МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тульский государственный университет»

Институт горного дела и строительства
Кафедра «Приборы управления»

Утверждено на заседании кафедры
«Приборы управления»
«22» января 2024 г., протокол №1

Заведующий кафедрой

_____ В.В. Матвеев

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по проведению практических (семинарских) занятий по дисциплине (модулю)

«Оптическая информатика»

основной профессиональной образовательной программы высшего образования – программы бакалавриата

по направлению подготовки

12.03.03 Фотоника и оптоинформатика

с направленностью (профилем)

Интеллектуальные фотонные системы

Формы обучения: очная

Идентификационный номер образовательной программы: 120303-01-24

Тула 2024 год

Разработчик методических указаний

Заведующий кафедрой, д.т.н., доцент  В.В. Матвеев

СОДЕРЖАНИЕ

Практическая работа №1. Основы языка программирования *Blackbird*

Установка грамматики

Примечание

Это требуется только в том случае, если вы хотите автоматически сгенерировать новые анализаторы *Blackbird* для дополнительных целевых языков или если вы хотите изменить грамматику *Blackbird* напрямую.

Если вместо этого вы хотите использовать существующие готовые анализаторы *Python / C ++ Blackbird*, ознакомьтесь с отдельными деталями установки для установки анализатора Python и компиляции и установки анализатора C ++.

Компиляция грамматики

Примечание

Этот шаг необязателен - *Blackbird* поставляется с уже предварительно скомпилированной грамматикой *ANTLR* как для *C ++*, так и для *Python*.

Для компиляции грамматики вам необходимо убедиться, что установлена *Java* версии 1.6 и выше. После установки *Java* выполните следующие действия для установки *ANTLR* 4.9.2:

```
$ cd /usr/local/lib
$ sudo wget https://www.antlr.org/download/antlr-4.9.2-complete.jar
$ export CLASSPATH=".:/usr/local/lib/antlr-4.9.2-complete.jar:$CLASSPATH"
$ alias antlr4='java -jar /usr/local/lib/antlr-4.9.2-complete.jar'
$ alias grun='java org.antlr.v4.gui.TestRig'
```

Затем вы можете автоматически сгенерировать слушателей и посетителей на базе *Python* и *C ++*, перейдя в корневой каталог *Blackbird* и запустив *make grammar*.

Документация

Для сборки документации требуются следующие дополнительные пакеты:

Sphinx >=1.5

sphinxcontrib-bibtex >=0.3.6

Они оба могут быть установлены через *pip*:

```
$ python3 -m pip install sphinx sphinxcontrib-bibtex
```

Чтобы создать *HTML*-документацию, перейдите в каталог верхнего уровня и запустите

```
$ make docs
```

Затем документацию можно найти в каталоге *doc/_build/html/* .

Синтаксис и грамматика

Примечание

Blackbird, используемый непосредственно внутри контекста движка *Strawberry Fields Engine*, является надмножеством описанного здесь языка ассемблера *Blackbird quantum*, поскольку он встроен в среду *Python*.

Вы можете думать об этом как о "*Blackbird* с улучшенным *Python*', поскольку можно использовать функции и конструкции Python, даже если они не являются 'официальной' частью спецификации *Blackbird*.

Введение

В этом разделе мы определяем структуру, синтаксис и грамматику кода *Blackbird*.

Философия

Blackbird был разработан с нуля, придерживаясь следующих принципов:

Инкапсулируйте любые универсальные фотонно-квантовые вычисления.

Будьте ясны, лаконичны и просты для чтения и следования. Эта простота должна позволять

-Удобочитаемость для человека - операции и выражения должны соответствовать существующим соглашениям и ссылаться на общепринятые обозначения в квантовых вычислениях

-Аппаратное выполнение - код должен быть однозначным, с одной квантовой операцией на строку

Быть простым в освоении, используя конструкции и операторы, знакомые по научным вычислениям.

Скрипт *Blackbird* должен содержать только один квантовый алгоритм или симуляцию, что делает его идеальным форматом для сохранения и загрузки алгоритмов фотонных квантовых вычислений.

Сходство с *Python*

Чтобы удовлетворить пунктам 2 и 3, *Blackbird* намеренно написан на *Python*, наследуя следующее:

-Чувствительность к регистру.

-# для комментариев к строкам.

-Новые строки указывают конец инструкции.

-Операторы и литералы аналогичны их эквивалентам в *Python*.

-Оператор | используется для применения встроенных квантовых операций к квантовым регистрам.

-После измерения квантовые режимы автоматически и неявно преобразуются в классические регистры.

-Результирующий результат неявно определяется наличием инструкций измерения.

Отличия от *Python*

Однако, в отличие от *Python*, мы также вводим следующие ограничения, чтобы *Blackbird* мог функционировать как язык квантового ассемблера на широком спектре квантового оборудования:

- Статически типизированный - вы должны объявить тип переменной, а переменные и аргументы конфликтующих типов не автоматически приводятся к правильному типу.

- Переменные массива могут быть объявлены, а доступ к отдельным элементам осуществляется посредством индексации, но *Blackbird* не поддерживает манипулирование массивом.

Метаданные

Все программы *Blackbird* должны начинаться с требуемого блока метаданных следующим образом:

```
name progam_name
version 1.0
```

name и *version* ключевые слова должны быть указаны в указанном выше порядке и являются обязательными. Название просто указывает название вашей программы *Blackbird*, а номер версии - это номер версии спецификации *Blackbird*, для которой она написана.

Кроме того, вы можете указать необязательное *target* ключевое слово:

```
target chip0
```

Это указывает на устройство или симулятор, на который нацелена программа *Blackbird*, то есть вы указываете, что содержащийся код *Blackbird* скомпилирован для целевого устройства / симулятора.

Кроме того, тип программы может быть указан с помощью необязательного type ключевого слова:

```
type tdm (temporal_modes=42, copies=1000)
```

где *TDM* будет соответствовать запуску эксперимента по мультиплексированию во временной области. Если метаданные типа программы опущены, предполагается программа выборки гауссовских бозонов по умолчанию.

Ключевые слова target и type также принимают параметры ключевых слов, используя синтаксис (*option*1=0.32, *option*2=40). Например, *copies*=1000 выше или:

```
target chip0 (shots=100)
```

Объявления переменных

Переменная может быть опционально определена в любой новой строке в скрипте *blackbird*.

Синтаксис для определения переменных следующий:

```
type name = expression
```

поддерживаются следующие типы:

- *int*: 0, 1, 5
- *float*: 8.0, 0.43, -0.123, 89.23e-10
- *complex*: 0+5j, 8.1-1j, 0.54+0.21j
- *bool: True, False*
- *str*: любая строка *ASCII*, заключенная в двойные кавычки, "*hello world*"

Примечание

- При использовании числа с плавающей запятой вы должны указать полное десятичное число. Т.е., 8 и 8. не являются допустимыми числами с плавающей запятой, но 8.0 есть.

- При использовании комплекса необходимо указывать как действительную, так и мнимую части. Т.е., 8 и 2*j* не являются допустимыми комплексными литералами, но 8+0*j* есть.

Примеры

```
int n = +5
int k = n

float m = -0.5432
float alpha = 0.5432
float x = 0.5+0.1
float Delta = 0.543

complex beta = 5.21
```

```
complex y = -0.43e-4+0.912j
complex z = +0.43e-4-0.912j

bool flag = True
str name = "program1"
```

Предупреждение

Разрешены все имена переменных, начинающиеся с буквы, кроме тех, которые состоят из одной буквы ‘*q*’, за которой следует целое число, например *q0, q1, q2* и т.д. Они зарезервированы для ссылок на квантовые регистры.

Операторы

Blackbird допускает выражения, использующие следующие операторы:

+: сложение, унарный положительный результат

-: вычитание, унарное отрицание

*: умножение

/: разделение

**: правоассоциативное возведение в степень.

Функции

Blackbird также поддерживает встроенные функции

sqrt()

exp(), log

sin(), cos(), tan()

arcsin(), arccos(), arctan()

sinh(), cosh(), tanh()

arcsinh(), arccosh(), arctanh()

и внутренняя константа *pi*

Вы также можете использовать ранее определенные имена переменных в своих выражениях:

```
float gamma = 2.0*cos(alpha*pi)
float test = n**2.0
```

Массивы

Чтобы определить массивы, укажите '*array*' после типа переменной. Затем каждая строка массива определяется в строке с отступом, столбцы разделяются запятыми.

```
float array A =
    -1.0, 2.0
    -0.1, 0.2

complex array U[3, 3] =
    -0.23191638+0.17828953j,0.58457815+0.41415933j,-0.05795454-0.46965132j
    +0.42259383+0.56368926j,-0.42219920+0.04735544j,-0.18902308-
0.01590913j
    -0.02396850+0.64301446j,0.09918161+0.36797446j,0.26993055+0.30341975j
```

Массивы поддерживают извлечение значений посредством линейной индексации. Например, *U*[4] будет соответствовать четвертому значению в приведенном выше массиве, если его сгладить, таким образом возвращая +0.42259383+0.56368926j.

Примечание

Для дополнительной проверки массива вы можете указать форму массива, используя квадратные скобки непосредственно после имени переменной (т.е. *U*[3, 3]), но это необязательно.

Программа *Quantum*

Оператор | используется для применения встроенных квантовых операций к квантовым регистрам.

Например:

```
# Statements have the following form:
Operation(parameters) | modes

# Depending on the operation, parameters may be optional
# Parameters can be variables of literals or expressions
complex alpha = 0.5+0.2
float delta = 0.5423
Coherent(alpha**2, Delta*sqrt(pi)) | 0
```

```
# Multiple modes are specified by comma separated integers
Interferometer(U) | [0, 1, 2, 3]

# Finish with measurements
MeasureFock(dark_counts=[0.1, 0.2]) | [0, 1]
```

В настоящее время устройство всегда принимает аргументы по ключевым словам, а операции принимают позиционные аргументы и аргументы по ключевым словам.

Чтобы передать измеренные значения режима в последовательные аргументы ворот, вы можете использовать зарезервированные переменные *qX*, где *X* — это целое число, представляющее режим *X*, в качестве параметров:

```
S2gate(0.43, 0.12) | [0, 1]
MeasureX | 0
MeasureP | 1
Xgate(sqrt(2)*q0+q1) | 2
```

После запуска программы *Blackbird* пользователь должен ожидать получения результатов в виде массива:

- каждый столбец представляет собой результат измерения, соответствующий измерениям в порядке их отображения в программе *blackbird*

- каждая строка представляет собой выстрел / прогон

For-циклы

Подобно *Python*, циклы for могут быть объявлены с использованием *for ... in ...* синтаксиса, за которым следуют строки операторов с отступом. Обратите внимание, что в конце оператора *for* нет двоеточия (:). Тип переменной цикла *for* должен быть объявлен, за которым должен следовать либо список значений указанного типа, либо диапазон с использованием синтаксиса *from:to:step*.

Например:

```
for int i in [0, 2, 1, 0, 2, 1]
    MZgate(phases[i], phases[i+1]) | [i, i+1]
```

где phases может быть массив, объявленный выше, или:

```
for int m in 2:10:2
    MeasureX | m
```

измерение в режимах 2, 4, 6 и 8.

Примечание

В настоящее время не поддерживаются следующие:

- Вложенные циклы for; допускаются только одиночные циклы for,

- Перебор массивов, например, *for int i in phases*.

Шаблоны

Шаблон *Blackbird* - это просто скрипт *Blackbird*, который содержит параметры шаблона.

Параметры шаблона используют синтаксис {*parameter_name*} и могут быть помещены в любое числовое выражение.

Для примера рассмотрим следующий шаблон телепортации состояния:

```
name StateTeleportation
version 1.0

# state to be teleported:
Coherent({alpha}) | 0

# teleportation algorithm
Squeezed(-{sq}) | 1
Squeezed({sq}) | 2
BSgate(pi/4, 0) | (1, 2)
BSgate(pi/4, 0) | (0, 1)
MeasureX | 0
MeasureP | 1
Xgate(sqrt(2)*q0) | 2
Zgate(sqrt(2)*q1) | 2
```

Здесь для подготовки начального состояния используется параметр шаблона {*alpha*}, в то время как значения состояний сжатых ресурсов задаются параметром {*sq*}.

Преимущество шаблонов Blackbird заключается в том, что скрипт *Blackbird* может инкапсулировать фотонную квантовую схему со свободными параметрами. Библиотека, использующая язык ассемблера *Blackbird quantum* (например, *Strawberry Fields*), может динамически обновлять параметры шаблона без необходимости перекомпиляции программы.

Включая подпрограммы

Может быть случай, когда у вас есть программа *Blackbird* или шаблон, представляющий примитив *circuit*, который вы, возможно, захотите повторно использовать в нескольких программах *Blackbird.*

Это возможно с помощью оператора *include*. Он имеет следующий синтаксис:

```
include "path/to/filename.xbb"
```

где путь к файлу указан относительно местоположения текущего скрипта *Blackbird.* Скрипт *Blackbird* может содержать несколько включений, и все они должны быть размещены после блока метаданных и перед определением квантовой программы / переменных.

Оператор include позволяет использовать внешнюю программу *Blackbird* в качестве подпрограммы в рамках существующего скрипта. Эта квантовая подпрограмма вызывается через name включенного скрипта Blackbird. Например, рассмотрим шаблон телепортации состояния, *state_teleportation.xbb*:

```
name StateTeleportation
version 1.0

# maximally entangled states
Squeezed(-{sq}) | 1
```

```
Squeezed({sq}) | 2
BSgate(pi/4, 0) | (1, 2)

# Alice performs the joint measurement
# in the maximally entangled basis
BSgate(pi/4, 0) | (0, 1)
MeasureX | 0
MeasureP | 1

# Bob conditionally displaces his mode
# based on Alice's measurement result
Xgate(sqrt(2)*q0) | 2
Zgate(sqrt(2)*q1) | 2
```

Этот шаблон принимает параметр *sq* (величина сжатия состояний ресурсов) и действует в трех режимах, телепортируя состояние из режима 0 в режим 2.

Теперь рассмотрим другой файл, *example_include.xbb* который включает в себя вышеупомянутую *StateTeleportation* операцию, импортированную из *state_teleportation.xbb* шаблона:

```
name ExampleInclude
version 1.0
target gaussian (shots=10)

include "state_teleportation.xbb"

float alpha = 0.3423

Coherent(a=alpha) | 0
Coherent(a=alpha) | 1
StateTeleportation(sq=1) | [0, 2, 3]
MeasureHeterodyne() | 3
```

Теперь мы можем вызвать *StateTeleportation* подпрограмму с помощью *sq*=1 и применить ее к режимам 0, 2 и 3.

Примечание

Обязательно избегайте циклических включений при использовании *include* инструкции.

Типы программ

Тип программы можно определить с помощью *type* ключевого слова в метаданных. Тип включает поддержку определенного набора экспериментов и может отличаться способом их определения внутри скрипта *Blackbird*. В настоящее время поддерживается только тип *tdm*, который расшифровывается как мультиплексирование во временной области и запускает фотонную квантовую схему в кодировании во временной области.

Мультиплексирование во временной области (*TDM)*

Для определения программы *TDM* вы можете объявить *tdm* тип в метаданных вместе с двумя различными аргументами ключевого слова: *temporal_modes*, соответствующим количеству временных интервалов, используемых в эксперименте, и *copies* определяющим, сколько раз запускается полная схема, используя каждый раз одни и те же массивы параметров.

```
type tdm (temporal_modes=2, copies=1000)
```

Для программы TDM требуется набор вентилей, которые повторяются количество раз, равное количеству временных режимов, которое определено в параметрах типа. Набор элементов управления должен быть определен только один раз, сопровождаемый массивами, содержащими параметры, которые должны использоваться в каждом цикле, также длиной, равной количеству временных режимов.

В программах *TDM* зарезервированы ключевые слова, начинающиеся с p, за которыми следует число; например, *p*0, *p*1 или *p*42. Это заполнители для параметров в соответствующих массивах (см. Пример сценария ниже). Используя это обозначение, предполагается, что каждое значение в массиве является значением параметра gate для временного режима с тем же индексным номером.

```
name tdm
version 1.0
type tdm (temporal_modes=2, copies=1000)

int array p0 =
    1, 2
int array p1 =
    3, 4

Sgate(0.7, 0) | 1
BSgate(p0, 0.0) | [0, 1]
MeasureHomodyne(phi=p1) | 0
```

В приведенном выше случае это означало бы, что *BSgate* будет использоваться первое значение в *p*0 для первого временного режима, а второе значение в *p*0 для второго временного режима. Массивы, не соответствующие этому соглашению об именовании, будут просто передаваться непосредственно в шлюз, т. е. параметром будет один и тот же массив для каждого временного режима.

Обзор *Blackbird* для *Python*

Пакет *Python Blackbird* предоставляет класс синтаксического анализа *Python* для анализа кода *Blackbird*, а также служебную функцию для автоматизации этой процедуры с учетом имени файла *Blackbird.*

Номенклатура

Анализатор

Программный компонент, который принимает входные данные и создает структуру данных. Примеры включают:

- Разбор текста *Blackbird* для формирования абстрактного синтаксического дерева, графического представления кода *Blackbird*. Это выполняется автоматически библиотекой *ANTLR*.

- Анализ абстрактного синтаксического дерева (AST) для извлечения полезных данных и построения структуры данных для обычных приложений,

таких как словарь *Python*. Для этого мы должны пройти через *AST* через слушателя или посетителя.

Прослушиватель

Языковой шаблон *ANTLR*, который автоматически пересекает *AST*. Перезаписывая методы прослушивателя, вы можете сохранять информацию в зависимости от того, где вы находитесь в дереве.

Посетитель

Похож на прослушиватель, но более гибкий. В шаблоне *visitor* вы управляете обходом *AST*, должны вручную посещать ветви. Это позволяет использовать циклы и условные выражения при обходе дерева.

Анализатор Python Blackbird построен с использованием шаблона прослушивателя ANTLR.

Модули

- blackbird.program: программный модуль Blackbird. Содержит основной программный класс Blackbird, используемый для инкапсуляции программ Blackbird в Python.

- blackbird.listener: модуль Blackbird listener. Содержит основной класс Blackbird listener, а также класс для инкапсуляции классической обработки режимов измерения в виде регистровых преобразований.

- blackbird.error: содержит анализатор ошибок для возврата полезных синтаксических ошибок пользователю.

- blackbird.auxiliary: вспомогательные функции синтаксического анализа.

Сериализация и десериализация Blackbird

Следующие функции легко доступны для

- load(): служебная функция, которая автоматизирует десериализацию скрипта Blackbird из файла (указанного по расположению файла), возвращая BlackbirdProgram объект.

- *loads():* служебная функция, которая автоматизирует десериализацию скрипта Blackbird из строки, возвращая BlackbirdProgram объект.

- *dump*(): служебная функция, которая автоматизирует сериализацию BlackbirdProgram объекта в файлоподобный объект, поддерживающий .write().

- *dumps*(): служебная функция, которая автоматизирует сериализацию BlackbirdProgram объекта в строку.

Основные классы

- BlackbirdProgram: класс, который инкапсулирует программу Blackbird, используя стандартные структуры данных Python, доступные через атрибуты.

- BlackbirdListenerPython Blackbird — прослушиватель, который просматривает абстрактное синтаксическое дерево с помощью ANTLR4, вычисляет выражения, извлекает переменные и сохраняет информацию о квантовых программах.

- Этот класс можно расширить, чтобы создавать более продвинутые слушатели Blackbird, которые выполняют действия (например, симуляции) при анализе дерева.

- RegRefTransform: класс для представления результатов измерений, обработанных классическим способом, в качестве параметров для последующих квантовых операций.

Если аргумент операции, который вы анализируете, является экземпляром этого класса, это означает, что требуется преобразование регистра.

Примечание

Все вышеперечисленные классы/функции можно импортировать из верхнего уровня пакета Blackbird Python:

```
from blackbird import BlackbirdListener, load, BlackbirdProgram
```

Краткое описание

load(имя файла)	Десериализуйте программу blackbird из файла в BlackbirdProgram объект.
loads(строка)	Десериализуйте программу blackbird из строки в BlackbirdProgram объект.
dump(blackbird, f)	Сериализуйте программу blackbird в файлоподобный объект, поддерживающий .write().
dumps(blackbird)	Сериализуйте программу blackbird в строку.

Установка

Установить анализатор Python Blackbird несложно; анализатор и все зависимости можно установить через pip:

```
$ pip install quantum-blackbird
```

В качестве альтернативы вы можете установить последнюю версию для разработки непосредственно с GitHub следующим образом. Для прослушивания Python Blackbird требуется NumPy и среда выполнения Python antlr4, которую можно установить через pip.:

```
$ pip install numpy antlr4-python3-runtime==4.9.2
```

Как только это будет установлено, просто клонируйте репозиторий Blackbird git, используйте pip для установки:

```
$ git clone https://github.com/XanaduAI/Blackbird
$ cd blackbird
$ pip install -e .
```

Пример

Эта страница содержит небольшое руководство, демонстрирующее, как использовать анализатор Python Blackbird в вашем приложении Python, чтобы включить загрузку или сохранение скриптов Blackbird.

Для начала просто импортируйте библиотеку Blackbird в модуль Python, где это требуется:

```
import blackbird
```

Десериализация Blackbird

Чтобы проанализировать и извлечь информацию из текстового файла Blackbird с расширением .xbb, просто используйте load() функцию.

Для примера рассмотрим следующий файл Blackbird:

```
name StateTeleportation
version 1.0
target gaussian (shots=1000)

# state to be teleported:
complex alpha = 1+0.5j
Coherent(alpha) | 0

# maximally entangled states
Squeezed(-4) | 1
Squeezed(4) | 2
BSgate(pi/4, 0) | (1, 2)

# Alice performs the joint measurement
# in the maximally entangled basis
BSgate(pi/4, 0) | (0, 1)
MeasureX | 0
MeasureP | 1

# Bob conditionally displaces his mode
# based on Alice's measurement result
Xgate(sqrt(2)*q0) | 2
Zgate(sqrt(2)*q1) | 2

MeasureHeterodyne() | 2
```

Чтобы импортировать это в наш модуль Python:

```
bb = blackbird.load("example/state_teleportation.xbb')
```

Возвращаемый объект bb представляет собой BlackbirdProgram, с различными атрибутами и свойствами, которые могут быть запрошены для возврата информации о программе:

- name: название программы Blackbird

- version: версия языка Blackbird, на который нацелена программа

- modes: набор неотрицательных целых чисел, определяющих номера режимов, которыми манипулирует программа

- target: словарь, содержащий информацию о целевом устройстве (т.е. О целевом устройстве, для которого компилируется скрипт Blackbird)

- operations: список операций, которые необходимо применить к устройству, во временном порядке.

Например,

```
>>> bb.name
StateTeleportation
>>> bb.operations
[{'args': [(1+0.5j)], 'kwargs': {}, 'modes': [0], 'op': 'Coherent'},
{'args': [-4], 'kwargs': {}, 'modes': [1], 'op': 'Squeezed'},
{'args': [4], 'kwargs': {}, 'modes': [2], 'op': 'Squeezed'},
{'args': [0.7853981633974483, 0],'kwargs': {},'modes': [1, 2],'op':
'BSgate'},
{'args': [0.7853981633974483, 5],'kwargs': {},'modes': [0, 1],'op':
'BSgate'},
{'modes': [0], 'op': 'MeasureX'},
{'modes': [1], 'op': 'MeasureP'},
{'args': [1.4142135623731*q0], 'kwargs': {}, 'modes': [2], 'op':
'Xgate'},
{'args': [1.4142135623731*q1], 'kwargs': {}, 'modes': [2], 'op':
'Zgate'},
{'args': [], 'kwargs': {}, 'modes': [2], 'op': 'MeasureHeterodyne'}]
```

Для получения дополнительной информации смотрите на BlackbirdProgram странице.

Примечание

Если программа Blackbird, подлежащая десериализации, представляет собой строку Python, а не имя файла, вы можете в качестве альтернативы использовать loads() функцию.

Сериализация Blackbird

BlackbirdProgram Объект, который был изменен или сконструирован, всегда можно сериализовать обратно в действующий скрипт Blackbird с помощью функций dump() и dumps().

Например, для сериализации в строку с помощью dumps():

```
>>> print(blackbird.dumps(bb))
name StateTeleportation
version 1.0
target gaussian (shots=1000)
|
Coherent(1.0+0.5j) | 0
Squeezed(-4) | 1
Squeezed(4) | 2
BSgate(0.7853981633974483, 0) | [1, 2]
BSgate(0.7853981633974483, 5) | [0, 1]
MeasureX | 0
MeasureP | 1
Xgate(1.4142135623731*q0) | 2
Zgate(1.4142135623731*q1) | 2
MeasureHeterodyne() | 2
```

Или для сериализации в файл или любой «файлоподобный» объект, поддерживающий метод write с помощью dump():

```
with open('new_file.xbb', 'w') as f:
    blackbird.dump(bb, f)
```

Шаблоны

Шаблонные скрипты Blackbird с шаблонными параметрами gate сохраняются в виде .xbb файлов. Например, рассмотрим следующий шаблон телепортации состояния:

```
name StateTeleportation
```

```
version 1.0

# state to be teleported:
Coherent({alpha}) | 0

# teleportation algorithm
Squeezed(-{sq}) | 1
Squeezed({sq}) | 2
BSgate(pi/4, 0) | (1, 2)
BSgate(pi/4, 0) | (0, 1)
MeasureX | 0
MeasureP | 1
Xgate(sqrt(2)*q0) | 2
Zgate(sqrt(2)*q1) | 2
```

Здесь для подготовки начального состояния используется параметр шаблона {alpha}, в то время как значения состояний сжатых ресурсов задаются параметром {sq}. Параметры шаблона всегда разделяются { и } и могут отображаться в любом месте выражения аргумента gate.

При десериализации этого шаблона с помощью Blackbird мы видим, что он обрабатывается как шаблон:

```
>>> bb = blackbird.load("teleportation.xbb")
>>> bb.is_template()
True
>>> bb.parameters
{'alpha', 'sq'}
```

Мы можем использовать загруженный шаблон Blackbird для создания экземпляров программ Blackbird с конкретными числовыми значениями для alpha и sq:

```
>>> bb_new = bb(alpha=0.54, sq=4)
>>> print(bb_new.serialize())
name StateTeleportation
version 1.0
target gaussian (shots=1000)
Coherent(0.54) | 0
```

```
Squeezed(-4) | 1
Squeezed(4) | 2
BSgate(0.7853981633974483, 0) | [1, 2]
BSgate(0.7853981633974483, 0) | [0, 1]
MeasureP | 1
MeasureX | 0
Zgate(1.4142135623731*q1) | 2
Xgate(1.4142135623731*q0) | 2
MeasureHeterodyne() | 2
```

Обзор Blackbird для C ++

Библиотека Blackbird C ++ предоставляет класс синтаксического анализа Blackbird для анализа кода Blackbird, а также служебную функцию для автоматизации этой процедуры с учетом имени файла Blackbird.

Основные компоненты

- Visitor: посетитель Blackbird, который анализирует абстрактное синтаксическое дерево с помощью ANTLR4, вычисляет выражения, извлекает переменные и сохраняет информацию о программе quantum.

Хотя производные классы могут быть определены для изменения действий, выполняемых при синтаксическом анализе дерева, в общем случае это не требуется. В большинстве случаев достаточно просто выполнить итерацию по возвращаемой информации, содержащейся в Program классе.

- parse_blackbird(): служебная функция, автоматизирующая синтаксический анализ абстрактного дерева синтаксиса Blackbird. Используйте эту функцию для синтаксического анализа произвольного кода Blackbird и возврата Program класса.

Обзор исходного кода

Исходный код библиотеки Blackbird C ++, содержащийся в папке blackbird_cpp, содержит следующие файлы:

- Blackbird.h: заголовок общедоступной библиотеки Blackbird. Содержит Visitor объявления, а также служебную функцию parse_blackbird(), которая автоматизирует процесс синтаксического анализа скрипта Blackbird с учетом потока файлов и возвращает программу Blackbird.

- Visitor.cpp: содержит исходный код для Visitor. Visitor является производным классом blackbirdBaseVisitor и предоставляет ANTLR4 действия для выполнения (например, вычисление выражений, хранение переменных, квантовые операции с очередями) по мере прохождения абстрактного синтаксического дерева.

Если необходимо добавить новую квантовую операцию или квантовое устройство, его инициализацию необходимо будет определить здесь, в соответствующем узле AST.

- BlackbirdProgram.h: содержит объявления для Program, Operation и всех производных классов.

- Известные устройства и вентили записаны в enum's blackbird::Device и blackbird::Gate. Они имеют то же значение, что и название команды Blackbird.

- Объявлены все элементы управления и классы измерений, включая конструкторы, количество режимов, количество параметров и проверку ввода / режима.

- Объявляется каждый класс устройства, включая конструкторы, функцию-член print_device_info и любые необходимые переменные-члены, например или размер отсечения, если применимо.

Если необходимо добавить новую квантовую операцию или квантовое устройство, объявление их класса необходимо будет сделать здесь.

- BlackbirdVariables.h: содержит шаблоны для настройки и получения назначенных переменных Blackbird из std::unordered_map, с их именем (std::string) в качестве ключа. Также включает некоторые базовые операции автоматического приведения и исключения приведения.

Автоматически созданные заголовки ANTLR4 и исходный код.

blackbirdBaseVisitor.cpp

blackbirdBaseVisitor.h

blackbirdLexer.cpp

blackbirdLexer.h

blackbirdParser.cpp

blackbirdParser.h

Предупреждение

Они создаются автоматически путем компиляции грамматики ANTLR4 src/blackbird.g4 (см. Компиляция грамматики), и их ни в коем случае не следует изменять. Если они будут изменены, ваши изменения будут перезаписаны после повторной компиляции грамматики!

Практическая работа №2. «Выборка бозонов и перманента»

1 Цель работы

Цель данной лабораторной работы – изучение применения бозонной выборки, которая является примером промежуточного квантового компьютера, предназначенного для экспериментальной реализации вычислений, считающихся неразрешимыми классически.

2 Основная теория

2.1 Теория происхождения

Бозонная дискретизация, представленная С. Ааронсоном и А. Архиповым, представляет собой небольшое отклонение от общего подхода в квантовых вычислениях.

Вместо того чтобы представлять теоретическую модель универсальных квантовых вычислений (т.е. структуру, позволяющую квантовое моделирование любого произвольного гамильтониана), устройства на основе бозонной дискретизации являются примером промежуточного квантового компьютера, предназначенного для экспериментальной реализации вычислений, которые считаются неразрешимыми классически.

Бозонная дискретизация предполагает следующую схему квантовой линейной оптики. Устанавливается массив однофотонных источников, предназначенных для одновременного излучения однофотонных состояний в многомодовый линейный интерферометр; затем результаты генерируются путем выборки из вероятности измерения однофотонных состояний с выхода линейного интерферометра.

Например, рассмотрим N-однофотонные состояния Фока, $|\psi\rangle = |m_1, m_2, \ldots, m_N\rangle$, состоящий из $b=\sum_i m_i$ фотонов, падающие на N-модовый линейный интерферометр, который выполняет следующее линейное преобразование входных операторов создания и аннигиляции мод:

$$\hat{a}^{\dagger}_{out_k} = \sum_{j=0}^{N} U_{kj} \hat{a}^{\dagger}_{in_j};$$

$$\hat{a}_{out_k} = \sum_{j=0}^{N} U^{\dagger}_{kj} \hat{a}_{in_j},$$

Здесь унитарный U полностью описывает интерферометр. Таким образом, вероятность обнаружения n_j фотонов на j-м выходном режиме задается:

$$\left|\langle n_1, n_2, \cdots, n_N | W | \psi \rangle\right|^2,$$

где W – представляет собой действие U на базисе Фока (W – просто гомоморфизм U).

Примечательный характер проблемы выборки бозонов для оспаривания расширенного тезиса Черча-Тьюринга заключается в том, что:

$$\left|\langle n_1, n_2, \cdots, n_N | W | \psi \rangle\right|^2 = \frac{\left|\mathrm{Per}\left(U_{st}\right)\right|^2}{m_1! m_2! \ldots! m_N! n_1! n_2! \ldots, n_N!},$$

т.е. выборочное распределение вероятности одиночного фотона пропорционально постоянной U_{st}, подматрицы унитарного интерферометра, зависящей от входного и выходного состояний Фока.

В то время как определитель может быть эффективно вычислен на классических компьютерах, вычисление перманента относится к классу задач вычислительной сложности *#P-Hard*, которые, как считается, классически трудно вычислить.

Это означает, что моделирование выборки бозонов не может быть эффективно выполнено на классическом компьютере, что представляет собой

потенциальный вызов расширенному тезису Черча-Тьюринга и демонстрирует возможности (неуниверсальных) квантовых вычислений.

3 Порядок выполнения работы

3.1 Построение схемы и моделирование

В квантовой линейной оптике многомодовый линейный интерферометр обычно подразделяется на двухмодовые делители луча (*BSgate*) и одномодовые фазовращатели (*Rgate*), что позволяет легко перевести его в квантовую схему *CV*.

Например, в случае четырехмодового интерферометра с произвольным 4×4 унитарный *U*, схема квантовой фотоники задается как представлено на рисунке 1.

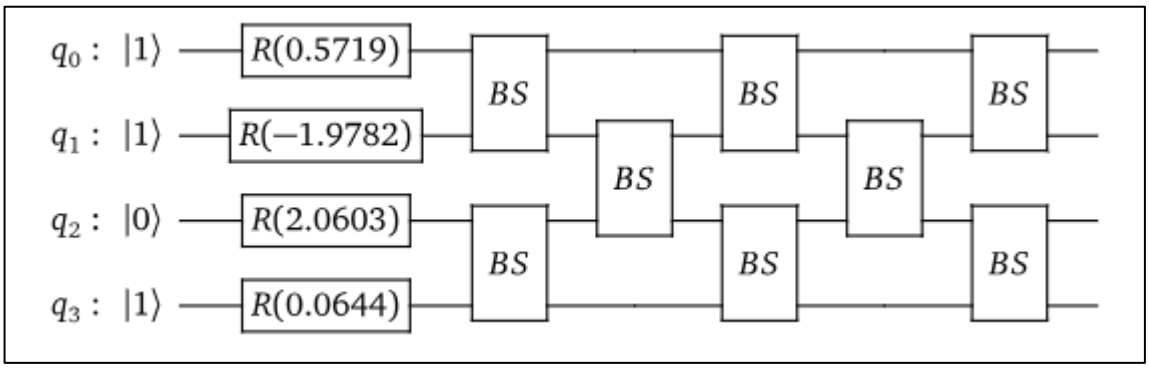

Рисунок 1 – Схема квантовой фотоники

В вышеописанном случае детекторы выполняют измерения состояния Фока, а параметры светоделителей и вращающихся затворов определяют унитарный *U*. Для того, чтобы учесть произвольные линейные единицы для *m*-импульсных мод, мы должны иметь минимум *m*+1 столбцов в массиве светоделителей.

Моделировать эту схему с помощью *Strawberry Fields* несложно: мы можем просто считать элементы слева направо и преобразовать ее в язык схем *Blackbird*.

Для начала мы создадим квантовую программу выборки бозонов с помощью *Strawberry Fields*:

```
import numpy as np
# set the random seed
np.random.seed(42)
# import Strawberry Fields
import strawberryfields as sf
from strawberryfields.ops import *
# initialize a 4 mode program
boson_sampling = sf.Program(4)
with boson_sampling.context as q:
    # 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)  | 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])
```

В этом конкретном примере следует отметить несколько моментов:

1. Мы инициализируем входное состояние $|\psi\rangle = |1,1,0,1\rangle$ путем создания однофотонного состояния Фока в модах 0, 1 и 3. Это делается с помощью оператора *Fock*. Режим 2 инициализируется как вакуумное состояние с помощью оператора *Vac* (сокращение от *Vacuum*). Это необязательно – режимы автоматически создаются в вакуумном состоянии при инициализации двигателя.

2. Далее мы применим элементы управления вращением, *Rgate*, к каждой моде. В результате поворот в фазовом пространстве происходит против часовой стрелки на угол ϕ.

3. Наконец, мы применяем массив светоделителей, используя оператор *BSgate* с аргументами (*theta*, *phi*).

    a. Амплитуда передачи в этом случае определяется следующим образом:

$$t = \cos\theta,$$

b. Амплитуда отражения определяется:

$$r = e^{i\phi}\sin\theta.$$

4. Параметры вращающегося вентиля и светоделителя выбраны произвольно, что приводит к произвольному унитарному U, действующему на операторы аннигиляции и креации входных мод. После выхода из матрицы светоделителей обозначим выходное состояние через $|\psi\rangle$.

5. Мы не проводим измерений Фока на выходе; это необходимо для того, чтобы обеспечить сохранение состояния, чтобы мы могли извлечь совместные вероятности фоковского состояния из матрицы светоделителей.

Если мы хотим смоделировать измерения Фока, мы можем дополнительно включить

```
MeasureFock() | q
```

после матрицы светоделителей. После построения схемы и запуска двиателя значения измерений состояния Фока будут доступны в атрибуте *samples* объекта *Result*, возвращаемого двигателем.

Теперь, когда программа создана, мы можем инициализировать движок и выполнить программу на одном из встроенных симуляторов состояний *Strawberry Fields*.

Поскольку только фоковские бэкенды поддерживают негауссовы начальные состояния, используемые в бозонной дискретизации, мы будем использовать бэкенд *NumPy 'fock'*, а также усечение/отсечение фоковских состояний на 7 (т.е. вся информация о квантовых амплитудах фоковских состояний $|n\rangle, n \geq 7$, отбрасывается).

```
eng=sf.Engine(backend="fock", backend_options={"cutoff_dim": 7})
```

Теперь мы можем выполнить программу с помощью движка:

```
results = eng.run(boson_sampling)
```

Метод состояния *all_fock_probs()* возвращает $\overbrace{D\times D\times\ldots\times D}^{\text{количество режимов}}$ массив, где D это усечение базиса Фока, содержащее предельные вероятности состояний Фока – в данном примере он имеет размерность $7\times 7\times 7$. Затем мы можем получить доступ к вероятности измерения определенного состояния выхода с помощью индексации:

$$\text{probs}[1,1,1,0] = |\langle 1,1,1,0|\psi\rangle|^2,$$

```
# extract the joint Fock probabilities
probs = results.state.all_fock_probs()
# print the joint Fock state probabilities
print(probs[1, 1, 0, 1])
print(probs[2, 0, 0, 1])
```

Выход:

```
0.17468916048563932
0.1064419272464234
```

3.2 Вычисление унитарных

Давайте исследуем эти результаты дальше. Для этого, однако, нам нужно вычислить унитарное преобразование, которое применяется нашими светоделителями и вращающимися затворами.

3.2.1 Вручную

Сначала импортируйте несколько полезных библиотек, например *NumPy*, а также функции *multi_dot* и *block_diag* из *NumPy* и *SciPy* соответственно:

```
import numpy as np
from numpy.linalg import multi_dot
from scipy.linalg import block_diag
```

Для начала сгенерируем матрицу, представляющую собой унитарное преобразование операторов аннигиляции и креации входных мод. Элементы вращения действуют следующим образом:

$$R(\phi)\hat{a} = \hat{a}e^{i\phi},$$

и, таким образом, столбец вращательных ворот имеет следующий блочно-диагональный вид:

$$U_\phi = \begin{bmatrix} e^{i\phi_1} & 0 & \cdots \\ 0 & e^{i\phi_2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix},$$

Генерируем это в *NumPy*:

```
Uphase                =                np.diag([np.exp(0.5719*1j),np.exp(-
1.9782*1j),np.exp(2.0603*1j),np.exp(0.0644*1j)])
```

Одиночный разделитель лучей, действующий на два входных режима $(\hat{a}_1, \hat{a}_2)$, вместо этого действует следующим образом:

$$BS(\theta,\phi)(\hat{a}_1,\hat{a}_2) = \begin{bmatrix} t & -r^* \\ r & t \end{bmatrix}\begin{bmatrix} \hat{a}_1 \\ \hat{a}_2 \end{bmatrix},$$

где $t = \cos\theta$ и $r = e^{i\phi}\sin\theta$.

Как и в случае с элементом вращения, они объединяются в блочно-диагональных матриц.

Прежде всего, нам нужно преобразовать аргументы *BSgate*, *theta* и *phi* (для удобства приведены ниже):

```
BSargs = [
    (0.7804, 0.8578),
    (0.06406, 0.5165),
    (0.473, 0.1176),
    (0.563, 0.1517),
    (0.1323, 0.9946),
    (0.311, 0.3231),
    (0.4348, 0.0798),
    (0.4368, 0.6157)
]
```

в амплитуды пропускания и отражения, *t* и *r* соответсвенно:

```
t_r_amplitudes = [(np.cos(q), np.exp(p*1j)*np.sin(q)) for q,p in BSargs]
```

Далее мы генерируем унитарные преобразования для отдельных лучевых рассеивателей:

```
BSunitaries = [np.array([[t, -np.conj(r)], [r, t]]) for t,r in
t_r_amplitudes]
```

Перед использованием функции *SciPy – scipy.linalg.block_diag*, для вычисления результирующего значения U_{BS_i}, т.е. единицу, соответствующую каждому столбцу «независимых» светоделителей в приведенной выше принципиальной схеме:

```
UBS1 = block_diag(*BSunitaries[0:2])
UBS2 = block_diag([[1]], BSunitaries[2], [[1]])
UBS3 = block_diag(*BSunitaries[3:5])
UBS4 = block_diag([[1]], BSunitaries[5], [[1]])
UBS5 = block_diag(*BSunitaries[6:8])
```

Наконец, мы объединяем унитарные элеименты в единое целое 4×4 с помощью матричного умножения; $U = U_{BS_5} U_{BS_4} U_{BS_3} U_{BS_2} U_{BS_1} U_{\phi}$.

Поскольку *numpy.dot* поддерживает только матричное умножение двух массивов, вместо него мы используем *numpy.linalg.multi_dot*:

```
U = multi_dot([UBS5, UBS4, UBS3, UBS2, UBS1, Uphase])
print(np.round(U, 4))
```

Выход:

```
[[0.2195-0.2565j 0.6111+0.5242j -0.1027+0.4745j -0.0273+0.0373j]
[0.4513+0.6026j 0.457 +0.0123j 0.1316-0.4504j 0.0353-0.0532j]
[0.0387+0.4927j -0.0192-0.3218j -0.2408+0.5244j -0.4584+0.3296j]
[-0.1566+0.2246j 0.11 -0.1638j -0.4212+0.1836j 0.8188+0.068j ]]
```

Мы находим, что:

$$U = \begin{bmatrix} 0.2195 - 0.2565i & 0.6111 + 0.5242i & -0.1027 + 0.4745i & -0.0273 + 0.0373i \\ 0.4513 + 0.6026i & 0.4570 + 0.0123i & 0.1316 - 0.4504i & 0.0353 - 0.0532i \\ 0.0387 + 0.4927i & -0.0192 - 0.3218i & -0.2408 + 0.5244i & -0.4584 + 0.3296i \\ -0.1566 + 0.2246i & 0.1100 - 0.1638i & -0.4212 + 0.1836i & 0.8188 + 0.068i \end{bmatrix}$$

3.2.2 Гауссовский компилятор

Хотя мы сделали это вручную, *Strawberry Fields* также поддерживает гауссовский унитарный компилятор, который позволяет нам скомпилировать нашу программу в единый гауссовский унитарный компилятор.

Обратите внимание, что мы должны создать новую программу без начальных состояний Фока, так как компилятор гауссовых унитарных операций работает только с гауссовыми операциями.

Компиляция программы:

```
prog_unitary = sf.Program(4)
prog_unitary.circuit = boson_sampling.circuit[4:]
prog_compiled=prog_unitary.compile(compiler="gaussian_unitary")
```

Напечатав *prog_compiled*, мы видим, что теперь он состоит из одной операции *GaussianTransform*, состоящей из одной симплектической матрицы:

```
prog_compiled.print()
```

Выход:

```
GaussianTransform([[ 0.2195 0.6111 -0.1027 -0.0273 0.2565 -0.5242 -0.4745
-0.0373]
 [ 0.4513 0.457 0.1316 0.0353 -0.6026 -0.0123 0.4504 0.0532]
 [ 0.0387 -0.0192 -0.2408 -0.4584 -0.4927 0.3218 -0.5244 -0.3296]
 [-0.1566 0.11 -0.4212 0.8188 -0.2246 0.1638 -0.1836 -0.068 ]
 [-0.2565 0.5242 0.4745 0.0373 0.2195 0.6111 -0.1027 -0.0273]
 [ 0.6026 0.0123 -0.4504 -0.0532 0.4513 0.457 0.1316 0.0353]
 [ 0.4927 -0.3218 0.5244 0.3296 0.0387 -0.0192 -0.2408 -0.4584]
 [ 0.2246 -0.1638 0.1836 0.068 -0.1566 0.11 -0.4212 0.8188]]) | (q[0],
q[1], q[2], q[3])
```

Мы можем легко извлечь эту симплектическую матрицу и переписать ее в виде унитарной матрицы:

```
S = prog_compiled.circuit[0].op.p[0]
U = S[:4, :4] + 1j*S[4:, :4]
print(U)
```

Выход:

```
[[0.2195-0.2565j 0.6111+0.5242j -0.1027+0.4745j -0.0273+0.0373j]
[0.4513+0.6026j 0.457 +0.0123j 0.1316-0.4504j 0.0353-0.0532j]
[0.0387+0.4927j -0.0192-0.3218j -0.2408+0.5244j -0.4584+0.3296j]
[-0.1566+0.2246j 0.11 -0.1638j -0.4212+0.1836j 0.8188+0.068j ]]
```

Что совпадает с результатом, приведенным выше.

3.2.3 Работа интерферометра

Strawberry Fields поддерживает операцию *Interferometer*, которая позволяет напрямую встраивать числовые унитарные матрицы в программы и декомпозировать их в светоделители и вентили поворота:

```
boson_sampling = sf.Program(4)
with boson_sampling.context as q:
    # prepare the input fock states
    Fock(1) | q[0]
    Fock(1) | q[1]
    Vac     | q[2]
    Fock(1) | q[3]
    Interferometer(U) | q
```

Скомпилируйте это для бэкенда *Fock* и выведите результат:

```
boson_sampling.compile(compiler="fock").print()
```

Выход:

```
Fock(1) | (q[0])
Fock(1) | (q[1])
Vac | (q[2])
Fock(1) | (q[3])
Rgate(-3.124) | (q[0])
BSgate(0.9465, 0) | (q[0], q[1])
Rgate(2.724) | (q[2])
BSgate(0.09485, 0) | (q[2], q[3])
Rgate(-0.9705) | (q[1])
BSgate(0.7263, 0) | (q[1], q[2])
Rgate(-1.788) | (q[0])
BSgate(0.8246, 0) | (q[0], q[1])
Rgate(5.343) | (q[0])
Rgate(2.93) | (q[1])
Rgate(3.133) | (q[2])
Rgate(0.07904) | (q[3])
BSgate(-0.533, 0) | (q[2], q[3])
Rgate(2.45) | (q[2])
BSgate(-0.03962, 0) | (q[1], q[2])
Rgate(2.508) | (q[1])
```

3.3 Сравнение с перманетом

Теперь, когда у нас есть унитарное преобразование интерферометра *U*, а также «экспериментальные» результаты, давайте сравним их и посмотрим, будет ли результат выборки бозонов:

$$\left|\left\langle n_1, n_2, \cdots, n_N \middle| \psi' \right\rangle\right|^2 = \frac{\left|\mathrm{Perm}\left(U_{st}\right)\right|^2}{m_1! m_2! \ldots! m_N! n_1! n_2! \ldots, n_N!},$$

Для этого примера мы рассмотрим состояние выхода $\left|2,0,0,1\right\rangle$. Извлекая $\left|\left\langle 2,0,0,1\middle|\psi'\right\rangle\right|^2$ из выходных данных, мы видим, что:

```
print(probs[2,0,0,1])
```

Выход:

```
0.1064419272464234
```

Прежде чем мы сможем вычислить правую часть уравнения, нам понадобится метод вычисления перманента. Поскольку перманент классически трудно вычислить, он не предусмотрен ни в *NumPy*, ни в *SciPy*, поэтому мы будем использовать библиотеку *Walrus*, установленную вместе с *Strawberry Fields*:

```
from thewalrus import perm
```

Определим подматрицу U_{st}. Сначала мы вычисляем подматрицу U_s, снимая m_k копии *k*-х столбцов *U*, где m_k – номера фотонов *k*-х входных состояний.

Поскольку наше входное состояние $\left|\psi\right\rangle = \left|1,1,0,1\right\rangle$, мы просто берем отдельные копии первого, второго и четвертого столбцов:

```
U[:,[0,1,3]]
```

Выход:

```
array([[ 0.2195-0.2565j, 0.6111+0.5242j, -0.0273+0.0373j],
 [ 0.4513+0.6026j, 0.457 +0.0123j, 0.0353-0.0532j],
 [ 0.0387+0.4927j, -0.0192-0.3218j, -0.4584+0.3296j],
 [-0.1566+0.2246j, 0.11 -0.1638j, 0.8188+0.068j ]])
```

Далее мы делаем n_k копий *k*-й строки U_s, где n_k – номер фотона *k*-го выходного состояния, которое измеряется. Здесь наше измерение $|2,0,0,1\rangle\langle 2,0,0,1|$, поэтому мы берем две копии первой строки и одну копию последней строки:

```
U[:,[0,1,3]][[0,0,3]]
```

Выход:

```
array([[ 0.2195-0.2565j, 0.6111+0.5242j, -0.0273+0.0373j],
 [ 0.2195-0.2565j, 0.6111+0.5242j, -0.0273+0.0373j],
 [-0.1566+0.2246j, 0.11 -0.1638j, 0.8188+0.068j ]])
```

Теперь, используя функцию *permanent*, которую мы импортировали выше, мы можем взять абсолютное значение квадрата постоянной. Наконец, мы делим на произведение факториалов входных и выходных чисел состояний. Поскольку $0! = 1! = 1$ нам нужно учесть только случай $2! = 2$:

```
print(np.abs(perm(U[:, [0,1,3]][[0,0,3]]))**2 / 2)
```

Выход:

```
0.10644192724642332
```

Сравнивая этот результат с результатом *Strawberry Fields*, мы видим, что они отличаются только на 17-м знаке после запятой. Вычисляем точную разницу в процентах,

```
BS = np.abs(perm(U[:, [0,1,3]][[0,0,3]]))**2 / 2
SF = probs[2,0,0,1]
print(100*np.abs(BS-SF)/BS)
```

Выход:

```
7.822737618618671e-14
```

Они совпадают с почти ничтожной погрешностью. Это объясняется высокой точностью фоковского бэкэнда, несмотря на усечение/отсечение фоковского состояния.

Такой же близкий результат можно получить и для любого другого измерения выходного фоковского состояния, сохраняющего, например, число фотонов:

1) $\left|3,0,0,0\right\rangle\left\langle 3,0,0,0\right|$:

```
print(probs[3,0,0,0])
print(np.abs(perm(U[:, [0,1,3]][[0,0,0]]))**2 / 6)
```

Выход:

```
0.0009458483347132492
0.0009458483347132489
```

2) $|1,1,0,1\rangle\langle 1,1,0,1|$:

```
print(probs[1,1,0,1])
print(np.abs(perm(U[:, [0,1,3]][[0,1,3]]))**2 / 1)
```

Выход:

```
0.17468916048563932
0.17468916048563934
```

Примечание:

Хотя бэкенды Фока возвращают только приближенное значение совместной вероятности состояний Фока и, таким образом, только аппроксимируют различные подматричные перманенты, они все равно вычисляют классически неразрешимую проблему.

Это происходит потому, что аппроксимация матричных перманентов остается счетно трудной классической задачей.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. M.A. Nielsen and I.L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2010. ISBN 9780511992773. URL: https://books.google.ca/books?id=JRz3jgEACAAJ.

2. Scott Aaronson and Alex Arkhipov. The computational complexity of linear optics. Theory of Computing, 9(1):143–252, 2013. doi:10.4086/toc.2013.v009a004.

3. Max Tillmann, Borivoje Dakić, René Heilmann, Stefan Nolte, Alexander Szameit, and Philip Walther. Experimental boson sampling. Nature Photonics, 7(7):540–544, May 2013. doi:10.1038/nphoton.2013.102.

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

5. Michael Reck, Anton Zeilinger, Herbert J. Bernstein, and Philip Bertani. Experimental realization of any discrete unitary operator. Physical Review Letters, 73(1):58–61, Jul 1994. doi:10.1103/physrevlett.73.58.

6. William R Clements, Peter C Humphreys, Benjamin J Metcalf, W Steven Kolthammer, and Ian A Walsmley. Optimal design for universal multiport interferometers. Optica, 3(12):1460–1465, 2016. doi:10.1364/OPTICA.3.001460.

7. Max Tillmann, Borivoje Dakić, René Heilmann, Stefan Nolte, Alexander Szameit, and Philip Walther. Experimental boson sampling. Nature Photonics, 7(7):540–544, May 2013. doi:10.1038/nphoton.2013.102.

Практическая работа № 3. «Выборка гауссовых бозонов и гафниан»

1 Цель работы

Рассмотрение применения выборки по гауссову бозону.

2 Основная теория

2.1 Фоновая теория

Хотя выборка бозонов позволяет экспериментально реализовать задачу выборки, которая в классическом понимании является счетно сложной, одной из основных проблем, возникающих при экспериментальных установках, является одна из масштабируемости из-за его зависимости от множества одновременно излучающих источников одиночных фотонов. В настоящее время в большинстве физических реализаций выборки бозонов используется процесс, известный как спонтанное параметрическое преобразование с понижением для генерации входных данных однофотонного источника. Однако этот метод недетерминирован – по мере увеличения количества режимов в устройстве среднее время, необходимое для того, чтобы каждый источник фотонов испустил одновременный фотон, увеличивается экспоненциально.

Для имитации детерминированного массива однофотонных источников было предложено несколько вариантов бозонной выборки; наиболее известным из них является бозонная выборка с разбросом. Однако недавняя вариация бозонной выборки, предложенная Гамильтоном и др., полностью устраняет необходимость в однофотонных фоковских состояниях, показывая, что падающие гауссовы состояния – в данном случае одномодовые сжатые состояния – могут создавать проблемы того же класса вычислительной сложности, что и бозонная выборка. Что ещё более важно, это устраняет проблему масштабируемости, связанную с источниками одиночных фотонов,

поскольку одномодовые сжатые состояния могут генерироваться детерминированно одновременно.

Схема выборки гауссовских бозонов при первоначальном наблюдении остаётся довольно похожей на схему выборки бозонов:

– N одномодовые сжатые состояния $|z\rangle$ с параметром сжатия $z = re^{i\phi}$, введите N модовый линейный интерферометр, описываемый унитарным U одновременно.

– Каждый выходной режим интферометра (обозначаемый состоянием $|\psi'\rangle$), затем измеряется в базисе Фока, $\otimes_i n_i|n_i\rangle\langle n_i|$.

Без потери общности мы можем исключить параметр фазы сжатия ϕ вставьте в интерферометр и установите ϕ=0 для удобства.

Используя методы фазового пространства, Гамильтон и др. [2] показали , что вероятность измерения состояния Фока , содержащего только 0 или 1 фотон на моду , определяется как:

$$\left|\langle n_1, n_2, \ldots, n_N|\psi'\rangle\right|^2 = \frac{\left|\mathrm{Haf}[(U(\bigoplus_i \tanh(r_i))U^T)]_{st}\right|^2}{\prod_{i=1}^{N}\cosh(r_i)}$$

т.е. выбранное распределение вероятностей одиночных фотонов пропорционально гафнову подматрицы $U(\bigoplus i\tanh(ri))UT$, зависит от выходной ковариационной матрицы.

Примечание:

Гафниан матрицы определяется как:

$$\mathrm{Haf}(A) = \frac{1}{n!2^n}\sum_{\sigma\in S_{2N}}\prod_{i=1}^{N}A_{\sigma(2i-1)\sigma(2i)}$$

где S_{2N} является набором всех перестановок $2N$ элементы.

В теории графов гафнов вычисляет количество совершенных совпадений в произвольном графе с матрицей смежности A.

Сравните это с постоянным, которое вычисляет количество идеальных совпадений на *двудольном* графике – гафновский график оказывается обобщением постоянного с соотношением:

$$\mathrm{Per}(A) = \mathrm{Haf}\left(\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}\right)$$

Поскольку любой алгоритм, который мог бы вычислить (или даже приблизить) гафниан, мог бы также вычислить постоянную – *a # P*-сложную задачу - из этого следует, что вычисление или аппроксимация гафниана также должна быть классически сложной задачей.

3 Порядок выполнения работы

3.1 Построение схемы и моделирование

Как и в случае с проблемой выборки бозонов, многомодовый линейный интерферометр может быть разделен на двухмодовые светоделители (*BSgate*) и одномодовые фазовращатели (*Rgate*), позволяющие осуществлять прямой перевод в квантовую схему CV.

Например, в случае 4-модового интерферометра с произвольным 4×4 унитарный U квантовая схема *CV* для выборки гауссовского бозона дана следующим образом (рис.1).

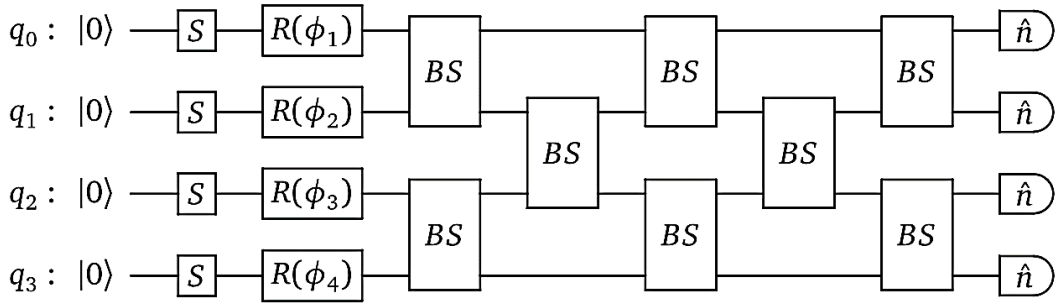

Рисунок 1 – Квантовая схема *CV* для выборки гауссовского бозона

В приведенном выше примере для всех состояний одномодового сжатия применяется идентичное сжатие $z=r$ параметры светоделителей и

вентилей вращения определяют унитарную U и, наконец, детекторы выполняют измерения состояния Фока в выходных режимах. Как и в случае с выборкой по бозону, для N режимы ввода, мы должны иметь минимум N+1 столбцы в массиве светоделителей.

Смоделировать эту схему с помощью *Strawberry Fields* несложно; мы можем просто считывать вентили слева направо и преобразовывать это в язык схем *Blackbird*.

Для начала мы создадим квантовую программу выборки бозонов с использованием *Strawberry Fields*:

```
import numpy as np
# set the random seed
np.random.seed(42)
# import Strawberry Fields
import strawberryfields as sf
from strawberryfields.ops import *
```

В отличие от выборки бозонов и постоянной схемы, мы непосредственно применим гауссову унитарную схему к схеме, используя операцию *Interferometer* . Сначала мы должны определить унитарную матрицу, которую мы хотели бы встроить в схему:

```
# define the linear interferometer
U = np.array([
 [ 0.219546940711-0.256534554457j, 0.611076853957+0.524178937791j,
    -0.102700187435+0.474478834685j,-0.027250232925+0.03729094623j],
 [ 0.451281863394+0.602582912475j, 0.456952590016+0.01230749109j,
    0.131625867435-0.450417744715j, 0.035283194078-0.053244267184j],
 [ 0.038710094355+0.492715562066j,-0.019212744068-0.321842852355j,
    -0.240776471286+0.524432833034j,-0.458388143039+0.329633367819j],
 [-0.156619083736+0.224568570065j, 0.109992223305-0.163750223027j,
    -0.421179844245+0.183644837982j, 0.818769184612+0.068015658737j]
])
```

Теперь мы можем использовать это для построения схемы:

```
# create the 4 mode Strawberry Fields program
```

```
gbs = sf.Program(4)
with gbs.context as q:
    # prepare the input squeezed states
    S = Sgate(1)
    S | q[0]
    S | q[1]
    S | q[2]
    S | q[3]
    # linear interferometer
    Interferometer(U) | q
```

В этом конкретном примере следует отметить несколько моментов:

1. Для подготовки входного одномодового состояния сжатого вакуума $|z\rangle$, где z=1, к каждой из мод (изначально находящихся в состоянии вакуума) мы применим сжимающие ворота *Sgate*.

2. Затем мы применим линейный интерферометр ко всем четырем модам, используя оператор разложения *Interferometer* и унитарную матрицу *U*. Этот оператор разлагает унитарную матрицу, представляющую линейный интерферометр, на одномодовые вращающиеся затворы *Rgate* и двухмодовые бамсплиттеры *BSgate*. После применения интерферометра обозначим выходное состояние через $|\psi'\rangle$.

Примечание:

Вы можете просмотреть разложенные светоделители и вентили вращения, которые соответствуют линейному интерферометру *U* позвонив по телефону *eng.print_applied()* после запуска двигателя.

Примечание:

Интерферометр применяется здесь идентичные от Бозон отбора проб и постоянного. В результате разлагаются ответвитель и параметров вращения ворота также будут идентичны.

3. Мы не выполняем измерения Фока на выходе; это делается для обеспечения сохранения состояния, чтобы мы могли извлечь вероятности совместного состояния Фока из матрицы светоделителей.

В отличие от бозон отбора проб учебник отсутствие фоковских состояний означает, что теперь мы можем использовать библиотеки *numpy*

основе *'gaussian'* серверная часть, вместе с 4-регистровый режим. Серверная часть *Gaussian* идеально подходит для моделирования выборки гауссовского бозона, поскольку все начальные состояния являются гауссовыми, и все необходимые операторы преобразуют гауссовы состояния в другие гауссовы состояния.

Теперь, когда программа создана, мы можем инициализировать движок и выполнить программу на одном из встроенных имитаторов состояния *Strawberry Fields*.

```
eng = sf.Engine(backend="gaussian")
results = eng.run(gbs)
```

Метод состояния *fock_prob()* принимает список или кортеж, содержащий состояние Фока, подлежащее измерению, и возвращает вероятность этого измерения. Например, [1,2,0,1] представляет измерение, приводящее к обнаружению 1 фотона в режиме q[0] и режиме q[3], и 2 фотона в режиме q[1], и вернет значение

$$prob(1,2,0,1) = || \langle 1,2,0,1 \mid \psi' \rangle ||^2$$

Метод состояний Фока *all_fock_probs()*, использовавшийся ранее для возврата ВСЕ Вероятности состояний Фока в виде массива не поддерживается серверной частью *Gaussian*. Это связано с тем, что вычисление вероятностей состояний Фока в гауссовском представлении имеет экспоненциальное масштабирование - хотя это хорошо для вычисления конкретных вероятностей на основе Фока, возврат становится требовательным с вычислительной точки зрения ВСЕ Вероятности состояний Фока с использованием гауссовского бэкенда.

Давайте воспользуемся *fock_prob()* в цикле *for* для извлечения вероятностей измерения различных состояний Фока:

```
# 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 = results.state.fock_prob(i)
    print("|{}>:  {}".format("".join(str(j)  for  j  in  i),
prob))
```

Выход:

```
|0000>: 0.17637844761413482
|1100>: 0.0685595637122452
|0101>: 0.00205609725897229
|1111>: 0.008342946399881956
|2000>: 0.010312945253440318
```

3.2 Входные данные в равной степени ограничены

Как было показано ранее, формула для вычисления вероятности выходного состояния Фока при выборке гауссовского бозона дана следующим образом:

$$\left|\langle n_1, n_2, \ldots, n_N|\psi'\rangle\right|^2 = \frac{\left|\mathrm{Haf}[(U(\bigoplus_i \tanh(r_i))U^T)]_{st}\right|^2}{\prod_{i=1}^{N}\cosh(r_i)}$$

где U это унитарное преобразование вращения / светоделителя для операторов аннигиляции и создания режимов ввода и вывода.

Однако в этом конкретном примере мы используем тот же самый параметр сжатия, $z=r$ для всех входных состояний - это позволяет нам упростить это уравнение. Начнем с того, что выражение гафниана просто становится $Haf[(UUTtanh\ (r))]_{st}$, устраняющие необходимость в тензорной сумме.

Таким образом, мы имеем:

$$\left|\left\langle n_1, n_2, \ldots, n_N \middle| \psi' \right\rangle\right|^2 = \frac{\left|\mathrm{Haf}[(UU^T \tanh(r))]_{st}\right|^2}{n_1! n_2! \cdots n_N! \cosh^N(r)}.$$

Теперь, когда у нас есть унитарное преобразование интерферометра U наряду с «экспериментальными» результатами, давайте сравним их и посмотрим, согласуется ли результат выборки гауссовского бозона в случае одинаково сжатых входных режимов с вероятностями моделирования *Strawberry Fields*.

3.3 Вычисление гафниана

Прежде чем мы сможем вычислить правую часть уравнения выборки гауссовского бозона, нам нужен метод вычисления гафниана. Поскольку гафниан классически сложен для вычисления, он не предусмотрен ни в *NumPy* или *SciPy*, поэтому мы будем использовать *Walrus* библиотека, установленная рядом с *Strawberry Fields*:

```
from thewalrus import hafnian as haf
```

Теперь для числителя в правой части мы сначала вычислим подматрицу $[(UUTtanh(r))]_{st}$:

```
B = (np.dot(U, U.T) * np.tanh(1))
```

В отличие от случая выборки бозонов, при выборке гауссовых бозонов мы определяем подматрицу, беря строки и столбцы, соответствующие измеренному состоянию Фока.

Например, для вычисления подматрицы в случае выходного измерения |1,1,0,0⟩|:

```
print(B[:, [0, 1]][[0, 1]])
```

Выход:

```
[[-0.10219728+0.32633851j  0.55418347+0.28563583j]
 [ 0.55418347+0.28563583j -0.10505237+0.32960794j]]
```

3.4 Сравнение с *Strawberry Fields*

Теперь, когда у нас есть метод вычисления гафниана, давайте сравним результат с результатом, предоставленным *Strawberry Fields*.

Измерение |0,0,0,0⟩ на выходе

Это соответствует гафниану пустой матрицы, которая равна просто 1:

```
print(1 / np.cosh(1) ** 4)
print(results.state.fock_prob([0, 0, 0, 0]))
```

Выход:

```
0.1763784476141347
0.17637844761413482
```

Измерение |1,1,0,0⟩ на выходе

```
B = (np.dot(U, U.T) * np.tanh(1))[:, [0, 1]][[0, 1]]
print(np.abs(haf(B)) ** 2 / np.cosh(1) ** 4)
print(results.state.fock_prob([1, 1, 0, 0]))
```

Выход:

```
0.0685595637121454
0.0685595637122452
```

Измерение |0,1,0,1⟩ **на выходе**

```
B = (np.dot(U, U.T) * np.tanh(1))[:, [1, 3]][[1, 3]]
print(np.abs(haf(B)) ** 2 / np.cosh(1) ** 4)
print(results.state.fock_prob([0, 1, 0, 1]))
```

Выход:

```
0.0020560972589728135
0.00205609725897229
```

Измерение |1,1,1,1⟩ **на выходе**

Это соответствует гафниану полной матрицы *B=UUTtanh (r)*:

```
B = (np.dot(U, U.T) * np.tanh(1))
print(np.abs(haf(B)) ** 2 / np.cosh(1) ** 4)
print(results.state.fock_prob([1, 1, 1, 1]))
```

Выход:

```
0.008342946399869365
0.008342946399881956
```

Измерение |2,0,0,0⟩**на выходе**

Поскольку у нас в режиме два фотона *q*[0], мы берем две копии первой строки и первого столбца, обязательно разделяя на 2!:

```
B = (np.dot(U, U.T) * np.tanh(1))[:, [0, 0]][[0, 0]]
print(np.abs(haf(B)) ** 2 / (2 * np.cosh(1) ** 4))
print(results.state.fock_prob([2, 0, 0, 0]))
```

Выход:

```
0.010312945253479155
0.010312945253440318
```

Результаты моделирования *Strawberry Fields* согласуются (с почти незначительной численной ошибкой) с ожидаемым результатом из уравнения выборки гауссовского бозона!

Предупреждение:

Повторите этот урок с:

1. Серверная часть *Fock*, такая как *'fock'* вместо серверной части *Gaussian*.

2. Различные параметры светоделителя и вращения.

3. Входные состояния с *разными* сжатыми значениями r_i. Вам нужно будет изменить код, чтобы учесть тот факт, что теперь необходимо вычислить определитель выходной ковариационной матрицы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. A. P. Lund, A. Laing, S. Rahimi-Keshari, T. Rudolph, J. L. O'Brien, and T. C. Ralph. Boson sampling from a gaussian state. Physical Review Letters, 113:100502, Sep 2014. doi:10.1103/PhysRevLett.113.100502.

2. Craig S. Hamilton, Regina Kruse, Linda Sansoni, Sonja Barkhofen, Christine Silberhorn, and Igor Jex. Gaussian boson sampling. Physical Review Letters, 119:170501, Oct 2017. arXiv:1612.01199, doi:10.1103/PhysRevLett.119.170501.

3. Michael Reck, Anton Zeilinger, Herbert J. Bernstein, and Philip Bertani. Experimental realization of any discrete unitary operator. Physical Review Letters, 73(1):58–61, Jul 1994. doi:10.1103/physrevlett.73.58.

4. William R Clements, Peter C Humphreys, Benjamin J Metcalf, W Steven Kolthammer, and Ian A Walsmley. Optimal design for universal multiport interferometers. Optica, 3(12):1460–1465, 2016. doi:10.1364/OPTICA.3.001460.

Практическая работа №4. «Выборка бозонов методом рассеивания»

1 Цель работы

Реализация бозонного семплирования по методу *Scattershot* в *Strawberry Fields*.

2 Основная теория

Как мы уже видели в учебнике по бозонной выборке, бозонный семплер – это квантовая машина, которая принимает детерминированный входной сигнал, состоящий из m режимов, n из которых одновременно посылают фотоны через интерферометр, моделируемый унитарной матрицей U. Выход интерферометра – это случайное распределение фотонов, которое можно вычислить классически с помощью постоянной U.

Выборка рассеянных бозонов (ВРБ) была мотивирована тем, что излучение n фотонов одновременно на вход экспериментально очень трудно реализовать для больших n. Проще построить случайный вход с использованием спонтанного параметрического преобразования с уменьшением, распределение которого задается следующим образом $P(k_i = k) = (1-\chi^2)\chi^2$, где k_i число фотонов в i-режиме, $\chi \in (-1,1)$ – заданный параметр. Преимущество спонтанного параметрического преобразования с уменьшением не только в том, что это когерентный источник фотонов, но и в том, что он всегда излучает четное количество фотонов: один, который можно использовать в схеме выборки бозонов, и один для измерения входного сигнала.

В квантовой оптике мы моделируем спонтанного параметрического преобразования с уменьшением с помощью двухмодовых фильтров сжатия $\hat{S}_2$ такие, что $\hat{S}_2|0\rangle|0\rangle = \sqrt{(1-\chi^2)}\sum_{k=0}^{\infty}\chi^k|k\rangle|k\rangle$. Первый импульс будет использоваться для измерения входного сигнала, а второй будет направлен в схему.

В *Strawberry Fields* (*SF*), этот двухрежимный отжимной затвор называется `S2gate` и принимает на вход параметр отжатия r связанный с χ по формуле $r = \tanh(\chi)$.

```
import numpy as np
import scipy as sp
from math
importfactorial,tanhimportitertoolsimportmatplotlib.pyplotasp
lt# %matplotlib inlineimport
   matplotlib.pathasmpathimportmatplotlib.linesasmlines
import matplotlib.patches as mpatches
from matplotlib.collections import PatchCollection

import strawberryfields as sf
from strawberryfields.ops import *

colormap                                                    =
np.array(plt.rcParams['axes.prop_cycle'].by_key()['color'])
```

3 Порядок выполнения работы

3.1 Создание схемы

3.1.1 Константы

Наша схема будет зависеть от нескольких параметров. Первые константы – это параметр сжатия $r \in [-1,1]$ и число отсечки, которое соответствует максимальному количеству фотонов на моду, учитываемому в вычислениях (используется для того, чтобы сделать моделирование трассируемым).

```
r_squeezing = 0.5 # squeezing parameter for the S2gate (here
taken randomly between -1 and 1)
cutoff = 7 # max number of photons computed per mode
```

Затем идет унитарная матрица, представляющая интерферометр. Мы решили реализовать 4-модовый бозонный сэмплер, и поэтому нам нужны 4×4-унитарная матрица. Для этого подойдет любая унитарная матрица, но

для простоты мы решили реализовать ее с помощью двух вращений: одно с углом θ_1 для кумодов 1 и 2, и еще один с углом θ_2 для кубитов 3 и 4. Итоговая матрица имеет вид:

$$\begin{pmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & \cos(\theta_2) & -\sin(\theta_2) \\ 0 & 0 & \cos(\theta_2) & \sin(\theta_2) \end{pmatrix} \text{ с } \theta_1, \theta_2 \in [0, 2\pi].$$

```
theta1 = 0.5
theta2 = 1

U = np.array([[np.cos(theta1), -np.sin(theta1), 0,                    0
],
              [np.sin(theta1),  np.cos(theta1), 0,                    0
],
              [0,                    0,                  np.cos(theta2), -
np.sin(theta2)],
              [0,                     0,                   np.sin(theta2),
np.cos(theta2)]])
```

3.1.2 Схема

Мы реализуем нашу схему с 8 кубитами, 4 на входе и 4 на выходе.

```
prog = sf.Program(8)
```

Затем мы можем объявить нашу схему. Первые четыре строки – это двухрежимные затворы сжатия, которые генерируют случайное число фотонов

```
with prog.context as q:
    S2gate(r_squeezing) | (q[0], q[4])
    S2gate(r_squeezing) | (q[1], q[5])
    S2gate(r_squeezing) | (q[2], q[6])
    S2gate(r_squeezing) | (q[3], q[7])
    Interferometer(U) | (q[4], q[5], q[6], q[7])
```

3.1.3 Работа

Проведите моделирование до «отсечки» фотонов в каждом режиме

```
eng=sf.Engine("fock", backend_options={"cutoff_dim":cutoff})
state = eng.run(prog).state
```

Выход:

```
/opt/hostedtoolcache/Python/3.7.17/x64/lib/python3.7/site-
packages/strawberryfields/program.py:661:
UserWarning: The circuit consists of 2 disconnected
components.
```

Получите вероятность, связанную с каждым состоянием

```
probs = state.all_fock_probs()
```

Измените форму 'probs' так, чтобы probs $[m_1,...,m_4,n_1,...,n_4]$ дает вероятность того, что на входе будет совместно состояние $(m_1,...,m_4)$(с m_i количество фотонов в режиме входа i) и выходное состояние $(n_1,...,n_4)$ $(m_1,...,m_4)$(с n_i количество фотонов в режиме выхода i)

```
probs = probs.reshape(*[cutoff]*8)
```

Сумма не равна 1 из-за конечного отсечения:

```
np.sum(probs)
```

Выход:

```
0.9998583445961168
```

3.2 Анализ

Цель этого раздела – сравнить смоделированную вероятность с теоретической.

3.2.1 Вычисление теоретической вероятности

Для этого сначала нужно вычислить теоретическую вероятность $P(\text{input} = (m_1, m_2, m_3, m_4), \text{output} = (n_1, n_2, n_3, n_4))$, где $m_i, n_i \in \square$ представляют собой количество фотонов соответственно в режимах входа и выхода i. Используя определение условной вероятности, мы можем разложить ее как:

$$P(\text{input}, \text{output}) = P(\text{input}|\text{output})P(\text{input})$$

Значение $P(\text{input}|\text{output})$ приводится в учебном пособии «Выборка бозонов»:

$$P(\text{input} = (m_1, m_2, m_3, m_4)|\text{output} = (n_1, n_2, n_3, n_4)) = \frac{\left|Perm(U_{st})\right|^2}{n_1! n_2! n_3! n_4! m_1! m_2! m_3! m_4!}$$

в то время как $P(\text{input})$ зависит от свойств спонтанного параметрического преобразования с уменьшением (см. введение) и может быть вычислена следующим образом:

$$\begin{aligned} P(\text{input} = (m_1, m_2, m_3, m_4)) &= \prod_{i=1}^{4} P(m_i) \\ &= \prod_{i=1}^{4} (1-\chi^2)\chi^{2m_i} \\ &= (1-\chi^2)^4 \chi^{2\sum m_i} \\ &= (1-\chi^2)^m \chi^{2n} \end{aligned}$$

с m количество режимов (здесь 4) и $n = \sum m_i$ общее количество фотонов. Значение $P(m_i)$ взято непосредственно из оригинальной статьи.

Используя его, мы можем приступить к вычислениям.

Во-первых, постоянная матрицы может быть вычислена с помощью библиотеки *The Walrus*:

```
from thewalrus import perm
```

Затем вероятность выхода при заданном входе. Для этого мы используем алгоритм, приведенный в разделе V справочника (3), чтобы вычислить матрицу U_{st} (под названием $U_{I,O}$ в цитируемой статье). Если говорить кратко, то она заключается в извлечении m_j раз больше столбца *j* из *U* для каждого *j*, и n_i раз больше ряда i из *U* для каждого i (с m_j и n_i по-прежнему представляющие количество фотонов соответственно на входе *j* и выходе *i*).

```
def get_proba_output(U, input, output):
    # The two lines below are the extracted row and column
indices.
    # For instance, for output=[3,2,1,0], we want
list_rows=[0,0,0,1,1,2].
    # sum(.,[]) is a Python trick to flatten the list
    list_rows = sum([[i] * output[i] for i in
range(len(output))],[])
    list_columns = sum([[i] * input[i] for i in
range(len(input))],[])

    U_st = U[:,list_columns][list_rows,:]
    perm_squared = np.abs(perm(U_st, method="ryser"))**2
    denominator = np.prod([factorial(inp) for inp in input])
* np.prod([factorial(out) for out in output])
    return perm_squared / denominator

def get_proba_input(input):
    chi = np.tanh(r_squeezing)
    n = np.sum(input)
    m = len(input)
    return (1 - chi**2)**m * chi**(2*n)
```

```
def get_proba(U, result):
    input, output = result[0:4], result[4:8]
    return     get_proba_output(U,     input,     output)     *
get_proba_input(input) # P(O,I) = P(O|I) P(I)
```

3.2.2 Сравнение теории и моделирования

```
print("Theory: \t", get_proba(U, [0,0,0,0,0,0,0,0]))
print("Simulation: \t", probs[0,0,0,0,0,0,0,0])

print("Theory: \t", get_proba(U, [1,0,0,0,1,0,0,0]))
print("Simulation: \t", probs[1,0,0,0,1,0,0,0])

print("Theory: \t", get_proba(U, [1,0,0,0,0,1,0,0]))
print("Simulation: \t", probs[1,0,0,0,0,1,0,0])
```

Выход:

```
Theory:          0.3825422953821255
Simulation:      0.3825422953821258
Theory:          0.06291578440230364
Simulation:      0.06291578440230369
Theory:          0.018776990012967093
Simulation:      0.018776990012967107
```

Мы видим, что результаты очень похожи.

3.3 Визуализация

Чтобы наглядно представить результаты и эффект бозонного выборщика рассеянного действия, мы нарисуем несколько примеров выборки.

3.3.1 Сделайте так, чтобы сумма вероятностей равнялась 1

Из-за вычислительных проблем сумма вероятностей не равна 1. Так как это мешает нам сделать правильную выборку, мы решили добавить недостающий вес к исходу [0,0,0,0, 0,0,0,0].

```
probs[0,0,0,0, 0,0,0,0] += 1 - np.sum(probs)
np.sum(probs)
```

Выход:

```
1.0
```

3.3.2 Образец

Получите все возможные варианты в виде списка результатов $[m_1, m_2, m_3, m_4, n_1, n_2, n_3, n_4]$

```
list_choices = list(itertools.product(*[range(cutoff)]*8))
list_choices[0]
```

Выход:

```
(0, 0, 0, 0, 0, 0, 0, 0)
```

Получите вероятность каждого показателя выбора

```
list_probs = [probs[list_choices[i]] for i in
range(len(list_choices))]
list_probs[0]
```

Выход:

```
0.38268395078600903
```

Сделайте выборку, используя это распределение вероятностей

```
choice =
list_choices[np.random.choice(range(len(list_choices)),
p=list_probs)]
choice
```

Выход:

```
(1, 1, 1, 0, 2, 0, 0, 1)
```

3.3.3 Визуализация

3.3.3.1 Константы

```
## Colors
color_interf = colormap[0]
color_lines = "black"
color_laser = colormap[3]
color_photons = "#F5D76E"
color_spdc = colormap[4]
color_meas = colormap[1]

color_text_interf = "white"
color_text_spdc = "white"
color_text_measure = "white"

## Sizes
unit = 0.05
radius_photons = 0.015
margin_photons = 0.01
margin_input_meas = 1*unit # space between the end of the
input measure and the interferometer

width_laser = 8*unit
width_spdc = 2*unit
width_lines_spdc = 1*unit
width_line_interf = 8*unit
width_measure = 2*unit
width_line_input = width_line_interf - margin_input_meas -
width_measure
width_interf = 8*unit
width_line_output = 6*unit

height_interf = 20*unit
height_spdc = 2*unit
```

```
## Positions
x_begin_laser = -0.5
x_begin_spdc = x_begin_laser + width_laser
x_end_spdc = x_begin_spdc + width_spdc
x_begin_lines_spdc = x_end_spdc
x_end_lines_spdc = x_end_spdc + width_lines_spdc
x_end_line_input = x_end_lines_spdc + width_line_input
x_begin_input_meas = x_end_line_input
x_end_input_meas = x_begin_input_meas + width_measure
x_begin_interf = x_end_lines_spdc + width_line_interf
x_end_interf = x_begin_interf + width_interf
x_end_line_output = x_end_interf + width_line_output
x_end_output_meas = x_end_line_output + width_measure

y_begin_interf = 0
sep_lines_interf = height_interf / 5
sep_lines_spdc = 2*unit
```

3.3.3.2 Описание

```
# Sampling

choice =
list_choices[np.random.choice(range(len(list_choices)),
p=list_probs)]

# Plot

fig, ax = plt.subplots()
fig.set_size_inches(12, 9)
fig.axis = "equal"

interf = mpatches.Rectangle((x_begin_interf,0),width_interf,
height_interf,

edgecolor=color_interf,facecolor=color_interf)
ax.add_patch(interf)

plt.text(x_begin_interf+width_interf/2,
y_begin_interf+height_interf/2, 'U',
         {'ha': 'center', 'va': 'center'}, size=40,
color=color_text_interf)

for i_line in range(4):
    y_line_interf = y_begin_interf + (i_line+1) *
sep_lines_interf
    y_line_input = y_line_interf + sep_lines_spdc
    y_line_laser = y_line_interf + (y_line_input -
y_line_interf) / 2
```

```
    # draw laser lines
    plt.plot([x_begin_laser,x_begin_spdc],
[y_line_laser,y_line_laser], color=color_laser)

    # draw lines for the output of the SPDC
    plt.plot([x_begin_lines_spdc,x_end_lines_spdc],
[y_line_laser,y_line_interf], color=color_lines)
    plt.plot([x_begin_lines_spdc,x_end_lines_spdc],
[y_line_laser,y_line_input], color=color_lines)

    # draw lines interferometer lines
    plt.plot([x_end_lines_spdc,x_begin_interf],
[y_line_interf,y_line_interf], color=color_lines)
    plt.plot([x_end_interf, x_end_line_output],
[y_line_interf,y_line_interf], color=color_lines)

    # draw lines for the input photons (before measure)
    plt.plot([x_end_lines_spdc,x_end_line_input],
[y_line_input,y_line_input], color=color_lines)

    # draw the input measures
    input_meas = mpatches.Rectangle((x_begin_input_meas,
y_line_input-width_measure/2),width_measure,width_measure,

edgecolor=color_meas,facecolor=color_meas)
    plt.text(x_begin_input_meas+width_measure/2,
y_line_input, str(choice[i_line]),
             {'ha': 'center', 'va': 'center'}, size=12,
color=color_text_measure)
    ax.add_patch(input_meas)

    # draw the output measures
    input_meas = mpatches.Rectangle((x_end_line_output,
y_line_interf-width_measure/2),width_measure,width_measure,

edgecolor=color_meas,facecolor=color_meas)
    plt.text(x_end_line_output+width_measure/2,
y_line_interf, str(choice[4+i_line]),
             {'ha': 'center', 'va': 'center'}, size=12,
color=color_text_measure)
    ax.add_patch(input_meas)

    # draw the SPDC
    spdc =
mpatches.Rectangle((x_begin_spdc,y_line_interf),width_spdc,he
ight_spdc,

edgecolor=color_spdc,facecolor=color_spdc, zorder=3)
    plt.text(x_begin_spdc+width_spdc/2,
y_line_interf+height_spdc/2, 'SPDC',
             {'ha': 'center', 'va': 'center'}, size=12,
color=color_text_spdc)
```

```
    ax.add_patch(spdc)

    # draw the input photons
    for i_photon in range(choice[i_line]):
        x_photon = x_end_line_input - margin_photons -
radius_photons - i_photon*(radius_photons*2 + margin_photons)
        circle = mpatches.Circle([x_photon,y_line_input],
radius_photons, color=color_photons, zorder=3)
        ax.add_patch(circle)

    # draw the output photons
    for i_photon in range(choice[4 + i_line]):
        x_photon = x_end_line_output - margin_photons -
radius_photons - i_photon*(radius_photons*2 + margin_photons)
        circle = mpatches.Circle([x_photon,y_line_interf],
radius_photons, color=color_photons, zorder=3)
        ax.add_patch(circle)

plt.title("Choice: {}".format(choice))
plt.axis('equal')
plt.axis('off')
```

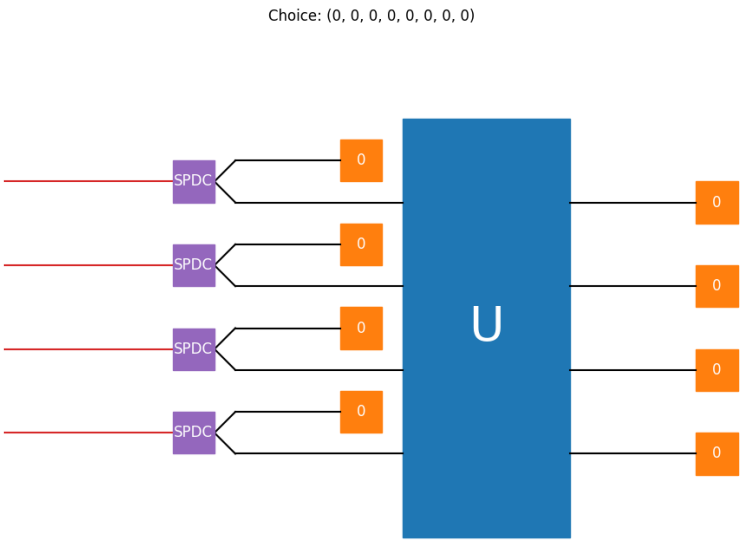

Рисунок 1 – Пример сэмплирования

Выход:

```
(-0.5875, 1.3375000000000004, -0.05, 1.05)
```

На рисунке 1 представлен пример сэмплирования (при каждом выполнении ячейка сэмплирует новое состояние).

В момент времени 0 лазер попадает в 4 SPDC, которые в результате производят *n* пар фотонов. Для каждой пары один фотон отправляется в измерительное устройство (на вход), а другой – в интерферометр. Затем интерферометр выдает эти *n* фотонов, но в разных режимах (разные линии на рисунке), следуя распределению вероятности, описанному выше. Измерительное устройство фиксирует эти выходные фотоны.

Состояние состоит как из входных фотонов (произведенных SPDC), так и из выходных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. A. P. Lund, A. Laing, S. Rahimi-Keshari, T. Rudolph, J. L O'Brien and T. C. Ralph. Boson Sampling from Gaussian States. Physical Review Letters, doi:10.1103/PhysRevLett.113.100502.

2. Scott Aaronson. Scattershot Boson Sampling: A new approach to scalable Boson Sampling experiments. Blog article.

3. Max Tillmann, Borivoje Dakić, René Heilmann, Stefan Nolte, Alexander Szameit, Philip Walther. Experimental Boson Sampling. Nature Photonics doi:10.1038/nphoton.2013.102.

Практическая работа №5. «Гауссово клонирование»

1 Цель работы

Изучение методов гауссового клонирования квантовых состояний.

2 Основная теория

Фундаментальная концепция квантовой механики, теорема об отсутствии клонирования утверждает, что неизвестное квантовое состояние не может быть скопировано точно – по сути, исключая любой алгоритм, который пытается создать идеальные копии произвольного квантового состояния или полагается на них. Тем не менее, теорема об отсутствии клонирования не исключает получения приближенных клонов квантового состояния.

Это привело к разработке так называемых «алгоритмов квантового клонирования», унитарных преобразований клонирования, которые обеспечивают идентичные копии произвольного входного состояния ценой точности, отличной от единицы.

3 Порядок выполнения работы

3.1 Реализация

Первый приближенный алгоритм клонирования был представлен в контексте квантовых вычислений с дискретными переменными Бузеком и Хиллери, и за ним быстро последовала реализация *CV* Серфом и др.

Здесь представлен класс машин для клонирования, удовлетворяющих *ковариации смещения*; то есть для двух входных состояний $|\psi\rangle$ и $|\phi\rangle$, с приблизительными состояниями клонирования $|\psi'\rangle$ и $|\phi'\rangle$ соответственно:

$$D(\alpha)|\psi\rangle=|\phi\rangle \quad \Rightarrow \quad D(\alpha)||\psi'\rangle=||\phi'\rangle$$

Другими словами, точность клонирования инвариантна относительно перемещений в фазовом пространстве, а неопределенности положения и импульса двух клонов удовлетворяют неравенству неопределенностей:

$$\Delta x_1 \Delta p_2 \geq \frac{1}{2}\hbar,$$

где $\Delta x1$ является x квадратура вариация состояния $|\psi\rangle$ и Δp_2 является p квадратура отклонения государственной $|\phi\rangle$.

Алгоритмы гауссовского клонирования, позволяющие создавать два приблизительных и идентичных клона входного состояния с теоретически оптимальной точностью, являются подклассом клонировщиков с ковариантом смещения, которые также демонстрируют вращательную ковариацию. В результате точность гауссовского клонирования инвариантна как при перемещении, так и при повороте в фазовом пространстве, и результирующие клонированные состояния имеют одинаковые x и p квадратурные отклонения, $\Delta x1=p1=\Delta x2=\Delta p2$. Показано, что именно этот подкласс является *CV*-эквивалентом универсального клонирования кубитов Бузека и Хиллери.

3.2 Анализ схем

Работая в рамках этой структуры, Андерсен и др. представили алгоритм симметричного гауссова клонирования (в дополнение к экспериментальным результатам) для оптимального теоретического клонирования когерентных состояний:

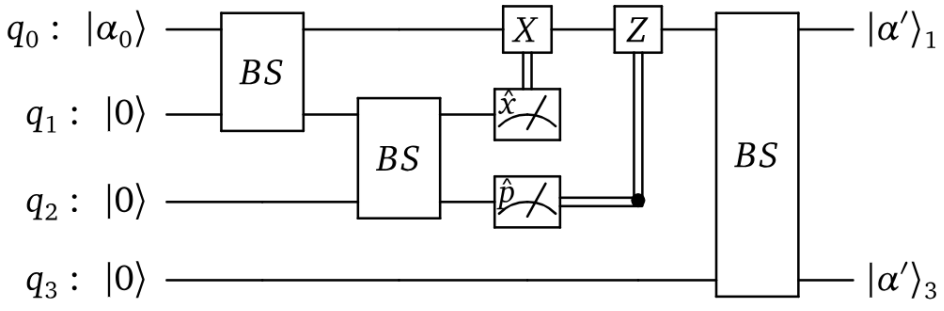

Рисунок 1 – Структурная схема алгоритма симметричного гауссова клонирования

Здесь, |α0⟩ представляет входное когерентное состояние, |α'⟩1| и |α'⟩3| представляют собой два идентичных, но приблизительных клона, а светоделители составляют 50 на 50 светоделителей (отсюда «симметричный» в алгоритме симметричного клонирования).

Давайте пройдемся по различным этапам схемы, описанной выше, и рассмотрим, что происходит.

1. Действие светоделителя 50 на 50 на когерентное состояние |α⟩ и вакуумное состояние |0⟩ это:

$$BS(|\alpha\rangle \otimes |0\rangle) = |\frac{1}{\sqrt{2}}\alpha\rangle \otimes |\frac{1}{\sqrt{2}}\alpha\rangle$$

Таким образом, после двух светоделителей схема существует в следующем состоянии:

$$\left|\frac{1}{\sqrt{2}}\alpha_0\right\rangle \otimes \left|\frac{1}{2}\alpha_0\right\rangle \otimes \left|\frac{1}{2}\alpha_0\right\rangle .$$

2. Выполнение гомодинного детектирования в режимах $q1$ и $q2$ результатом являются две нормально распределенные измеряемые переменные u и v соответственно:

$$u \sim N\left(\sqrt{\frac{\hbar}{2}}\mathrm{Re}(\alpha_0), \frac{\hbar}{2}\right), \quad v \sim N\left(\sqrt{\frac{\hbar}{2}}\mathrm{Im}(\alpha_0), \frac{\hbar}{2}\right).$$

3. Два контролируемых перемещения $X(\sqrt{2}u)=D(u/\sqrt{\hbar})$ и $Z(\sqrt{2}v)=D(iv/\sqrt{\hbar})$ затем выполняются в режиме $q0$.

$$D\left(\frac{1}{\sqrt{\hbar}}(u+iv)\right)\left|\frac{1}{\sqrt{2}}\alpha_0\right\rangle = \left|\frac{1}{\sqrt{2}}\alpha_0 + \frac{1}{\sqrt{\hbar}}(u+iv)\right\rangle = |\tilde{\alpha_0}\rangle$$

Поскольку мы перемещаем когерентное состояние, результатом контролируемых перемещений остается чистое когерентное состояние. Однако, поскольку параметры контролируемых перемещений сами по себе являются случайными величинами, мы должны описать результирующее когерентное состояние случайной переменной $\widetilde{\alpha_0}$~N(μ,cov).

Здесь, $\widetilde{\alpha_0}$ распределяется случайным образом в соответствии с многомерным нормальным распределением с вектором средних μ=√2(Re(α0),Im(α0)) и ковариационная матрица cov=I/2.

Наконец, мы применим другой светоделитель для режима $q0$ и режим $q3$ в вакуумном состоянии, чтобы получить два наших клонированных результата:

$$BS(|\tilde{\alpha_0}\rangle \otimes |0\rangle) = \left|\frac{1}{\sqrt{2}}\tilde{\alpha_0}\right\rangle \otimes \left|\frac{1}{\sqrt{2}}\tilde{\alpha_0}\right\rangle = |\alpha'\rangle \otimes |\alpha'\rangle .$$

где α'~$N(\mu,cov)$, $\mu=(Re(\alpha 0),Im(\alpha 0))$, $cov=I/4\alpha$.

3.3 Когерентная средняя точность

Если бы мы выполнили схему клонирования по Гуассиану для ансамбля идентичных входных состояний |α0⟩ клонированный результат может быть описан следующим смешанным состоянием,

$$\rho = \iint d^2\alpha' \frac{2}{\pi} e^{-2|\alpha'-\alpha_0|^2} |\alpha'\rangle \langle\alpha'| ,$$

где экспоненциальный член представляет собой *PDF* случайной величины α′ из (4) выше. Чтобы вычислить среднюю точность по ансамблю клонированных состояний, достаточно вычислить внутреннее произведение:

$$F = \langle\alpha_0 \mid \rho \mid \alpha_0\rangle.$$

От ФОКа основе разложения целостного государства (см. когерентное состояние), то можно легко видеть, что $|\langle \alpha_0 \mid \alpha' \rangle|^2 = \mathrm{e}^{-|\alpha_0 - \alpha'|^2}$

Следовательно,

$$F = \frac{2}{\pi} \iint d^2\alpha' e^{-2|\alpha' - \alpha_0|^2} |\langle \alpha_0 \mid \alpha' \rangle|^2 = \frac{2}{\pi} \iint d^2\alpha \, e^{-3|\alpha|^2} = \frac{2}{3},$$

где мы сделали замену α=α'−. Обратите внимание, что средняя точность не зависит от начального состояния α0.

Примечание:

Приведенное выше вычислено в случае единичной квантовой эффективности η=1. Когда η<1η<1, существует ненулевая неопределенность в гомодинном измерении, σH=1-/., и на практике симметричная схема гауссова клонирования имеет среднюю точность клонирования по состоянию, заданную:

$$F(\sigma_H) = \frac{2}{3 + \sigma_H}.$$

В случае гауссовского бэкенда, σH=2×10^4(см. *GaussianBackend.measure_homodyne*).

3.4 Перемещенные сжатые состояния

В дополнение к когерентным состояниям, эта схема клонирования была дополнительно проанализирована Оливаресом и др. [7] в случаях других гауссовских входных состояний, таких как сжатые состояния и тепловые состояния. В частности, когда входное гауссово состояние является смещенным сжатым состоянием,

$$|\psi\rangle = |\alpha, z\rangle = D(\alpha) S(z) |0\rangle.$$

Эта схема обеспечивает оптимальную точность 2/3 *только* в том случае, если параметр сжатия известен заранее, поскольку это позволяет применить унитарную операцию $S(z)^{-1}$ восстановить когерентное состояние перед клонированием. Конечно, это невозможно, если мы хотим клонировать произвольное неизвестное смещенное сжатое состояние; в этом случае схема, описанная выше, приводит к следующей точности.:

$$F(r,\sigma_H) = \frac{4}{\sqrt{(6+2\sigma_H)^2 + 32(1+\sigma_H)\sinh^2(r)}}$$

где $z=re^{i\phi}$ и σH это неопределенность в гомодинном измерении.

Обратите внимание, что $F \to 0$ как $r \to \infty$; т.е. чем более сильно сжато состояние, тем ниже точность клонирования.

3.5 Код

Симметричная схема гауссовского клонирования, показанная выше, может быть реализована с использованием *Strawberry Fields*:

```
import strawberryfields as sf
from strawberryfields.ops import *
from numpy import pi, sqrt
import numpy as np
gaussian_cloning = sf.Program(4)
with gaussian_cloning.context as q:
    # state to be cloned
    alpha = 0.7+1.2j
    Coherent(np.abs(alpha), np.angle(alpha)) | q[0]
    # 50-50 beamsplitter
    BS = BSgate(pi/4, 0)
    # symmetric Gaussian cloning scheme
    BS | (q[0], q[1])
    BS | (q[1], q[2])
    MeasureX | q[1]
    MeasureP | q[2]
    Xgate(q[1].par * sqrt(2)) | q[0]
    Zgate(q[2].par * sqrt(2)) | q[0]
    # after the final beamsplitter, modes q[0] and q[3]
    # will contain identical approximate clones of the
    # initial state Coherent(0.1+0j)
    BS | (q[0], q[3])
```

Поскольку все операции и измерения выполняются по гауссу, мы можем использовать серверную часть по гауссу:

```
eng = sf.Engine(backend="gaussian")
```

Режимы 1 и 2 являются вспомогательными режимами; нас интересует извлечение режимов 0 и 3.

```
results = eng.run(gaussian_cloning, modes=[0, 3])
```

После построения схемы и запуска движка мы можем вызвать метод *state fidelity_coherent()* для вычисления точности двух клонированных выходных состояний по сравнению с входным когерентным состоянием $\alpha=0.7+1.2j$.

Обратите внимание, что мы извлекаем квадратный корень, поскольку метод возвращает точность умножения обоих режимов.

```
fidelity = sqrt(results.state.fidelity_coherent([0.7+1.2j,
0.7+1.2j]))
print(fidelity)
```

Выход:

```
0.5572817882335409
```

Хотя *fidelity_coherent* поддерживается как серверными системами *Gaussian*, так и *Fock, 'gaussian'* серверная часть дополнительно поддерживает извлечение среднего смещения выходных состояний, используя *displacement()*:

```
alpha = results.state.displacement()
```

Проверка того, что они являются идентичными клонами с точностью до числовой ошибки:

```
print(alpha[0] - alpha[1] <= 1e-15)
```

Выход:

```
True
```

Чтобы вычислить среднюю точность по ансамблю, нам нужно будет запустить схему несколько раз и вычислить среднюю точность по всем прогонам:

```
# run the engine over an ensemble
reps = 1000
f = np.empty([reps])
a = np.empty([reps], dtype=np.complex128)
for i in range(reps):
    eng.reset()
    results = eng.run(gaussian_cloning, modes=[0])
    f[i] = results.state.fidelity_coherent([0.7+1.2j])
    a[i] = results.state.displacement()
print("Fidelity of cloned state:", np.mean(f))
print("Mean displacement of cloned state:", np.mean(a))
print("Mean covariance matrix of cloned state:",
np.cov([a.real, a.imag]))
```

Выход:

```
Fidelity of cloned state: 0.6598662763090357
Mean displacement of cloned state: (0.7124677707668282+1.2145
178434535502j)
Mean covariance matrix of cloned state: [[0.25597159
0.00122702]
```

```
[0.00122702 0.25298511]]
```

Выводя диаграмму рассеяния *a.real* против *a.imag* (рис. 2), мы видим, что они действительно распределены как многомерное нормальное распределение со средним значением $\sim 0.7+1.2j$ и ковариация $\sim I/4$:

```
import seaborn as sns
sns.set(style="ticks")
sns.jointplot(a.real, a.imag, color="#4CB391")
```

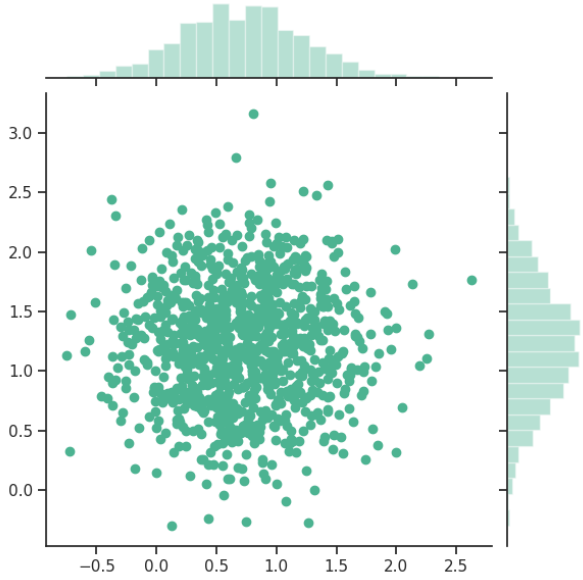

Рисунок 2 – Диаграмма рассеяния

Выход:

```
<seaborn.axisgrid.JointGrid object at 0x7f13ff19c8d0>
```

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. P. van Loock and Samuel L. Braunstein. Telecloning of continuous quantum variables. Physical Review Letters, Nov 2001. doi:10.1103/physrevlett.87.247901.

2. W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. Nature, 299(5886):802–803, Oct 1982. doi:10.1038/299802a0.

3. D. Dieks. Communication by EPR devices. Physics Letters A, 92(6):271–272, Nov 1982. doi:10.1016/0375-9601(82)90084-6.

4. V. Bužek and M. Hillery. Quantum copying: beyond the no-cloning theorem. Physical Review A, 54(3):1844–1852, Sep 1996. doi:10.1103/physreva.54.1844.

5. N. J. Cerf, A. Ipe, and X. Rottenberg. Cloning of continuous quantum variables. Physical Review Letters, 85(8):1754–1757, Aug 2000. doi:10.1103/physrevlett.85.1754.

6. Ulrik L. Andersen, Vincent Josse, and Gerd Leuchs. Unconditional quantum cloning of coherent states with linear optics. Physical Review Letters, Jun 2005. doi:10.1103/physrevlett.94.240503.

7. Stefano Olivares, Matteo G. A. Paris, and Ulrik L. Andersen. Cloning of gaussian states by linear optics. Physical Review A, Jun 2006. doi:10.1103/physreva.73.062330.