represent a quantum register — now reset to the vacuum state — as well as storing the numerical value of the measurement, accessible via the attribute `q[i].val`. Note that this numerical value is only available if a computation has been run up to the point of measurement. We may also use a classical measurement result *symbolically* as a parameter in later gates without first running the computation. To do this, we simply pass the measured register (e.g., `q[i]`) explicitly as an argument to the required gate. As before, the Strawberry Fields quantum register object is contextual — when passed as a gate argument, Strawberry Fields implicitly accesses the encapsulated classical register.

```
# Numerical evaluation of a measurement
# result using eng.run()
with eng:
  MeasureX | q[0]
eng.run("gaussian")
val = q[0].val

# Use a measured register symbolically
# in another gate
with eng:
  MeasureX    | q[0]
  Dgate(q[0]) | q[1]
eng.run("gaussian")
```

Codeblock 6: *Evaluating measurement results numerically and using them symbolically.*

In quantum algorithms, it is common to process a measurement result classically and use the post-processed value as a parameter for further operations in a circuit. Strawberry Fields provides the `convert` decorator to transform a user-specified numerical function into one which acts on registers.

```
@sf.convert
def neg(x):
  return -x

# A Blackbird computation using classical
# data processing
with eng:
  MeasureX         | q[0]
  Dgate(neg(q[0])) | q[1]
eng.run("gaussian")
```

Codeblock 7: *Symbolically processing a measured value before using it.*

Post-selection

The measurement Operations in Strawberry Fields are stochastic in nature, with outcomes determined by some underlying quantum probability distribution. Often it is convenient to select specific values for these measurements rather than sampling them. For instance, we might want to explore the conditional state created by a specific value, determine the measurement-dependent corrections we need to make in a teleportation circuit, or even design an algorithm which inherently contains post-selection. This functionality is supported in Strawberry Fields through the optional keyword argument `select`, which can be supplied for any measurement Operation. The measurement outcome will then return exactly this value, while the remaining modes will be projected into the conditional state corresponding to this value[3].

```
with eng:
  Fock(3) | q[0]
  Fock(2) | q[1]
  BSgate() | (q[0], q[1])
  MeasureFock(select=4) | q[0]
  MeasureFock() | q[1]
eng.run("fock", cutoff_dim=6)
assert q[0].val == 4
```

Codeblock 8: *Selecting a specific desired measurement outcome.*

Decompositions

In addition to the core CV operations discussed above, Strawberry Fields also provides support for some important decompositions frequently used in quantum optics. These include the (a) Williamson decomposition [59], for decomposing arbitrary Gaussian states to a symplectic transformation acting on a thermals state, (b) the Bloch-Messiah decomposition [60–62], for decomposing the action of symplectic transformations to interferometers and single-mode squeezing, and (c) the Clements decomposition [63], for decomposing multi-mode linear interferometers into arrays of beamsplitters and rotations of fixed depth. In all cases, the resulting decomposition into the universal CV gate set may be viewed via the engine method `eng.print_applied()`. Strawberry Fields thus provides a natural environment for embedding graphs and matrices in quantum optical circuits, and viewing the resulting physical components.

```
U = np.array([[1-1j, np.sqrt(2)],
              [-np.sqrt(2), 1+1j]])/2

eng, q = sf.Engine(2)

with eng:
  Squeezed(0.43) | q[0]
  Interferometer(U) | (q[0], q[1])

eng.run("gaussian")
eng.print_applied()
```

[3] Users should be careful to avoid post-selection on measurement values which have no probability of occuring given the current circuit state. In this case, the expected behaviour of a backend is not defined.

```
# >> Squeezed(0.43, 0) | (q[0])
# >> Rgate(2.356) | (q[0])
# >> BSgate(0.7854, 0) | (q[0], q[1])
# >> Rgate(-3.142) | (q[0])
# >> Rgate(0.7854) | (q[1])
```

Codeblock 9: *Using the in-built Clements decomposition to decompose a* 2×2 `Interferometer` *into beamsplitters and phase rotations.*

Optimization and Quantum Machine Learning

Strawberry Fields can perform quantum circuit simulations using both numerical and symbolic representations. Numerical computation is the default operating mode and is supported by all three supplied backends. Symbolic computation is enabled only for the TensorFlow backend. In this section, we outline the main TensorFlow functionalities accessible through the Strawberry Fields frontend interface. More details about the corresponding TensorFlow backend are discussed in the next section. TensorFlow [55] models calculations abstractly using a *computational graph*, where individual operations are represented as nodes and their dependencies by directed edges. This viewpoint separates the symbolic representation of a computation from its numerical evaluation, and makes optimization and machine learning more amenable. On top of this, TensorFlow provides a number of advanced functionalities, including automatic gradient computation, GPU utilization, built-in optimization algorithms, and various other machine learning tools. Note that the term 'quantum machine learning' will be used here in a hybrid sense, i.e., applying conventional machine learning methods to quantum systems.

To build a TensorFlow computational graph using Strawberry Fields, we instantiate an Engine, declare a circuit in Blackbird code, then execute `eng.run` on the TensorFlow (`"tf"`) backend. To keep the underlying simulation fully symbolic, the extra argument `eval=False` must be given.

```
# Create Engine and run symbolic computation
import strawberryfields as sf
from strawberryfields.ops import *
import tensorflow as tf

eng, q = sf.Engine(num_subsystems=1)
with eng:
  Dgate(0.5) | q[0]
  MeasureX | q[0]
eng.run("tf",
        cutoff_dim=5,
        eval=False)

# Registers contain symbolic measurements
print(q[0].val)
# >> Tensor("Measure_homodyne/Meas_result:0",
 ↪  shape=(), dtype=float64)
```

Codeblock 10: *Creating a TensorFlow computational graph for a quantum circuit.*

When we do this, any registers measured in the circuit will be populated with unevaluated `Tensor` objects rather than numerical values (without `eval=False`, the TensorFlow backend returns purely numerical results, similar to the other simulators). These Tensors can still be evaluated numerically by running them in a TensorFlow `Session`. In this case, measurement results will be determined stochastically on each evaluation.

```
# Evaluate Tensors numerically
sess = tf.Session()
sess.run(tf.global_variables_initializer())
print(sess.run(q[0].val))
# >> a numerical measurement value
print(sess.run(q[0].val))
# >> a (different) numerical measurement value
```

Codeblock 11: *Numerically evaluating Tensors.*

When specifying a circuit in Blackbird, we can make use of various special symbolic TensorFlow classes, such as `Variable`, `Tensor`, `placeholder`, or `constant`.

```
# Declare circuits using Tensorflow objects
alpha = tf.Variable(0.1)
phi = tf.constant(0.5)
theta_bs = tf.placeholder(tf.float64)
phi_bs = tf.nn.relu(phi)  # a Tensor object

eng, q = sf.Engine(num_subsystems=2)
with eng:
  Coherent(alpha)           | q[0]
  Measure                   | q[0]
  BSgate(theta_bs, phi_bs) | (q[0], q[1])
  MeasureHomodyne(phi)      | q[1]
eng.run("tf",
        cutoff_dim=5,
        eval=False)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
feed_dict = {theta_bs: 0.5, q[0].val: 2}
print(sess.run(q[1].val, feed_dict=feed_dict))
# >> a numerical measurement value
```

Codeblock 12: *Using abstract TensorFlow classes as circuit parameters.*

In the above example, we supplied an additional `feed_dict` argument when evaluating. This is a Python dictionary which specifies the numerical values (typically, coming from a dataset) for every `placeholder` that appears in a circuit. However, as can be seen from the example, it is also possible to substitute desired values for other nodes in the computation, including the values stored in quantum registers. This allows us to easily post-select measurement values and explore the resulting conditional states.

We can also perform post-processing of measurement results when working with TensorFlow objects. In this case, the functions decorated by `convert` should be written using TensorFlow `Tensors`, `Variables`, and operations.

```python
# Use a simple neural network as
# a processing function
@sf.convert
def NN(x):
  weight = tf.Variable(0.5, dtype=tf.float64)
  bias = tf.Variable(0.1, dtype=tf.float64)
  return tf.sigmoid(weight * x + bias)

eng, q = sf.Engine(num_subsystems=2)
with eng:
  MeasureP        | q[0]
  Dgate(NN(q[0])) | q[1]
  MeasureX        | q[1]
eng.run("tf",
        cutoff_dim=5,
        eval=False)
print([r.val for r in q])
# >> [<tf.Tensor
 ↪  'Measure_homodyne_2/Meas_result:0'
 ↪  shape=() dtype=float64>, <tf.Tensor
 ↪  'Measure_homodyne_3/Meas_result:0'
 ↪  shape=() dtype=float64>]

```

Codeblock 13: *Processing a measurement result using a neural network. For compactness, the example uses a width 1 perceptron, but any continuous processing function supported by TensorFlow can be used.*

The TensorFlow backend additionally supports the use of batched processing, allowing for many evaluations of a quantum circuit to potentially be computed in parallel. Scalars are automatically broadcast to the specified batch size. Finally, we can easily run circuit simulations on special-purpose hardware like GPUs or TPUs.

```python
batch_size = 3
eng, q = sf.Engine(num_subsystems=1)

alpha = tf.Variable([0.1] * batch_size)
# scalars are automatically cast to batch size
beta = tf.Variable(0.5)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

with tf.device("/gpu:0"):
  with eng:
    Dgate(alpha) | q[0]
    Dgate(beta)  | q[0]
    MeasureX     | q[0]
  eng.run("tf",
          cutoff_dim=10,
          batch_size=batch_size,
          eval=False)
print(q[0].val)
# >>
 ↪  Tensor("Measure_homodyne_4/Meas_result:0",
 ↪  shape=(3,), dtype=float64,
 ↪  device=/device:GPU:0)
```

Codeblock 14: *Running a batched computation and explicitly placing the computation on a GPU.*

By taking advantage of these additional functionalities of the TensorFlow backend, we can straightforwardly perform optimization and machine learning on quantum circuits in Strawberry Fields [64, 65]. A complete code example for optimization of a quantum circuit is located in Appendix C.

Strawberry Fields' Quantum Simulators

The ultimate goal is for Blackbird programs to be carried out on photonic quantum computers. To lay the groundwork for these emerging devices, Strawberry Fields comes with a suite of three CV quantum computer simulators specially designed for the CV model. These simulators target different use-cases and support different functionality. For example, many important algorithms in the CV formalism involve only Gaussian states, operations, and measurements. We can take advantage of this structure to more efficiently simulate such computations. Other circuits have inherently non-Gaussian elements to them; for these, the Fock basis provides the standard description. These representations are available in Strawberry Fields in the *Gaussian backend* and the *Fock backend*, respectively. The third built-in backend is the *TensorFlow backend*. Also using the Fock representation, this backend is geared primarily towards optimization and machine learning applications.

Most Blackbird operations are supported across all three backends. A small subset, however, are not supported uniformly due to mathematical incompatibility. For example, the Gaussian backend does not generally support non-Gaussian gates or the preparation/measurement of Fock states. Sometimes it is also useful to work with a backend directly. To allow this, Strawberry Fields provides a backend API, giving access to additional methods and properties which are not part of the streamlined frontend API. A standalone backend can be created in Strawberry Fields using `strawberryfields.backend.load_backend(name)` where `name` is one of `"gaussian"`, `"fock"`, or `"tf"` (for comparison, the backend associated to an Engine `eng` is available via `eng.backend`).

Three important backend methods, common to all simulators, are `begin_circuit`, `reset`, and `state`. The first command instantiates the circuit simulation, the second resets the simulation back to an initial vacuum state, clearing all previous operations, and the third returns a class which encapsulates the current quantum state of the simulator. In addition to containing the numerical (or symbolic) state data, state classes also contain a number of useful methods and attributes for further exploring the quantum state, such as `fidelity`, `mean_photon`, or `wigner`. As a convenience for the user, all simulations carried out via `eng.run` will return a state class representing the final circuit state (see Appendix C for examples).

Gaussian Backend

This backend, written in NumPy, uses the symplectic formalism to represent CV systems. At a high level, this repre-

sentation tracks the quantum state of an N-mode quantum system using two Gaussian components: a $2N$-dimensional displacement vector $\bar{\mathbf{r}}$ and a $2N \times 2N$-dimensional covariance matrix $\mathbf{V}$ (a deeper technical overview is located in Appendix A). After we have created a Gaussian state (either via `state = eng.run(backend="gaussian")` or by directly calling the `state` method of a Gaussian backend), we can access $\bar{\mathbf{r}}$ and $\mathbf{V}$ via `state.means` and `state.cov`, respectively. Other useful Gaussian state methods are `displacement` and `squeezing`, which return the Gaussian parameters associated to the underlying state.

The scaling of the symplectic representation with the number of modes is $\mathcal{O}(N^2)$. On one hand, this is quite powerful. It allows us to to efficiently simulate any computations which are fully Gaussian. On the other, the formalism is not expressive enough to simulate more general quantum computations. Only a small number of non-Gaussian methods are available for this backend. These are auxiliary methods where we extract some non-Gaussian information from a Gaussian state, but do not update the state of the circuit. One such method is `fock_prob`, which is implemented using an optimized – yet still exponentially scaling – algorithm. This method enables simulation of the *Gaussian boson sampling* algorithm [10] using the Gaussian backend; see Appendix C for a complete code example.

Fock Backend

This backend, also written in NumPy, uses a fundamentally different description for qumodes than the Gaussian representation. As discussed in the introductory sections, the Fock representation encodes quantum computation in a countably infinite-dimensional Hilbert space. This representation is faithful, allowing a precise description of CV systems, in particular non-Gaussian circuits. Yet simulating infinite-dimensional systems leads to some computational tradeoffs which are not present for qubit simulators. Most importantly, we impose a *cutoff dimension* D for simulations (chosen by the user), so the Fock backend only covers a restricted set of number states $\{|0\rangle, ..., |D-1\rangle\}$ for each mode. The size of simulated quantum systems thus depends on both the number of subsystems N and the cutoff, being $\mathcal{O}(D^N)$. We contrast this with qubit systems, where the base is fixed, i.e., $\mathcal{O}(2^N)$. While these scalings are both exponential, in practice simulating qumode systems for $D > 2$ is more computationally demanding than qubits. This is because the (truncated) qumode subsystems have a higher dimension and thus encode more information than their qubit counterparts.

Physically, imposing a cutoff is a reasonable strategy since higher photon-number states must have higher energy and, in practice, quantum-optical systems will have bounded energy (e.g., limited by the power of a laser). On the other hand, there are certainly states which can be easily prepared in the lab, yet would not fit accurately on the simulator. Thus, some care must be taken to trade off between the numerical cutoff value, the number of modes, and the energy scale of the circuit. If the energy scale is sufficiently low that all states fit within the specified cutoff, then simulations with the Fock and Gaussian backends will be in numerical agreement.

Like the Gaussian backend, the Fock backend has a `state` method which encapsulates the numerical state, while also providing a number of methods and attributes specific to the Fock representation (such as `ket`, `trace`, and `all_fock_probs`.). Unlike the Gaussian representation, mixed state simulations take up more resources than pure states. Pure states are represented in the Fock backend by an D^N-dimensional complex vector and mixed states by a D^{2N}-dimensional density matrix. Because of this extra overhead, by default the Fock backend will internally represent a quantum circuit as long as possible as a pure state, switching to the mixed state representation only when it becomes necessary. Most importantly, for $N > 2$ qumodes, *all state preparation Operations* (`Vacuum`, `Squeezed`, `Fock`, etc.) *cause the representation to become mixed*. This is because the mode where the state is prepared could be entangled with other modes. To keep physically consistent, the Fock backend will first trace out the relevant mode, necessitating a mixed state representation. When possible, *it is recommended to apply gates to the (default) vacuum state in order to efficiently prepare pure states*. If desired, a mixed state simulation can be enforced by passing the argument `pure=False` when calling `begin_circuit`.

TensorFlow Backend

The other built-in backend for Strawberry Fields is coded using TensorFlow. As a simulator, it uses the same internal representation as the Fock backend (Fock basis, finite cutoff, pure vs. mixed state representations, etc.) and has the same methods. It can operate as a numerical simulator similar to the other backends. Its main purpose, however, is to leverage the many powerful tools provided by TensorFlow to enable optimization and machine learning on quantum circuits. Much of this functionality was presented in the previous section, so we will not repeat it here.

Like the other simulators, users can query the TensorFlow backend's `state` method to access the internal representation of a circuit's quantum state. This functions similarly to the Fock backend's `state` method, except that the state returned can be an unevaluated Tensor object when the keyword argument `eval` is set to `False`. This state Tensor can be combined with any supported TensorFlow operations (`norm`, `self_adjoint_eig`, `inv`, etc.) to enable optimization and machine learning on various properties of quantum circuits and quantum states.

Conclusions

We have introduced Strawberry Fields, a multi-faceted software platform for continuous-variable quantum computing. The main components of this library – a custom quantum programming language (Blackbird), a compiler engine, and a suite of quantum simulators targeting distinct

applications – have been presented in detail. Further information is available in both the Appendices and the Strawberry Fields online documentation.

The stage is now set for the broader community to use Strawberry Fields for exploration, research, and development of new quantum algorithms, specialized circuits, and machine learning models. We anticipate the creation of further software applications and backend modules for the Strawberry Fields platform (developed both internally and externally), providing advanced functionality and applications for quantum computing and quantum machine learning.

Acknowledgements

We thank our colleagues at Xanadu for testing Strawberry Fields, reviewing this white paper, and providing helpful feedback. In particular, we thank Patrick Rebentrost for valuable discussions and suggestions.

Appendix A: The CV model

In this Appendix, we provide more technical and mathematical details about the CV model, the quantum computing paradigm underlying Strawberry Fields.

Universal Gates for CV Quantum Computing

In discrete qubit systems, the notion of a *universal gate set* has the following meaning: given a set of universal gates, we can approximate an arbitrary unitary by composition of said gates. In the CV setting, we have a similar situation: we can approximate a broad set of *generators* – i.e., the Hamiltonians appearing in Eq. (10) – by combining elements of a CV universal gate set. However, unlike the qubit case, we do not try to approximate all conceivable unitaries. Rather, we seek to create all generators that are a *polynomial* function of the quadrature (or mode) operators of the system [48, 49]. Remember that generators of second-degree or lower belong to the class of *Gaussian* operations, while all higher degrees are *non-Gaussian*.

We can create a higher-order generator out of lower-order generators $\hat{A}$ and $\hat{B}$ by using the following two concatenation identities [48]:

$$e^{-i\hat{A}\delta t}e^{-i\hat{B}\delta t}e^{i\hat{A}\delta t}e^{i\hat{B}\delta t} = e^{[\hat{A},\hat{B}]\delta t^2} + O(\delta t^3), \quad \text{(A1a)}$$

$$e^{i\hat{A}\delta t/2}e^{i\hat{B}\delta t/2}e^{i\hat{B}\delta t/2}e^{i\hat{A}\delta t/2} = e^{i(\hat{A}+\hat{B})\delta t} + O(\delta t^2). \quad \text{(A1b)}$$

If we have two second-degree generators, such as $\hat{u} = \hat{x}^2 + \hat{p}^2$ (the generator for the rotation gate) and $\hat{s} = \hat{x}\hat{p} + \hat{p}\hat{x}$ (the generator for the squeezing gate), and a third-degree (or higher) generator, such as $\hat{x}^3$ (the generator for the cubic phase gate), we can easily construct generators of all higher-degree polynomials, e.g., $\hat{x}^4 = -[\hat{x}^3,[\hat{x}^3,\hat{u}]]/(18\hbar^2)$. This reasoning can be extended by induction to any finite-degree polynomial in $\hat{x}$ and $\hat{p}$ (equivalently, in $\hat{a}$ and $\hat{a}^\dagger$) [48, 49].

In the above argument, it is important that at least one of the generators is third-degree or higher. Indeed, commutators of second-degree polynomials of $\hat{x}$ and $\hat{p}$ are also second-degree polynomials and thus their composition using Eq. (A1) cannot generate higher-order generators. The claim can be easily extended to N-mode systems and multivariate polynomials of the operators

$$\hat{\mathbf{r}} = (\hat{x}_1,\ldots,\hat{x}_N,\hat{p}_1,\ldots,\hat{p}_N). \quad \text{(A2)}$$

Combining single-mode universal gates (including at least one of third-degree or higher) with some multimode interaction, e.g., the beamsplitter interaction generated by $\hat{b}_{i,j} = \hat{p}_i\hat{x}_j - \hat{x}_i\hat{p}_j$, we can construct arbitrary-degree polynomials of the quadrature operators of an N-mode system.

With the above discussion in mind, we can combine the set of single-qumode gates generated by $\{\hat{x}_i,\hat{x}_i^2,\hat{x}_i^3,\hat{u}_i\}$ and the two-mode gate generated by $\hat{b}_{i,j}$ for all pairs of modes into a universal gate set. The first, third and fourth single-mode generators correspond to the displacement, cubic phase and rotation gates in Table III, while the two-mode generator corresponds to the beamsplitter gate. Finally, the $\hat{x}^2$ generator corresponds to a quadratic phase gate, $\hat{P}(s) = \exp(is\hat{x}^2/(2\hbar))$. This gate can be written in terms of the single-mode squeezing gate and the rotation gate as follows: $\hat{P}(s) = \hat{R}(\theta)\hat{S}(re^{i\phi})$, where $\cosh(r) = \sqrt{1+(s/2)^2}$, $\tan(\theta) = s/2, \phi = -\theta - \text{sign}(s)\pi/2$. Equivalently, we could have included the squeezing generator $\hat{x}\hat{p} + \hat{p}\hat{x}$ in place of the quadratic phase and still had a universal set.

We have just outlined an efficient method to construct any gate of the form $\exp(-iHt)$, where the generator H is a polynomial of the quadrature (or mode) operators. How can this be used for quantum computations? As shown in Eq. (4), the eigenstates of the $\hat{x}$ quadrature form an (orthogonal) basis for representing qumode states. Thus, these states are often taken as the computational basis of the CV model (though other choices are also available). By applying gates as constructed above and performing measurements in this computational basis (i.e., homodyne measurements), we can carry out a CV quantum computation. One primitive example [49] is to compute the product of two numbers – whose values are stored in two qumode registers – and store the result in the state of a third qumode. Consider the generator $\hat{x}_1\hat{x}_2\hat{p}_3/\hbar$, which will cause the "position" operators to evolve according to

$$\hat{x}_1 \to \hat{x}_1, \quad \hat{x}_2 \to \hat{x}_2, \quad \hat{x}_3 \to \hat{x}_3 + \hat{x}_1\hat{x}_2 t. \quad \text{(A3)}$$

A measurement in the computational basis of mode 3 will reveal the value of the product x_1x_2. Note that these encodings of classical continuous degress of freedom into quantum registers allows for the generalization of neural networks into the quantum regime [66]. In the following sections we show how more complicated algorithms are constructed.

Multiport Gate Decompositions

One important set of gates for which it is critical to derive a decomposition in terms of universal gates is the set of multiport interferometers. A multiport interferometer, represented by a unitary operator $\hat{\mathcal{U}}$ acting on N modes, will map (in the Heisenberg picture) the annihilation operator of mode i into a linear combination of all other modes

$$\hat{a}_i \to \hat{\mathcal{U}}^\dagger\hat{a}_i\hat{\mathcal{U}} = \sum_j (U_{ij}\hat{a}_j). \quad \text{(A4)}$$

In order to preserve the commutation relations of different modes, the matrix U must also be unitary $UU^\dagger = U^\dagger U = \mathbb{I}_N$. Note that every annihilation operator is mapped to a linear combination of annihilation operators and thus annihilation and creation operators are not mixed. Because of this, multiport interferometers generate all passive (in the sense that no photons are created or destroyed) linear optics transformations. In a pioneering work by Reck *et al.* [67], it was shown that any multiport interferometer can be realized using $N(N-1)/2$ beamsplitter gates distributed over $2N-3$

layers. Recently this result was improved upon by Clements *et al.* [63], who showed that an equivalent decomposition can be achieved with the same number of beamsplitters but using only $N+1$ layers.

Gaussian Operations

As mentioned in the previous subsections, generators that are at most quadratic remain closed under composition of their associated unitaries. In the Heisenberg picture these quadratic generators will produce all possible linear transformations between the quadrature (or mode) operators,

$$\hat{a}_i \to \sum_j U_{ij}\hat{a}_j + V_{ij}\hat{a}_j^\dagger. \tag{A5}$$

These operations are known as Gaussian operations. All gates in Table III are Gaussian operations except for the cubic phase gate. Pure Gaussian states are the set of states that can be obtained from the (multimode) vacuum state by Gaussian operations [50, 68]. Mixed Gaussian states are obtained by applying Gaussian operations to thermal states or marginalizing pure Gaussian states. Analogous to Gaussian states, we can also define Gaussian measurements as the set of measurements whose positive-operator valued measure (POVM) elements can be obtained from vacuum via Gaussian transformations. Homodyne and heterodyne measurements are prominent examples of Gaussian measurements, whereas photon counting and photodetection are prominent examples of *non*-Gaussian measurements.

An important result for the CV formalism is that Gaussian quantum computation, i.e., computation that occurs with Gaussian states, operations and measurements, can be *efficiently* simulated on a classical computer (this is the foundation for the Gaussian backend in Strawberry Fields). This result is the CV version [69] of the Gottesman-Knill theorem of discrete-variable quantum information [49]. Hence we need non-Gaussian operations in order to achieve quantum supremacy in the CV model. Interestingly, even in the restricted case where all states and gates are Gaussian, with only the final measurements being non-Gaussian, there is strong evidence that such a circuit cannot be efficiently simulated classically [10, 70]. More discussion and example code for this situation (known as *Gaussian boson sampling*) is provided in Appendix C.

Symplectic Formalism

In this section we review the symplectic formalism which lies at the heart of the Gaussian backend of Strawberry Fields. The symplectic formalism is an elegant and compact description of Gaussian states in terms of covariance matrices and mean vectors [50, 68]. To begin, the commutation relations of the $2N$ position and momentum operators of Eq. (A2) can be easily summarized as $[\hat{r}_i, \hat{r}_j] = \hbar i \Omega_{ij}$ where $\Omega = \left(\begin{smallmatrix} 0 & \mathbb{I}_N \\ -\mathbb{I}_N & 0 \end{smallmatrix}\right)$ is the *symplectic matrix*. Using the symplectic matrix, we can define the *Weyl operator* $D(\xi)$ (a multimode displacement operator) and the *characteristic function* $\chi(\xi)$ of a quantum N mode state ρ:

$$\hat{D}(\xi) = \exp(i\hat{\mathbf{r}}\Omega\xi), \quad \chi(\xi) = \langle \hat{D}(\xi)\rangle_\rho, \tag{A6}$$

where $\xi \in \mathbb{R}^{2N}$. We can now consider the Fourier transform of the characteristic function to obtain the *Wigner function* of the state $\hat{\rho}$

$$W(\mathbf{r}) = \int_{\mathbb{R}^{2N}} \frac{d^{2N}\xi}{(2\pi)^{2N}} \exp(-i\mathbf{r}\Omega\xi)\chi(\xi). \tag{A7}$$

The $2N$ real arguments $\mathbf{r}$ of the Wigner function are the eigenvalues of the quadrature operators from $\hat{\mathbf{r}}$.

The above recipe maps an N-mode quantum state living in a Hilbert space to the real symplectic space $\mathcal{K} := (\mathbb{R}^{2N}, \Omega)$, which is called *phase space*. The Wigner function is an example of a *quasiprobability distribution*. Like a probability distribution over this phase space, the Wigner function is normalized to one; however, unlike a probability distribution, it may take negative values. Gaussian states have the special property that their characteristic function (and hence their Wigner function) is a Gaussian function of the variables $\mathbf{r}$. In this case, the Wigner function takes the form

$$W(\mathbf{r}) = \frac{\exp\left(-\frac{1}{2}(\mathbf{r}-\bar{\mathbf{r}})\mathbf{V}^{-1}(\mathbf{r}-\bar{\mathbf{r}})\right)}{(2\pi)^N\sqrt{\det\mathbf{V}}} \tag{A8}$$

where $\bar{\mathbf{r}} = \langle\hat{\mathbf{r}}\rangle_\rho = \mathrm{Tr}(\hat{\mathbf{r}}\hat{\rho})$ is the displacement or mean vector and $\mathbf{V}_{ij} = \frac{1}{2}\langle\Delta r_i \Delta r_j + \Delta r_i \Delta r_j\rangle_\rho$ with $\Delta\hat{\mathbf{r}} = \hat{\mathbf{r}} - \bar{\mathbf{r}}$. Note that the only pure states that have non-negative Wigner functions are the pure Gaussian states [58].

Each type of Gaussian state has a specific form of covariance matrix $\mathbf{V}$ and mean vector $\bar{\mathbf{r}}$. For the single-mode vacuum state, we have $\mathbf{V} = \frac{\hbar}{2}\mathbb{I}_2$ and $\bar{\mathbf{r}} = (0,0)^T$. A thermal state (Eq. (8)) has the same (zero) displacement but a covariance matrix $\mathbf{V} = (2\bar{n}+1)\frac{\hbar}{2}\mathbb{I}_2$, where $\bar{n}$ is the mean photon number. A coherent state (Eq. (6)), obtained by displacing vacuum, has the same $\mathbf{V}$ as the vacuum state but a nonzero displacement vector $\bar{\mathbf{r}} = 2\sqrt{\frac{\hbar}{2}}(\mathrm{Re}(\alpha), \mathrm{Im}(\alpha))$. Lastly, a squeezed state (Eq. (7)) has zero displacement and covariance matrix $\mathbf{V} = \frac{\hbar}{2}\mathrm{diag}(e^{-2r}, e^{2r})$. In the limit $r \to \infty$, the squeezed state's variance in the $\hat{x}$ quadrature becomes zero and the state becomes proportional to the $\hat{x}$-eigenstate $|x\rangle$ with eigenvalue 0. Consistent with the uncertainty principle, the squeezed state's variance in $\hat{p}$ blows up. We can also consider the case $r \to -\infty$, where we find a state proportional to the eigenstate $|p\rangle$ of the $\hat{p}$ quadrature with eigenvalue 0. In the laboratory and in numerical simulation we must approximate every quadrature eigenstate using a finitely squeezed state (being careful that the variance of the relevant quadrature is much smaller than any other uncertainty relevant to the system). Any other quadrature eigenstate can be obtained from the $x = 0$ eigenstate by applying suitable displacement and rotation operators. Finally, note that Gaussian operations will transform the vector of means via affine transformations and the covariance matrix via congruence transformations; for a detailed dis-

cussion of these transformations, see Sec. 2 of [50].

Given a $2N \times 2N$ real symmetric matrix, how can we check that it is a valid covariance matrix? If it is valid, which operations (displacement, squeezing, multiport interferometers) should be performed to prepare the corresponding Gaussian state? To answer the first question: a $2N \times 2N$ real symmetric matrix $\tilde{\mathbf{V}}$ corresponds to a Gaussian quantum state if and only if $\tilde{\mathbf{V}} + i\frac{\hbar}{2}\Omega \geq 0$ (the matrix inequality is understood in the sense that the eigenvalues of the quantity $\tilde{\mathbf{V}} + i\frac{\hbar}{2}\Omega$ are nonnegative). The answer to the second question is provided by the *Bloch-Messiah reduction* [60–62]. This reduction shows that any N-mode Gaussian state (equivalently any covariance matrix and vector of means) can be constructed by starting with a product of N thermal states $\bigotimes_i \rho_i(\bar{n}_i)$ (with potentially different mean photon numbers), then applying a multiport interferometer $\hat{\mathcal{U}}$, followed by single-mode squeezing operations $\bigotimes_i S_i(z_i)$, followed by another multiport $\hat{\mathcal{V}}$, followed by single-mode displacement operations $\bigotimes_i D_i(\alpha_i)$. Explicitly,

$$\rho_{\text{Gaussian}} = \hat{\mathcal{W}}\left(\bigotimes_i \rho_i(\bar{n}_i)\right)\hat{\mathcal{W}}^\dagger, \tag{A9}$$

$$\hat{\mathcal{W}} = \left(\bigotimes_i D_i(\alpha_i)\right)\hat{\mathcal{V}}\left(\bigotimes_i S_i(z_i)\right)\hat{\mathcal{U}}. \tag{A10}$$

Note that if the Gaussian state is pure (which happens if and only if $\det(\mathbf{V}) = (\hbar/2)^{2N}$), the occupation number of the thermal states in the Bloch-Messiah decomposition are all zero and the first interferometer will turn the vacuum to vacuum again. Thus for pure Gaussian states we need only generate N single-mode squeezed states and send them through a single multiport interferometer $\hat{\mathcal{V}}$ before displacing. For a recent discussion of this decomposition see Ref. [71, 72]. More generally, the occupation numbers of the different thermal states in Eq. (A9) $n_i = (\nu_i - 1)/2$ can be obtained by calculating the symplectic eigenvalues ν_i of the covariance matrix $\mathbf{V}$. The symplectic eigenvalues come in pairs and are just the standard eigenvalues of the matrix $|i(2/\hbar)\Omega\mathbf{V}|$ where the modulus is understood in the operator sense (see Sec. II.C.1. of Ref. [50]).

Appendix B: Strawberry Fields Operations

In this Appendix, we present a complete list of the CV states, gates, and measurements available in Strawberry Fields.

Operation	Name	Definition
`Vacuum()`	Vacuum state	The vacuum state $\|0\rangle$, representing zero photons
`Coherent(a)`	Coherent state	A displaced vacuum state, $\|\alpha\rangle = D(\alpha)\|0\rangle$, $\alpha \in \mathbb{C}$
`Squeezed(r,phi)`	Squeezed state	A squeezed vacuum state, $\|z\rangle = S(z)\|0\rangle$, where $z = re^{i\phi}, \quad r, \phi \in \mathbb{R}$, $r \geq 0$, $\phi \in [0, 2\pi)$
`DisplacedSqueezed(a,r,phi)`	Displaced squeezed state	A squeezed then displaced vacuum state, $\|\alpha, z\rangle = D(\alpha)S(z)\|0\rangle$
`Thermal(n)`	Thermal state	$\rho(\bar{n}) = \sum_{n=0}^{\infty}[\bar{n}^n/(1+\bar{n})^{n+1}]\|n\rangle\langle n\|$, where $\bar{n} \in \mathbb{R}^+$ is the mean photon number
`Fock(n)*`	Fock state or number state	$\|n\rangle$, where $n \in \mathbb{N}_0$ represents the photon number
`Catstate(a,p)*`	Cat state	$\frac{1}{\sqrt{\mathcal{N}}}(\|\alpha\rangle + \exp(i\pi p)\|-\alpha\rangle)$, where $p = 0, 1$ gives an even/odd cat state and $\mathcal{N}$ is the normalization
`Ket(x)*`	Arbitrary Fock-basis ket	Prepare an arbitrary multi-mode pure state, represented by array `x`, in the Fock basis.
`DensityMatrix(x)*`	Arbitrary Fock basis state	Prepare an arbitrary multi-mode mixed state, represented by a density matrix array `x` in the Fock basis.

Table V: *State preparations available in Strawberry Fields. Those indicated with an asterisk (*) are non-Gaussian.*

Operation	Name	Definition
`Dgate(a)`	Displacement gate	$D(\alpha) = \exp(\alpha\hat{a}^\dagger - \alpha^*\hat{a}) = \exp\left(-i(\mathrm{Re}(\alpha)\hat{p} - \mathrm{Im}(\alpha)\hat{x})\sqrt{2/\hbar}\right), \quad \alpha \in \mathbb{C}$
`Xgate(x)`	Position displacement gate	$X(x) = D(x/\sqrt{2\hbar}) = \exp(-ix\hat{p}/\hbar)$, $x \in \mathbb{R}$
`Zgate(p)`	Momentum displacement gate	$Z(p) = D(ip/\sqrt{2\hbar}) = \exp(ip\hat{x}/\hbar)$, $p \in \mathbb{R}$
`Sgate(r,phi)`	Squeezing gate	$S(z) = \exp\left((z^*\hat{a}^2 - z\hat{a}^{\dagger 2})/2\right)$, where $z = r\exp(i\phi), \quad r, \phi \in \mathbb{R}$, $r \geq 0$, $\phi \in [0, 2\pi)$
`Rgate(theta)`	Rotation gate	$R(\theta) = \exp\left(i\theta\hat{a}^\dagger\hat{a}\right)$, where $\theta \in [0, 2\pi)$
`Fouriergate()`	Fourier gate	$F = R(\pi/2) = \exp\left(i(\pi/2)\hat{a}^\dagger\hat{a}\right)$
`Pgate(s)`	Quadratic phase gate	$P(s) = \exp\left(is\hat{x}^2/(2\hbar)\right)$, where $s \in \mathbb{R}$
`Vgate(g)*`	Cubic phase gate	$V(\gamma) = \exp\left(i\gamma\hat{x}^3/(3\hbar)\right)$, where $\gamma \in \mathbb{R}$
`Kgate(k)*`	Kerr interaction gate	$K(\kappa) = \exp\left(i\kappa\left(\hat{a}^\dagger\hat{a}\right)^2\right)$, where $\kappa \in \mathbb{R}$

Table VI: *Single mode gate operations available in Strawberry Fields. Those indicated with an asterisk (*) are non-Gaussian and can only be used with a backend that uses the Fock representation.*

Operation	Name	Definition
`BSgate(theta,phi)`	Beamsplitter	$B(\theta,\phi)=\exp\left(\theta(e^{i\phi}\hat{a}_1\hat{a}_2^\dagger - e^{-i\phi}\hat{a}_1^\dagger\hat{a}_2)\right)$, where the transmissivity and reflectivity amplitudes are $t=\cos\theta$, $r=e^{i\phi}\sin\theta$
`S2gate(r,p)`	Two-mode squeezing gate	$S_2(z)=\exp\left(z^*\hat{a}_1\hat{a}_2 - z\hat{a}_1^\dagger\hat{a}_2^\dagger\right)$, where $z=r\exp(i\phi)$
`CXgate(s)`	Controlled-X or addition gate	$CX(s)=\exp(-is\hat{x}_1\hat{p}_2/\hbar)$, $s\in\mathbb{R}$
`CZgate(s)`	Controlled phase shift gate	$CZ(s)=\exp(is\hat{x}_1\hat{x}_2/\hbar)$, $s\in\mathbb{R}$
`CKgate(k)*`	Controlled Kerr interaction gate	$CK(\kappa)=\exp\left(i\kappa\hat{a}_1^\dagger\hat{a}_1\hat{a}_2^\dagger\hat{a}_2\right)$, $\kappa\in\mathbb{R}$

Table VII: *Two-mode gate operations available in Strawberry Fields.*

Operation	Name	Definition
`Gaussian(cov,mu)`	Gaussian state preparation	Prepares an arbitrary N-mode Gaussian state defined by covariance matrix $V\in\mathbb{R}^{2N\times 2N}$ and means vector $\mu\in\mathbb{R}^{2N}$ using the Williamson decomposition
`GaussianTransform(S)`	Gaussian transformation	Applies an N-mode Gaussian transformation defined by symplectic matrix $S\in\mathbb{R}^{2N\times 2N}$ using the Bloch-Messiah decomposition
`Interferometer(U)`	Multi-mode linear interferometer	Applies an N-mode interferometer defined by unitary matrix $U\in\mathbb{C}^{N\times N}$ using the Clements decomposition

Table VIII: *Multi-mode Gaussian decompositions available in Strawberry Fields.*

Operation	Name	Definition
`MeasureHomodyne(phi)`	Homodyne measurement	Projects the state onto $\lvert x_\phi\rangle\langle x_\phi\rvert$ where $\hat{x}_\phi=\cos\phi\,\hat{x}+\sin\phi\,\hat{p}$
`MeasureHeterodyne()`	Heterodyne measurement	Projects the state onto the coherent states, sampling from the joint Husimi distribution $\frac{1}{\pi}\langle\alpha\rvert\rho\lvert\alpha\rangle$
`MeasureFock()*`	Photon counting	Projects the state onto $\lvert n\rangle\langle n\rvert$

Table IX: *Measurement operations available in Strawberry Fields. Those indicated with an asterisk (*) are non-Gaussian and can only be used with a backend that uses the Fock representation.*

Gate Decompositions

In addition, the Strawberry Fields frontend can be used to provide decompositions of certain compound gates. The following gate decompositions are currently supported.

Quadratic phase gate

The quadratic phase shift gate `Pgate(s)` is decomposed into a squeezing and a rotation,

$$P(s)=R(\theta)S(re^{i\phi}),$$

where $\cosh(r)=\sqrt{1+(s/2)^2}$, $\tan(\theta)=s/2$, and $\phi=-\text{sign}(s)\frac{\pi}{2}-\theta$.

Two-mode squeeze gate

The two-mode squeeze gate `S2gate(z)` is decomposed into a combination of beamsplitters and single-mode squeezers

$$S_2(z)=B^\dagger(\pi/4,0)[S(z)\otimes S(-z)]B(\pi/4,0)$$

Controlled addition gate

The controlled addition or controlled-X gate `CXgate(s)` is decomposed into a combination of beamsplitters and single-mode squeezers

$$CX(s) = B(\phi, 0)\,[S(r) \otimes S(-r)]\,B(\pi/2 + \phi, 0),$$

where $\sin(2\phi) = -1/\cosh(r)$, $\cos(2\phi) = -\tanh(r)$, and $\sinh(r) = -s/2$.

Controlled phase gate

The controlled phase shift gate `CZgate(s)` is decomposed into a controlled addition gate, with two rotation gates acting on the target mode,

$$CZ(s) = [\mathbb{I} \otimes R(\pi/2)]\,CX(s)\left[\mathbb{I} \otimes R(\pi/2)^\dagger\right]$$

Appendix C: Quantum Algorithms

In this Appendix, we present full example code for several important algorithms, subroutines, and use-cases for Strawberry Fields. These examples are presented in more detail in the online documentation located at strawberryfields.readthedocs.io.

Quantum Teleportation

Quantum teleportation — sometimes referred to as state teleportation to avoid confusion with gate teleportation — is the reliable transfer of an unknown quantum state across spatially separated qubits or qumodes, through the use of a classical transmission channel and quantum entanglement [73]. Considered a fundamental quantum information protocol, it has applications ranging from quantum communication to enabling distributed information processing in quantum computation [74].

In general, all quantum teleportation circuits work on the same basic principle. Two distant operators, Alice and Bob, share a maximally entangled quantum state (in discrete variables, any one of the four Bell states, and in CVs, a maximally entangled state for a fixed energy), and have access to a classical communication channel. Alice, in possession of an unknown state which she wishes to transfer to Bob, makes a joint measurement of the unknown state and her half of the entangled state, by projecting onto the Bell or quadrature basis. By transmitting the results of her measurement to Bob, Bob is then able to transform his half of the entangled state to an accurate replica of the original unknown state, by performing a conditional phase flip (for qubits) or displacement (for qumodes) [75]. The CV teleportation circuit is shown in Fig. 3.

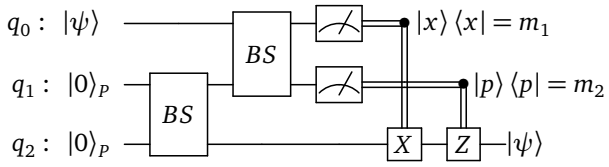

FIG. 3: *State teleportation of state $|\psi\rangle$ from mode q_0 to mode q_2.*

```
import strawberryfields as sf
from strawberryfields.ops import *
from strawberryfields.utils import scale
from numpy import pi, sqrt

eng,q = sf.Engine(3)

with eng:
  psi, alice, bob = q[0], q[1], q[2]

  # state to be teleported:
  Coherent(1+0.5j) | psi

  # 50-50 beamsplitter
  BS = BSgate(pi/4, 0)

  # maximally entangled states
  Squeezed(-2) | alice
  Squeezed(2) | bob
  BS | (alice, bob)

  # Alice performs the joint measurement
  # in the maximally entangled basis
  BS | (psi, alice)
  MeasureX | psi
  MeasureP | alice

  # Bob conditionally displaces his mode
  # based on Alice's measurement result
  Xgate(scale(psi, sqrt(2))) | bob
  Zgate(scale(alice, sqrt(2))) | bob
  # end circuit

state = eng.run("gaussian")
# view Bob's output state and fidelity
print(q[0].val,q[1].val)
print(state.displacement([2]))
print(state.fidelity_coherent([0,0,1+0.5j]))
```

Codeblock 15: *Quantum teleportation of a coherent state in Strawberry Fields. Note: while the CV quantum teleportation algorithm relies on infinitely squeezed resource states, in practice a squeezing magnitude of -2 ($\sim$ 18 dB) is sufficient.*

Gate Teleportation

In the quantum state teleportation algorithm discussed in the previous section, the quantum state is transferred from the sender to the receiver. However, quantum teleportation can be used in a much more powerful manner, by simultaneously transforming the teleported state — this is known as gate teleportation.

In gate teleportation, rather than applying a quantum unitary directly to the state prior to teleportation, the unitary is applied indirectly, via the projective measurement of the first subsystem onto a particular basis. This measurement-based approach provides significant advantages over applying unitary gates directly, for example by reducing resources, and in the application of experimentally hard-to-implement gates [74]. In fact, gate teleportation forms a universal quantum computing primitive, and is a precursor to cluster state models of quantum computation [76].

First described by Gottesman and Chuang [77] in the case of qubits, gate teleportation was generalised for the CV case by Bartlett and Munro [78] (see Fig. 4). In an analogous process to the discrete-variable case, we begin with the algorithm for local state teleportation. Unlike the spatially-separated quantum state teleportation we considered in the previous section, local teleportation can transport the state using only two qumodes; the state we are teleporting is entangled directly with the squeezed vacuum state in the momentum space through the use of a controlled phase gate.

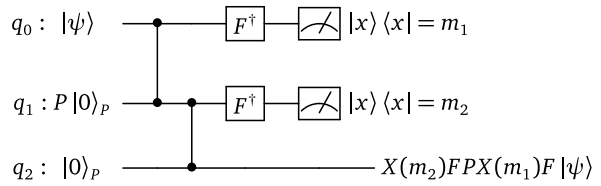

FIG. 4: *Gate teleportation of a quadratic phase gate P onto input state $|\psi\rangle$.*

To recover the teleported state exactly, we must perform Weyl-Heisenberg corrections to the second mode; here, that would be $F^\dagger X(m)^\dagger$, where m is the correction based on the Homodyne measurement. However, for convenience and simplicity, it is common to simply write the circuit without the corrections applied explicitly.

```
import strawberryfields as sf
from strawberryfields.ops import *

eng, q = sf.Engine(4)

with eng:
  # create initial states
  Squeezed(0.1) | q[0]
  Squeezed(-2)  | q[1]
  Squeezed(-2)  | q[2]

  # apply the gate to be teleported
  Pgate(0.5) | q[1]

  # conditional phase entanglement
  CZgate(1) | (q[0], q[1])
  CZgate(1) | (q[1], q[2])

  # projective measurement onto
  # the position quadrature
  Fourier.H | q[0]
  MeasureX | q[0]
  Fourier.H | q[1]
  MeasureX | q[1]
  # compare against the expected output
  # X(q1).F.P(0.5).X(q0).F.|z>
  # not including the corrections
  Squeezed(0.1) | q[3]
  Fourier       | q[3]
  Xgate(q[0])   | q[3]
  Pgate(0.5)    | q[3]
  Fourier       | q[3]
  Xgate(q[1])   | q[3]
  # end circuit

state = eng.run("gaussian")
print(state.reduced_gaussian([2]))
print(state.reduced_gaussian([3]))
```

Codeblock 16: *Gate teleportation of a quadratic phase gate in Strawberry Fields.*

Note that additional gates can be added to the gate teleportation scheme described above simply by introducing additional qumodes with the appropriate projective measurements, all 'stacked vertically' (i.e., coupled to each consecutive qumode via a conditional phase gate). It is from this primitive that the model of cluster state quantum computation can be derived [76].

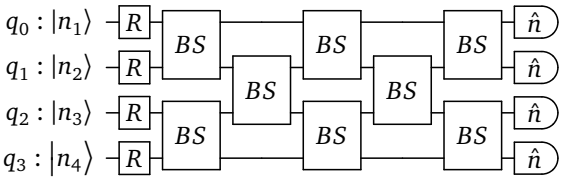

FIG. 5: *4-mode boson sampling.*

Boson Sampling

Introduced by Aaronson and Arkhipov [9], boson sampling presented a slight deviation from the general approach in quantum computation. Rather than presenting a theoretical model of universal quantum computation (i.e., a framework that enables quantum simulation of any arbitrary Hamiltonian [51]), boson sampling-based devices are instead an example of an intermediate quantum computer, designed to experimentally implement a computation that is intractable classically [79–82].

Boson sampling proposes the following quantum linear optics scheme. An array of single photon sources is set up, designed to simultaneously emit single photon states into a multimode linear interferometer; the results are then generated by sampling from the probability of single photon measurements from the output of the linear interferometer.

For example, consider N single photon Fock states, $|\psi\rangle = |m_1, m_2, \ldots, m_N\rangle$, composed of $b = \sum_i m_i$ photons, incident on an N-mode linear interferometer described by the unitary U acting on the input mode creation and annihilation operators. It was shown that the probability of detecting n_j photons at the jth output mode is given by [9]

$$\Pr(n_1, \ldots, n_N) = \frac{|\mathrm{Per}(U_{st})|^2}{m_1! \cdots m_N! n_1! \cdots n_N!}; \qquad \text{(C1)}$$

i.e., the sampled single photon probability distribution is proportional to the permanent of U_{st}, a submatrix of the interferometer unitary, dependent upon the input and output Fock states. Whilst the determinant can be calculated efficiently on classical computers, calculation of the permanent belongs to the computational complexity class #P-Hard problems [83], which are strongly believed to be classically hard to calculate. This implies that simulating boson sampling is an intractable task for classical computers, providing an avenue for the demonstration of quantum supremacy.

Continuous-variable quantum computation is ideally suited to the simulation and demonstration of the boson sampling scheme, due to its grounding in quantum optics. In quantum linear optics, the multimode linear interferometer is commonly decomposed into two-mode beamsplitters (`BSgate`) and single-mode phase shifters (`Rgate`) [67], allowing for a straightforward translation into a CV quantum

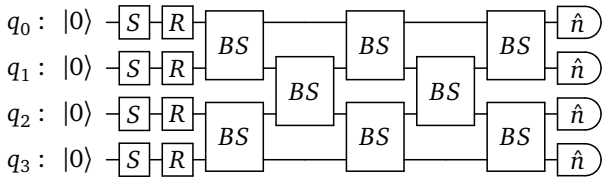

FIG. 6: *4-mode Gaussian boson sampling.*

circuit. In order to allow for arbitrary linear unitaries on m imput modes, we must have a minimum of $m+1$ columns in the beamsplitter array [63].

```
import numpy as np
import strawberryfields as sf
from strawberryfields.ops import *

# initialise the engine and register
eng, q = sf.Engine(4)

with eng:
  # prepare the input fock states
  Fock(1) | q[0]
  Fock(1) | q[1]
  Vac     | q[2]
  Fock(1) | q[3]

  # rotation gates
  Rgate(0.5719)
  Rgate(-1.9782)
  Rgate(2.0603)
  Rgate(0.0644)

  # beamsplitter array
  BSgate(0.7804, 0.8578)  | (q[0], q[1])
  BSgate(0.06406, 0.5165) | (q[2], q[3])
  BSgate(0.473, 0.1176)   | (q[1], q[2])
  BSgate(0.563, 0.1517)   | (q[0], q[1])
  BSgate(0.1323, 0.9946)  | (q[2], q[3])
  BSgate(0.311, 0.3231)   | (q[1], q[2])
  BSgate(0.4348, 0.0798)  | (q[0], q[1])
  BSgate(0.4368, 0.6157)  | (q[2], q[3])
  # end circuit

# run the engine
state = eng.run("fock", cutoff_dim=7)

# extract the joint Fock probabilities
probs = state.all_fock_probs()

# print the joint Fock state probabilities
print(probs[1,1,0,1])
print(probs[2,0,0,1])
```

Codeblock 17: *4-mode boson sampling example in Strawberry Fields. Parameters are chosen arbitrarily.*

Gaussian Boson Sampling

While boson sampling allows the experimental implementation of a sampling problem that is hard to simulate classically, one of the setbacks in experimental setups is scalability, due to its dependence on an array of simultaneously emitting single photon sources. Currently, most physical implementations of boson sampling make use of a process known as Spontaneous Parametric Down-Conversion (SPDC) to generate the single-photon source inputs. However, this method is non-deterministic – as the number of modes in the apparatus increases, the average time required until every photon source emits a simultaneous photon increases exponentially.

In order to simulate a deterministic single photon source array, several variations on boson sampling have been proposed; the most well-known being scattershot boson sampling [70]. However, a recent boson sampling variation by [10] negates the need for single photon Fock states altogether, by showing that incident Gaussian states – in this case, single mode squeezed states – can produce problems in the same computational complexity class as boson sampling. Even more significantly, this mitigates the scalability problem with single photon sources, as single mode squeezed states can be simultaneously generated experimentally.

With an input ensemble of N single-mode squeezed states with squeezing parameter $z = r \in \mathbb{R}$, incident on a linear-interferometer described by unitary U, it can be shown that the probability of detecting an output photon pattern $(n_1, \ldots, n_N)$, where $n_k \in \{0, 1\}$, is given by [10]

$$\Pr(n_1, \ldots, n_N) = \frac{\left|\mathrm{Haf}[(U \bigoplus_i \tanh(r_i) U^T)_S]\right|^2}{\prod_i \cosh(r_i)}, \qquad \text{(C2)}$$

where S denotes the subset of modes where a photon was detected and $\mathrm{Haf}[\cdot]$ is the *Hafnian* [84, 85]. That is, the sampled single photon probability distribution is proportional to the hafnian of a submatrix of $U \bigoplus_i \tanh(r_i) U^T$. The hafnian is known to be a *generalization* of the permanent, and can be used to count the number of perfect matchings of an arbitrary graph [86]. The formula above can be generalized to pure Gaussian states with finite displacements by using the loop hafnian function which counts the number of perfect matchings of graphs with loops[56]. Since any algorithm that could calculate the hafnian could also calculate the permanent, it follows that calculating the hafnian remains a classically hard problem; indeed, the best known classical algorithm for the calculation of a hafnian of an arbitrary symmetric complex matrix of size N scales like $O(N^3 2^{N/2})$ [87]. The hardness of approximate GBS, under imperfections such as loss, is a subject of current research [88, 89].

```
import numpy as np
import strawberryfields as sf
from strawberryfields.ops import *

# initialise the engine and register
eng, q = sf.Engine(4)

with eng:
  # prepare the input squeezed states
```

```
  S = Sgate(1)
  S | q[0]
  S | q[1]
  S | q[2]
  S | q[3]

  # rotation gates
  Rgate(0.5719)  | q[0]
  Rgate(-1.9782) | q[1]
  Rgate(2.0603)  | q[2]
  Rgate(0.0644)  | q[3]

  # beamsplitter array
  BSgate(0.7804, 0.8578)  | (q[0], q[1])
  BSgate(0.06406, 0.5165) | (q[2], q[3])
  BSgate(0.473, 0.1176)   | (q[1], q[2])
  BSgate(0.563, 0.1517)   | (q[0], q[1])
  BSgate(0.1323, 0.9946)  | (q[2], q[3])
  BSgate(0.311, 0.3231)   | (q[1], q[2])
  BSgate(0.4348, 0.0798)  | (q[0], q[1])
  BSgate(0.4368, 0.6157)  | (q[2], q[3])
  # end circuit

# run the engine
state = eng.run("gaussian")

# Fock states to measure at output
measure_states = [[0,0,0,0], [1,1,0,0],
 ↪  [0,1,0,1], [1,1,1,1], [2,0,0,0]]

# extract the probabilities of calculating
# several different Fock states at the output
# and print them to the terminal
for i in measure_states:
    prob = state.fock_prob(i)
    print("|{}>: {}".format("".join(str(j) for
     ↪  j in i), prob))
```

Codeblock 18: *4-mode Gaussian boson sampling example in Strawberry Fields. Parameters are chosen arbitrarily.*

Instantaneous Quantum Polynomial (IQP)

Like boson sampling and Gaussian boson sampling before it, the instantaneous quantum polynomial (IQP) protocol is a restricted, non-universal model of quantum computation, designed to implement a computation that is classically intractable and verifiable by a remote adjudicator. First introduced by Shepherd and Bremner [90], IQP circuits are defined by the following conditions: (i) there are N inputs in state $|0\rangle$ acted on by Hadamard gates, (ii) the resulting computation is diagonal in the computational basis by randomly selecting from the set $U = \{R(\pi/4), \sqrt{CZ}\}$ (hence the term 'instantaneous'; since all gates commute, they can be applied in any temporal order), and (iii) the output qubits are measured in the Pauli-X basis. Efficient classical sampling of the resulting probability distribution $H^{\otimes N} U H^{\otimes N} |0\rangle^{\otimes N}$ — even approximately [13] or in the presence of noise [91] — has been shown to be #P-hard, and would result in the collapse of the polynomial hierarchy to the third level [12, 92].

Unlike boson sampling and Gaussian boson sampling,

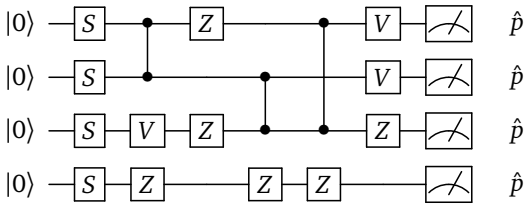

FIG. 7: *4-mode instantaneous quantum polynomial (IQP) CV circuit example.*

however, the IQP protocol was not constructed with quantum linear optics in mind. Nevertheless, the IQP model was recently extended to the CV formulation of quantum computation by Douce et al. [17], taking advantage of the ability to efficiently prepare large resource states, and the higher efficiencies afforded by homodyne detection. Furthermore, the computational complexity results of the discrete-variable case apply equally to the so-called CV-IQP model, assuming a specific input squeezing parameter dependent on the circuit size. The CV-IQP model is defined as follows:

1. inputs are squeezed momentum states $|z\rangle$, where $z = r \in \mathbb{R}$ and $r < 0$;
2. the unitary transformation is diagonal in the $\hat{x}$ quadrature basis, by randomly selecting from the set of gates $U = \{Z(p), CZ(s), V(\gamma)\}$;
3. and finally, homodyne measurements are performed on all modes in the $\hat{p}$ quadrature.

```
import strawberryfields as sf
from strawberryfields.ops import *

# initialise the engine and register
eng, q = sf.Engine(4)

with eng:
  # prepare the input squeezed states
  S = Sgate(-1)
  S | q[0]
  S | q[1]
  S | q[2]
  S | q[3]

  # gates diagonal in the x quadrature
  CZgate(0.5719) | (q[0], q[1])
  Vgate(-1.9782) | q[2]
  Zgate(2.0603)  | q[3]
  Zgate(-0.7804) | q[0]
  Zgate(0.8578)  | q[2]
  CZgate(1.321)  | (q[1], q[2])
  Zgate(0.473)   | q[3]
  CZgate(0.9946) | (q[0], q[2])
  Zgate(0.1223)  | q[3]
  Vgate(0.6157)  | q[0]
  Vgate(0.3110)  | q[1]
  Zgate(-1.243)  | q[2]
```

```
  # end circuit

# run the engine
eng.run("fock", cutoff_dim=5)
```

Codeblock 19: *4-mode instantaneous quantum polynomial (IQP) example in Strawberry Fields.*

Moreover, the resulting probability distributions have been shown to be given by integrals of oscillating functions in large dimensions, which are believed to be intractable to compute by classical computers. This leads to the result that even in the case of finite squeezing and reduced measurement precision, approximate sampling from the output CV-IQP model remains classically intractable [17, 93].

Hamiltonian Simulation

The simulation of atoms, molecules and other biochemical systems is another application uniquely suited to quantum computation. For example, the ground state energy of large systems, the dynamical behaviour of an ensemble of molecules, or complex molecular behaviour such as protein folding, are often computationally hard or downright impossible to determine via classical computation or experimentation [94, 95].

In the discrete-variable qubit model, efficient methods of Hamiltonian simulation have been discussed at length, providing several implementations depending on properties of the Hamiltonian, and resulting in a linear simulation time [96, 97]. Efficient implementations of Hamiltonian simulation also exist in the CV formulation [98], with specific application to Bose-Hubbard Hamiltonians (describing a system of interacting bosonic particles on a lattice of orthogonal position states [99]). As such, this method is ideally suited to photonic quantum computation.

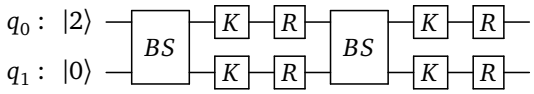

FIG. 8: *Bose-Hubbard Hamiltonian simulation for a 2-node lattice, with $k = 2$. The error is of the order $\mathcal{O}(t^2/k)$.*

Considering a lattice composed of two adjacent nodes characterised by Adjacency matrix A, the Bose-Hubbard Hamiltonian is given by

$$\begin{aligned} H &= J\sum_i\sum_j A_{ij}\hat{a}_i^\dagger\hat{a}_j + \frac{1}{2}U\sum_i \hat{n}_i(\hat{n}_i - 1) \\ &= J(\hat{a}_1^\dagger\hat{a}_2 + \hat{a}_2^\dagger\hat{a}_1) + \frac{1}{2}U(\hat{n}_1^2 - \hat{n}_1 + \hat{n}_2^2 - \hat{n}_2), \end{aligned} \tag{C3}$$

where J represents the transition of the boson between nodes, and U is the on-site interaction potential. Applying the Lie product formula, we find that

$$\begin{aligned} e^{iHt} = \Big[&\exp\left(-i\frac{Jt}{k}(\hat{a}_1^\dagger\hat{a}_2 + \hat{a}_2^\dagger\hat{a}_1)\right)\exp\left(-i\frac{Ut}{2k}\hat{n}_1^2\right) \\ &\exp\left(-i\frac{Ut}{2k}\hat{n}_2^2\right)\exp\left(i\frac{Ut}{2k}\hat{n}_1\right)\exp\left(i\frac{Ut}{2k}\hat{n}_2\right)\Big]^k \\ &+ \mathcal{O}\left(t^2/k\right), \end{aligned} \tag{C4}$$

where $\mathcal{O}\left(t^2/k\right)$ is the order of the error term, derived from the Lie product formula. Comparing this to the form of various gates in the CV circuit model, we can write this as the product of beamsplitters, Kerr gates, and rotation gates:

$$e^{iHt} = \{BS(\theta,\phi)[K(r)R(-r)\otimes K(r)R(-r)]\}^k + \mathcal{O}\left(t^2/k\right) \tag{C5}$$

where $\theta = -Jt/k$, $\phi = \pi/2$, and $r = -Ut/2k$. Using $J = 1$, $U = 1.5$, $k = 20$, and a timestep of $t = 1.086$, this can be easily implemented in Strawberry Fields using only $20\times 3 = 60$ gates.

```
import strawberryfields as sf
from strawberryfields.ops import *
from numpy import pi

# initialise the engine and register
eng, q = sf.Engine(2)

# set the Hamiltonian parameters
J = 1           # hopping transition
U = 1.5         # on-site interaction
k = 20          # Lie product decomposition
 ↪  terms
t = 1.086       # timestep
theta = -J*t/k
r = -U*t/(2*k)

with eng:
  # prepare the initial state
  Fock(2) | q[0]

  # Two node tight-binding
  # Hamiltonian simulation

  for i in range(k):
    BSgate(theta, pi/2) | (q[0], q[1])
    Kgate(r)  | q[0]
    Rgate(-r) | q[0]
    Kgate(r)  | q[1]
    Rgate(-r) | q[1]
  # end circuit

# run the engine
state = eng.run("fock", cutoff_dim=5)
# the output state probabilities
print(state.fock_prob([0,2]))
print(state.fock_prob([1,1]))
print(state.fock_prob([2,0]))
```

Codeblock 20: *Tight-binding Hamiltonian simulation for a 2-node lattice in Strawberry Fields.*

For more complex Hamiltonian CV decompositions, including those with nearest-neighbour, see Kalajdzievski et al. [98]. This decomposition is also implemented in the SFOpenBoson plugin [45], which provides an interface between OpenFermion, the quantum electronic structure package, and Strawberry Fields. This allows arbitrary Bose-Hubbard Hamiltonians, generated in OpenFermion, to be simulated using Strawberry Fields.

Optimization of Quantum Circuits

One of the unique features of Strawberry Fields is that it has been designed from the start to support modern computational methods like automatic gradients, optimization, and machine learning by leveraging a TensorFlow backend. We have already given an overview in the main text of how these features are accessed in Strawberry Fields. We present here a complete example for the optimization of a quantum circuit. Our goal in this circuit is to find the Dgate parameter which leads to the highest probability of a $n = 1$ Fock state output. This simple baseline example can be straightforwardly extended to much more complex circuits. As optimization is a key ingredient of machine learning, this example can also serve as a springboard for more advanced data-driven modelling tasks.

We note that optimization here is in a *variational* sense, i.e., we choose an ansatz for the optimization by fixing the discrete structure of our circuits (the selection and arrangement of specific states, gates, and measurements). The optimization then proceeds over the parameters of these operations. Finally, we emphasize that for certain circuits and objective functions, the optimization might be non-convex in general. Thus we should not assume (without proof) that the solution obtained via a Strawberry Fields optimization is always the global optimum. However, it is often the case in machine learning that local optima can still provide effective solutions, so this may not be an issue, depending on the application.

```
import strawberryfields as sf
from strawberryfields.ops import *
import tensorflow as tf

eng, q = sf.Engine(1)

alpha = tf.Variable(0.1)
with eng:
  Dgate(alpha) | q[0]
state = eng.run("tf", cutoff_dim=7,
 ↪  eval=False)

# loss is probability for the Fock state n=1
prob = state.fock_prob([1])
loss = -prob  # negative sign to maximize prob

# Set up optimization
optimizer = tf.train.GradientDescentOptimizer(
 ↪  learning_rate=0.1)
minimize_op = optimizer.minimize(loss)


# Create TF Session and initialize variables
sess = tf.Session()
sess.run(tf.global_variables_initializer())

# Carry out optimization
for step in range(50):
  prob_val, _ = sess.run([prob, minimize_op])
  print("Value at step {}: {}".format(step,
   ↪  prob_val))
```

Codeblock 21: *Example of optimizing quantum circuit parameters in Strawberry Fields. This can be extended to a machine learning setting by incorporating data and a more sophisticated loss function.*

[1] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999. doi: 10.1137/S0036144598347011.

[2] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*, pages 212–219. ACM, 1996.

[3] A. W. Harrow, A. Hassidim, and S. Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103 (15):150502, 2009. doi:10.1103/PhysRevLett.103.150502.

[4] J. Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, 2018. doi:10.22331/q-2018-08-06-79.

[5] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5, 2014. doi:10.1038/ncomms5213.

[6] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016. doi:10.1088/1367-2630/18/2/023023.

[7] E. Farhi, J. Goldstone, and S. Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

[8] E. Farhi and A. W. Harrow. Quantum supremacy through the quantum approximate optimization algorithm. *arXiv preprint arXiv:1602.07674*, 2016.

[9] S. Aaronson and A. Arkhipov. The computational complexity of linear optics. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 333–342. ACM, 2011. doi:10.1145/1993636.1993682.

[10] C. S. Hamilton, R. Kruse, L. Sansoni, S. Barkhofen, C. Silberhorn, and I. Jex. Gaussian boson sampling. *Physical Review Letters*, 119:170501, 2017. doi: 10.1103/PhysRevLett.119.170501.

[11] L. Chakhmakhchyan, R. Garcia-Patron, and N. J. Cerf. Boson sampling with Gaussian measurements. *Physical Review A*, 96:032326, 2017. doi:10.1103/PhysRevA.96.032326.

[12] M. J. Bremner, R. Jozsa, and D. J. Shepherd. Classical simulation of commuting quantum computations implies collapse of the polynomial hierarchy. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, page rspa20100301. The Royal Society, 2010. doi: 10.1098/rspa.2010.0301.

[13] M. J. Bremner, A. Montanaro, and D. J. Shepherd. Average-case complexity versus approximate simulation of commuting quantum computations. *Physical Review Letters*, 117(8): 080501, 2016. doi:10.1103/PhysRevLett.117.080501.

[14] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M. J. Bremner, J. M. Martinis, and H. Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595, 2018. doi:10.1038/s41567-018-0124-x.

[15] S. Aaronson and L. Chen. Complexity-Theoretic Foundations of Quantum Supremacy Experiments. In R. O'Donnell, editor, *32nd Computational Complexity Conference (CCC 2017)*, volume 79 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:67. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. ISBN 978-3-95977-040-8. doi:10.4230/LIPIcs.CCC.2017.22.

[16] C. Neill, P. Roushan, K. Kechedzhi, S. Boixo, S. V. Isakov, V. Smelyanskiy, A. Megrant, B. Chiaro, A. Dunsworth, K. Arya, et al. A blueprint for demonstrating quantum supremacy with superconducting qubits. *Science*, 360 (6385):195–199, 2018. doi:10.1126/science.aao4309.

[17] T. Douce, D. Markham, E. Kashefi, E. Diamanti, T. Coudreau, P. Milman, P. van Loock, and G. Ferrini. Continuous-variable instantaneous quantum computing is hard to sample. *Physical Review Letters*, 118(7), 2017. doi: 10.1103/PhysRevLett.118.070503.

[18] A. Finnila, M. Gomez, C. Sebenik, C. Stenson, and J. Doll. Quantum annealing: a new method for minimizing multidimensional functions. *Chemical Physics Letters*, 219(5-6): 343–348, 1994. doi:10.1016/0009-2614(94)00117-0.

[19] M. W. Johnson, M. H. Amin, S. Gildert, T. Lanting, F. Hamze, N. Dickson, R. Harris, A. J. Berkley, J. Johansson, P. Bunyk, et al. Quantum annealing with manufactured spins. *Nature*, 473(7346):194–198, 2011. doi:10.1038/nature10012.

[20] M.-H. Yung, J. Casanova, A. Mezzacapo, J. McClean, L. Lamata, A. Aspuru-Guzik, and E. Solano. From transistor to trapped-ion computers for quantum chemistry. *Scientific Reports*, 4:3589, 2014. doi:10.1038/srep03589.

[21] P. O'Malley, R. Babbush, I. Kivlichan, J. Romero, J. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, et al. Scalable quantum simulation of molecular energies. *Physical Review X*, 6(3):031007, 2016. doi: 10.1103/PhysRevX.6.031007.

[22] Y. Shen, X. Zhang, S. Zhang, J.-N. Zhang, M.-H. Yung, and K. Kim. Quantum implementation of the unitary coupled cluster for simulating molecular electronic structure. *Physical Review A*, 95(2):020501, 2017. doi: 10.1103/PhysRevA.95.020501.

[23] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242–246, 2017. doi: 10.1038/nature23879.

[24] G. Rosenberg, P. Haghnegahdar, P. Goddard, P. Carr, K. Wu, and M. L. de Prado. Solving the optimal trading trajectory problem using a quantum annealer. *IEEE Journal of Selected Topics in Signal Processing*, 10(6):1053–1060, 2016. doi: 10.1109/JSTSP.2016.2574703.

[25] A. Lucas. Ising formulations of many NP problems. *Frontiers in Physics*, 2:5, 2014. doi:10.3389/fphy.2014.00005.

[26] F. Neukart, D. Von Dollen, G. Compostella, C. Seidel, S. Yarkoni, and B. Parney. Traffic flow optimization using a quantum annealer. *Frontiers in ICT*, 4:29, 2017. doi: 10.3389/fict.2017.00029.

[27] H. Neven, V. S. Denchev, G. Rose, and W. G. Macready. Training a large scale classifier with the quantum adiabatic algorithm. *arXiv preprint arXiv:0912.0779*, 2009.

[28] K. L. Pudenz and D. A. Lidar. Quantum adiabatic machine learning. *Quantum Information Processing*, 12(5):2027–2070, 2013. doi:10.1007/s11128-012-0506-4.

[29] D. Crawford, A. Levit, N. Ghadermarzy, J. S. Oberoi, and P. Ronagh. Reinforcement learning using quantum Boltzmann machines. *Quantum Information & Computation*, 18 (1-2):0051–0074, 2018. doi:10.26421/QIC18.1-2.

[30] M. H. Amin, E. Andriyash, J. Rolfe, B. Kulchytskyy, and R. Melko. Quantum boltzmann machine. *Phys. Rev. X*, 8: 021050, May 2018. doi:10.1103/PhysRevX.8.021050.

[31] D. Ristè, M. P. Da Silva, C. A. Ryan, A. W. Cross, A. D. Córcoles, J. A. Smolin, J. M. Gambetta, J. M. Chow, and B. R. Johnson. Demonstration of quantum advantage in machine learning. *npj Quantum Information*, 3(1):16, 2017. doi: 10.1038/s41534-017-0017-3.

[32] G. Verdon, M. Broughton, and J. Biamonte. A quantum algorithm to train neural networks using low-depth circuits. *arXiv preprint arXiv:1712.05304*, 2017.

[33] J. Otterbach, R. Manenti, N. Alidoust, A. Bestwick, M. Block, B. Bloom, S. Caldwell, N. Didier, E. S. Fried, S. Hong, et al. Unsupervised machine learning on a hybrid quantum computer. *arXiv preprint arXiv:1712.05771*, 2017.

[34] M. Schuld and N. Killoran. Quantum machine learning in feature hilbert spaces. *Physical Review Letters*, 122:040504, Feb 2019. doi:10.1103/PhysRevLett.122.040504.

[35] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Notices*, volume 48, pages 333–342. ACM, 2013. doi:10.1145/2491956.2462177.

[36] D. Wecker and K. M. Svore. Liqui|>: A software design architecture and domain-specific language for quantum computing. *arXiv preprint arXiv:1402.4467*, 2014.

[37] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi. ScaffCC: Scalable compilation and analysis of quantum programs. *Parallel Computing*, 45: 2–17, 2015. doi:10.1016/j.parco.2014.12.001.

[38] M. Smelyanskiy, N. P. Sawaya, and A. Aspuru-Guzik. qHiPSTER: the quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195*, 2016.

[39] S. Pakin. A quantum macro assembler. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–8. IEEE, 2016. doi:10.1109/HPEC.2016.7761637.

[40] R. S. Smith, M. J. Curtis, and W. J. Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[41] D. S. Steiger, T. Häner, and M. Troyer. ProjectQ: an open source software framework for quantum computing. *Quantum*, 2:49, 2018. doi:10.22331/q-2018-01-31-49.

[42] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[43] E. S. Fried, N. P. Sawaya, Y. Cao, I. D. Kivlichan, J. Romero, and A. Aspuru-Guzik. qtorch: The quantum tensor contraction handler. *PloS one*, 13(12):e0208510, 2018. doi: 10.1371/journal.pone.0208510.

[44] A. J. McCaskey, E. F. Dumitrescu, D. Liakh, M. Chen, W.-c.

Feng, and T. S. Humble. Extreme-scale programming model for quantum acceleration within high performance computing. *arXiv preprint arXiv:1710.01794*, 2017.

[45] J. R. McClean, I. D. Kivlichan, D. S. Steiger, Y. Cao, E. S. Fried, C. Gidney, T. Häner, V. Havlíček, Z. Jiang, M. Neeley, et al. OpenFermion: The electronic structure package for quantum computers. *arXiv preprint arXiv:1710.07629*, 2017.

[46] S. Liu, X. Wang, L. Zhou, J. Guan, Y. Li, Y. He, R. Duan, and M. Ying. $Q|SI\rangle$: A quantum programming environment. In *Symposium on Real-Time and Hybrid Systems*, pages 133–164. Springer, 2018. doi:10.1007/978-3-030-01461-2_8.

[47] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, page 7. ACM, 2018. doi:10.1145/3183895.3183901.

[48] S. Lloyd and S. L. Braunstein. Quantum computation over continuous variables. *Physical Review Letters*, 82(8):1784, 1999. doi:10.1103/PhysRevLett.82.1784.

[49] S. L. Braunstein and P. van Loock. Quantum information with continuous variables. *Reviews of Modern Physics*, 77:513–577, 2005. doi:10.1103/RevModPhys.77.513.

[50] C. Weedbrook, S. Pirandola, R. García-Patrón, N. J. Cerf, T. C. Ralph, J. H. Shapiro, and S. Lloyd. Gaussian quantum information. *Reviews of Modern Physics*, 84(2):621, 2012. doi:10.1103/RevModPhys.84.621.

[51] M. A. Nielsen and I. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2002.

[52] A. Graves, G. Wayne, and I. Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[53] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, 2016. doi:10.1038/nature20101.

[54] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. doi:10.1109/MCSE.2011.37.

[55] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[56] N. Quesada. A faster calculation of franck-condon factors and fock matrix elements of gaussian unitaries using loop hafnians. *arXiv preprint arXiv:1811.09597*, 2018.

[57] D. Gottesman, A. Kitaev, and J. Preskill. Encoding a qubit in an oscillator. *Physical Review A*, 64(1):012310, 2001. doi:10.1103/PhysRevA.64.012310.

[58] A. Ferraro, S. Olivares, and M. G. Paris. Gaussian states in continuous variable quantum information. *arXiv preprint quant-ph/0503237*, 2005.

[59] J. Williamson. On the algebraic problem concerning the normal forms of linear dynamical systems. *American Journal of Mathematics*, 58(1):141, 1936. doi:10.2307/2371062.

[60] C. Bloch and A. Messiah. The canonical form of an antisymmetric tensor and its application to the theory of superconductivity. *Nuclear Physics*, 39:95–106, 1962. doi:10.1016/0029-5582(62)90377-2.

[61] S. L. Braunstein. Squeezing as an irreducible resource. *Physical Review A*, 71(5):055801, 2005. doi:10.1103/PhysRevA.71.055801.

[62] R. Simon, N. Mukunda, and B. Dutta. Quantum-noise matrix for multimode systems: $U(n)$ invariance, squeezing, and normal forms. *Physical Review A*, 49(3):1567, 1994. doi:10.1103/PhysRevA.49.1567.

[63] W. R. Clements, P. C. Humphreys, B. J. Metcalf, W. S. Kolthammer, and I. A. Walsmley. Optimal design for universal multiport interferometers. *Optica*, 3(12):1460–1465, 2016. doi:10.1364/OPTICA.3.001460.

[64] J. M. Arrazola, T. R. Bromley, J. Izaac, C. R. Myers, K. Bradler, and N. Killoran. Machine learning method for state preparation and gate synthesis on photonic quantum computers. *Quantum Science and Technology*, 2018. doi:10.1088/2058-9565/aaf59e.

[65] N. Quesada and A. M. Brańczyk. Gaussian functions are optimal for waveguided nonlinear-quantum-optical processes. *Physical Review A*, 98:043813, 2018. doi:10.1103/PhysRevA.98.043813.

[66] N. Killoran, T. R. Bromley, J. M. Arrazola, M. Schuld, N. Quesada, and S. Lloyd. Continuous-variable quantum neural networks. *arXiv preprint arXiv:1806.06871*, 2018.

[67] M. Reck, A. Zeilinger, H. J. Bernstein, and P. Bertani. Experimental realization of any discrete unitary operator. *Physical Review Letters*, 73:58–61, 1994. doi:10.1103/PhysRevLett.73.58.

[68] A. Serafini. *Quantum Continuous Variables: A Primer of Theoretical Methods*. CRC Press, 2017.

[69] S. D. Bartlett, B. C. Sanders, S. L. Braunstein, and K. Nemoto. Efficient classical simulation of continuous variable quantum information processes. *Physical Review Letters*, 88:097904, 2002. doi:10.1103/PhysRevLett.88.097904.

[70] A. Lund, A. Laing, S. Rahimi-Keshari, T. Rudolph, J. L. O'Brien, and T. Ralph. Boson sampling from a Gaussian state. *Physical Review Letters*, 113(10):100502, 2014. doi:10.1103/PhysRevLett.113.100502.

[71] G. Cariolaro and G. Pierobon. Bloch-Messiah reduction of Gaussian unitaries by Takagi factorization. *Physical Review A*, 94(6):062109, 2016. doi:10.1103/PhysRevA.94.062109.

[72] G. Cariolaro and G. Pierobon. Reexamination of Bloch-Messiah reduction. *Physical Review A*, 93(6):062115, 2016. doi:10.1103/PhysRevA.93.062115.

[73] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters*, 70:1895–1899, 1993. doi:10.1103/PhysRevLett.70.1895.

[74] A. Furusawa and P. van Loock. *Quantum Teleportation and Entanglement: A Hybrid Approach to Optical Quantum Information Processing*. Wiley, 2011.

[75] W. Steeb and Y. Hardy. *Problems and Solutions in Quantum Computing and Quantum Information*. World Scientific, 2006.

[76] M. Gu, C. Weedbrook, N. C. Menicucci, T. C. Ralph, and P. van Loock. Quantum computing with continuous-variable clusters. *Physical Review A*, 79:062318, 2009. doi:10.1103/PhysRevA.79.062318.

[77] D. Gottesman and I. L. Chuang. Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations. *Nature (London)*, 402:390–393, 1999. doi:10.1038/46503.

[78] S. D. Bartlett and W. J. Munro. Quantum teleportation of optical quantum gates. *Physical Review Letters*, 90:117901, 2003. doi:10.1103/PhysRevLett.90.117901.

[79] M. Tillmann, B. Dakić, R. Heilmann, S. Nolte, A. Szameit, and P. Walther. Experimental boson sampling. *Nature Photonics*, 7(7):540–544, 2013. doi:10.1038/nphoton.2013.102.

[80] N. Spagnolo, C. Vitelli, M. Bentivegna, D. J. Brod, A. Crespi, F. Flamini, S. Giacomini, G. Milani, R. Ramponi, P. Mataloni, R. Osellame, E. F. Galvão, and F. Sciarrino. Experimental validation of photonic boson sampling. *Nature Photonics*, 8 (8):615–620, 2014. doi:10.1038/nphoton.2014.135.

[81] A. Crespi, R. Osellame, R. Ramponi, D. J. Brod, E. F. Galvão, N. Spagnolo, C. Vitelli, E. Maiorino, P. Mataloni, and F. Sciarrino. Integrated multimode interferometers with arbitrary designs for photonic boson sampling. *Nature Photonics*, 7(7): 545–549, 2013. doi:10.1038/nphoton.2013.112.

[82] J. B. Spring, B. J. Metcalf, P. C. Humphreys, W. S. Kolthammer, X.-M. Jin, M. Barbieri, A. Datta, N. Thomas-Peter, N. K. Langford, D. Kundys, J. C. Gates, B. J. Smith, P. G. R. Smith, and I. A. Walmsley. Boson sampling on a photonic chip. *Science*, 339(6121):798–801, 2012. doi: 10.1126/science.1231692.

[83] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979. doi: 10.1016/0304-3975(79)90044-6.

[84] E. R. Caianiello. *Combinatorics and renormalization in quantum field theory*, volume 38. Benjamin, 1973.

[85] A. Barvinok. *Combinatorics and complexity of partition functions*, volume 274. Springer, 2016.

[86] K. Brádler, P.-L. Dallaire-Demers, P. Rebentrost, D. Su, and C. Weedbrook. Gaussian boson sampling for perfect matchings of arbitrary graphs. *Physical Review A*, 98:032310, Sep 2018. doi:10.1103/PhysRevA.98.032310.

[87] A. Björklund, B. Gupt, and N. Quesada. A faster hafnian formula for complex matrices and its benchmarking on the titan supercomputer. *arXiv preprint arXiv:1805.12498*, 2018.

[88] N. Quesada, J. M. Arrazola, and N. Killoran. Gaussian boson sampling using threshold detectors. *Physical Review A*, 98: 062322, 2018. doi:10.1103/PhysRevA.98.062322.

[89] B. Gupt, J. M. Arrazola, N. Quesada, and T. R. Bromley. Classical benchmarking of gaussian boson sampling on the titan supercomputer. *arXiv preprint arXiv:1810.00900*, 2018.

[90] D. Shepherd and M. J. Bremner. Temporally unstructured quantum computation. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 465(2105):1413–1439, 2009. doi: 10.1098/rspa.2008.0443.

[91] M. J. Bremner, A. Montanaro, and D. J. Shepherd. Achieving quantum supremacy with sparse and noisy commuting quantum computations. *Quantum*, 1:8, 2017. doi:10.22331/q-2017-04-25-8.

[92] A. P. Lund, M. J. Bremner, and T. C. Ralph. Quantum sampling problems, BosonSampling and quantum supremacy. *npj Quantum Information*, 3(1), 2017. doi:10.1038/s41534-017-0018-2.

[93] J. M. Arrazola, P. Rebentrost, and C. Weedbrook. Quantum supremacy and high-dimensional integration. *arXiv preprint arXiv:1712.07288*, 2017.

[94] A. Aspuru-Guzik, A. D. Dutoi, P. J. Love, and M. Head-Gordon. Simulated quantum computation of molecular energies. *Science*, 309(5741):1704–1707, 2005. doi: 10.1126/science.1113479.

[95] J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics*, 109(5):735–750, 2011. doi: 10.1080/00268976.2011.552441.

[96] A. M. Childs and N. Wiebe. Hamiltonian simulation using linear combinations of unitary operations. *Quantum Information and Computation*, 12(11-12):901–924, 2012. doi: 10.26421/QIC12.11-12.

[97] D. W. Berry, G. Ahokas, R. Cleve, and B. C. Sanders. Efficient quantum algorithms for simulating sparse Hamiltonians. *Communications in Mathematical Physics*, 270(2):359–371, 2006. doi:10.1007/s00220-006-0150-x.

[98] T. Kalajdzievski, C. Weedbrook, and P. Rebentrost. Continuous-variable gate decomposition for the bose-hubbard model. *Physical Review A*, 97:062311, Jun 2018. doi:10.1103/PhysRevA.97.062311.

[99] T. Sowiński, O. Dutta, P. Hauke, L. Tagliacozzo, and M. Lewenstein. Dipolar molecules in optical lattices. *Physical Review Letters*, 108:115301, 2012. doi: 10.1103/PhysRevLett.108.115301.