Изучаем квантовые вычисления на Python и Q#

Сара Кайзер
Кристофер Гранад

MANNING

ДМК издательство

Сара Кайзер и Кристофер Гранад

Изучаем квантовые вычисления на Python и Q#

Learn Quantum Computing with Python and Q#

A HANDS-ON APPROACH

SARAH KAISER
AND **CHRIS GRANADE**

MANNING
Shelter Island

Изучаем квантовые вычисления на Python и Q#

ПРАКТИЧЕСКИЙ ПОДХОД

САРА КАЙЗЕР
и **КРИСТОФЕР ГРАНАД**

ДМК Пресс
издательство
Москва, 2021

УДК 519.588
ББК 22.314
К15

Кайзер С., Гранад К.
К15 Изучаем квантовые вычисления на Python и Q# / пер. с анг. А. В. Логунова. – М.: ДМК Пресс, 2021. – 430 с.: ил.

ISBN 978-5-97060-935-4

Технологический прорыв, связанный с распространением квантовых компьютеров, уже не за горами. В этой книге технологии будущего обсуждаются с практической стороны: комплект инструментов от компании Microsoft и язык Q# предоставляют вам возможность поупражняться в квантовых вычислениях.

В части I вы создадите симулятор квантового устройства на языке Python, в части II научитесь применять новые навыки написания квантовых приложений с помощью языка Q# и Комплекта инструментов для квантовой разработки, а в части III – имплементировать алгоритм, который умножает целые числа экспоненциально быстрее, чем самый лучший из известных стандартных алгоритмов.

Издание предназначено для разработчиков программного обеспечения. Предварительного опыта работы с квантовыми вычислениями, а также знания математики или физики на продвинутом уровне не требуется.

УДК 519.588
ББК 22.314

ISBN 978-1-6172-9613-0 (англ.)
ISBN 978-5-97060-935-4 (рус.)

Мы посвящаем эту книгу
следующему поколению квантовых разработчиков,
которые работают над тем, чтобы сделать эту сферу
более безопасной и инклюзивной

Оглавление

Содержание

Вводное слово

На протяжении большей части своей истории квантовые вычисления являлись полем деятельности физиков – хотя, возможно, некоторые из них и имели склонность к компьютерным наукам, это было вовсе не обязательно так. Популярный учебник «Квантовые вычисления и квантовая информация» Майкла А. Нильсена и Исаака Л. Чжуана (*Quantum Computation and Quantum Information*, Michael A. Nielsen and Isaac L. Chuang) до сих пор является одним из лучших и был написан двумя квантовыми физиками. Конечно же, ученые-компьютерщики всегда были рядом, но некоторые теоретики носят на себе в качестве знака отличия то, что они написали мало строк кода. Это квантовый мир, и в нем я, Кайзер и Гранад достигли совершеннолетия. Я мог бы легко погрозить кулаком новой когорте студентов и прикрикнуть на них, что, дескать, когда я был в вашем возрасте, мы не писали исходный код – мы задыхались от меловой пыли!

Я познакомился с Крисом Гранадом, когда мы оба были аспирантами. Тогда мы писали статьи в академические журналы по физике со строками исходного кода, но они отклонялись редакцией за то, что «это не является физикой». Однако нас это не остановило. И теперь, много лет спустя, данная книга представляет для меня окончательное оправдание! Она научит вас всему, что вы когда-либо захотите и что вам потребуется узнать о квантовых вычислениях, без необходимости в физике – хотя если вы действительно хотите узнать связь с физикой, то Кайзер и Гранад предлагают и это 😆? И смайлики тоже есть 😊!

С тех пор я прошел долгий путь, и я многим обязан Гранаду, как и сфере квантовых вычислений, за понимание того, что между прилагательным «квантовый» и существительным «вычисления» существует нечто большее, чем просто теоремы и доказательства. Кайзер также научила меня большему, чем я полагал, о необходимости участия инженера-программиста в разработке квантовых технологий. Кайзер и Гранад превратили свой опыт в слова и строки кода, чтобы все могли извлечь из них пользу, как и я.

Хотя изначально цель состояла в том, чтобы создать «не учебник», эта книга, безусловно, могла бы быть использована как таковая в уни-

верситетской аудитории, так как введение в квантовые вычисления постепенно перемещается с физических факультетов в школы информатики. Интерес к квантовым вычислениям огромен и продолжает расти, и бóльшая его часть исходит не от физики – разработчики программно-информационного обеспечения, операционные менеджеры и финансовые руководители – все хотят знать, что такое квантовые вычисления и как их заполучить в свои руки. Прошли те времена, когда квантовые вычисления были чисто академическим занятием. Эта книга служит потребностям растущего квантового сообщества.

Хотя я и ссылался на снижение доли физиков в сфере квантовых вычислений, я не хочу сбрасывать их со счетов. Подобно тому как и я когда-то был луддитом разработки программно-информационного обеспечения, настоящая книга в действительности предназначена для всех, в особенности для тех, кто уже работает в этой сфере и хочет узнать о программной стороне квантовых вычислений в знакомой обстановке.

Запустите свой любимый редактор кода и приготовьтесь напечатать print(«Привет, квантовый мир!»).

КРИС ФЕРРИ,
доктор философии, доцент,
Центр квантового программно-информационного обеспечения,
Сидней, Новый Южный Уэльс, Австралия

Предисловие

Квантовые вычисления были нашей страстью на протяжении более 20 лет вместе взятых, и мы с энтузиазмом берем свой опыт и используем его для того, чтобы вовлечь в квантовые технологии еще больше людей. Мы вместе защитили докторские диссертации, и при этом мы с немалым трудом преодолевали исследовательские вопросы, соревнования на каламбуры и настольные игры, помогая раздвигать границы того, что было возможно при участии кубитов. По большей части это означало разработку нового программно-информационного обеспечения и инструментов, которые будут помогать нам и нашим коллективам лучше проводить исследования, которые являлись великолепным мостом между «квантовой» и «вычислительной» частями данного предмета. Однако при разработке различных программно-информационных проектов нам нужно было научить наших коллег-разработчиков тому, над чем мы работаем. При этом мы все время задавались вопросом, а почему нет хорошей книги по квантовым вычислениям, которая имела бы техническую направленность, но не была бы учебником. Так вот. То, на что вы сейчас смотрите, является ответом на этот вопрос. 💜

Мы написали эту книгу так, чтобы она была доступна разработчикам, не в стиле учебника, который так характерен для других книг по квантовым вычислениям. Когда мы сами изучали квантовые вычисления, этот процесс был очень захватывающим, но в то же время немного страшил и пугал. Это не обязательно должно быть именно так, поскольку многое из того, что делает квантовые вычисления запутанными, связано не с их содержимым, а с тем, как они подаются.

К сожалению, квантовые вычисления часто описываются как «странные», «жуткие» или как находящиеся за пределами нашего понимания, тогда как истина заключается в том, что квантовые вычисления за свою 35-летнюю историю уже довольно хорошо изучены. Используя комбинацию разработки программно-информационного обеспечения и математики, вы можете строить базовые концепции, необходимые для понимания квантовых вычислений, и проводить разведку этой удивительной новой технологической сферы.

Наша цель в этой книге состоит в том, чтобы помочь вам изучить основы технологии и снабдить вас инструментами, которые вы сможете использовать для строительства квантовых решений завтрашнего дня. В центре нашего внимания будет практический опыт разработки исходного кода для квантовых вычислений. В части I вы создадите свой собственный симулятор квантового устройства на языке Python; в части II вы научитесь применять свои новые навыки для написания квантовых приложений с помощью языка Q# и Комплекта инструментов для квантовой разработки; и в части III вы научитесь имплементировать алгоритм, который умножает целые числа экспоненциально быстрее, чем самый лучший обычный алгоритм из известных на сегодняшний день. Это будет вашей работой на всем протяжении, которая и составит *ваше* квантовое путешествие.

Мы включили в него как можно больше практических приложений, но самое замечательное состоит в том, что как раз в них вы и будете участвовать! Квантовые вычисления находятся на острие технологии, откуда они двинутся вперед, и нам нужен мост между огромным объемом того, что известно о возможностях квантовых компьютеров, и задачами, которые люди должны решать. Построив этот мост, мы перейдем от квантовых алгоритмов, которые были созданы для больших исследований, к квантовым алгоритмам, которые могут повлиять на все общество в целом. И вы поможете построить этот мост. Добро пожаловать в квантовое путешествие; мы здесь, чтобы помочь сделать его увлекательным!

Признательности

Берясь за эту книгу, мы с самого начала не знали, во что ввязываемся; единственное, что мы знали, – так это, что такой ресурс должен существовать. Написание книги дало нам огромную возможность усовершенствовать и развить наши навыки в объяснении и преподавании предмета, с которым мы были знакомы. Все специалисты, с которыми мы работали в издательстве Manning, были замечательными – Дейрдре Хайам, наш редактор-постановщик; Тиффани Тейлор, наш редактор-копирайтер; Кэти Теннант, наш корректор; Иван Мартинович, наш редактор-рецензент, – и они помогли нам обеспечить, чтобы эта книга стала лучшей из всех, которые могут быть для наших читателей.

Мы благодарим Оливию Ди Маттео и Криса Ферри за все их ценные отзывы и замечания, которые помогли сделать наши объяснения точными и ясными.

Мы также благодарим всех рецензентов рукописи, которые рассматривали ее на различных стадиях разработки и чьи вдумчивые отзывы сделали эту книгу намного лучше: Алена Элиота, Клайва Харбера, Дэвида Рэймонда, Дебмалью Яш, Дмитрия Денисенко, Доминго Салазара, Эммануэля Медину Лопеса, Джеффа Кларка, Хавьера, Картикеяраяна Раджендрана, Кшиштофа Камычека, Кумара Унникришнана, Паскуале Зирполи, Патрика Ригана, Пола Отто, Рафаэлля Вентальо, Рональда Тишляра, Сандера Зегвельда, Стива Сассмана, Тома Хеймана, Туан А. Трана, Уолтера Александр Мата Лопеса и Уильяма Э. Уилера.

Мы благодарим всех подписчиков Программы раннего доступа издательства Manning (MEAP), которые помогли найти ошибки, опечатки и места для улучшения объяснений. Многие участники также предоставили обратную связь, разместив вопросы в нашем репозитории с образцами кода: мы благодарим их всех!

Мы хотели бы отметить множество замечательных заведений в районе Сиэтла (в частности, Caffe Ladro, Miir, Milstead & Co. и Downpour Coffee Bar), которые позволяли нам пить кофе чашку за чашкой и оживленно дискутировать о кубитах, а также замечательных людей из Fremont Brewing, которые всегда были рядом, когда нам нужна была пинта пива.

Всегда было приятно прерываться от работы, когда случайный прохожий задавал нам вопросы о том, над чем мы работаем!

Мы также хотели бы поблагодарить талантливых членов команды Quantum Systems в Microsoft за работу по предоставлению разработчикам наилучших инструментов для внедрения квантовых вычислений. В частности, мы благодарим Беттину Хейм за работу над тем, чтобы сделать Q# удивительным языком, а также за то, что она была хорошим другом.

Наконец, мы благодарим нашу немецкую овчарку Чуи, которая обеспечивала нам столь необходимые развлечения и причины для того, чтобы отдохнуть от компьютера.

САРА КАЙЗЕР

Моя семья всегда была рядом со мной, и я благодарю их за терпение и поддержку во время работы над этим проектом. Хотела бы поблагодарить своего терапевта, без которого эта книга никогда бы не появилась. Но больше всего хочу поблагодарить своего соавтора и партнера Криса. Он был со мной, несмотря ни на что, и всегда поощрял и вдохновлял меня делать то, что я умела, в чем он никогда не сомневался. 💔

КРИС ГРАНАД

Эта книга была бы невозможна без удивительной любви и поддержки моего партнера и соавтора, доктора Сары Кайзер. Вместе мы прошли через многое и достигли большего, чем я когда-либо мечтал. Наша совместная история всегда была посвящена созданию более профессионального, более безопасного и инклюзивного квантового сообщества, и эта книга является чудесной возможностью сделать еще один шаг на этом пути. Спасибо тебе, Сара, за все это.

И она была бы невозможной без поддержки моей семьи и друзей. Спасибо вам за то, что вы были всегда рядом, будь то обмен фотографиями милых щенков, сочувствие по поводу последних новостных заголовков или участие в наблюдении за ночным метеорным дождем в игре Animal Crossing. Наконец, я также благодарю фантастическое интернет-сообщество, на которое я опирался на протяжении многих лет, пытаясь понять мир со многих новых точек зрения.

О книге

Добро пожаловать в книгу «Изучаем квантовые вычисления на Python и Q#»! Эта книга познакомит вас с миром квантовых вычислений, используя язык программирования Python в качестве удобной отправной точки, строя решения, написанные на специализированном языке программирования Q#, разработанном в компании Microsoft. Мы используем подход, основанный на примерах и играх, к обучению квантовым вычислениям и концепциям разработки, которые помогут вам сразу же приступить к написанию исходного кода.

Глубокое погружение: плавать с маской и трубкой – это нормально!

Квантовые вычисления – это обширная междисциплинарная сфера исследований, объединяющая в себе идеи из программирования, физики, математики, машиностроения и компьютерных наук. Время от времени на протяжении всей книги мы будем уделять время тому, чтобы указывать на то, как идеи из этих разнообразных областей используются в квантовых вычислениях, помещая изучаемые понятия в этот богатый контекст.

В то время как эти отступления служат для того, чтобы вызывать любопытство и побуждать на дальнейшие исследования, они по своей природе тангенциальны. Из этой книги вы получите все необходимое для того, чтобы наслаждаться квантовым программированием на Python и Q#, независимо от того, окунаетесь ли вы в эти глубокие пучины или нет. Глубокое погружение иногда будет веселым и поучительным, но если оно не является вашим коньком, то ничего; плавать с маской и трубкой – совершенно нормально.

Кто должен прочитать эту книгу

Эта книга предназначена для тех, кто интересуется квантовыми вычислениями и практически не имеет опыта работы с квантовой механикой, но имеет некоторый опыт программирования. При обучении писать квантовые симуляторы на Python и квантовые программы на Q#,

специализированном языке компании Microsoft для квантовых вычислений, мы используем традиционные идеи и методы программирования, чтобы помочь вам в этом. При этом будет полезно общее понимание концепций программирования, таких как циклы, функции и присвоения значений переменным.

Мы также используем несколько математических понятий из линейной алгебры, таких как векторы и матрицы, которые помогают нам описывать квантовые понятия; если вы знакомы с компьютерной графикой или машинным обучением, то многие из этих понятий похожи. В ходе работы мы будем использовать Python для обзора наиболее важных математических понятий, но знакомство с линейной алгеброй будет полезным.

Как эта книга организована: дорожная карта

Данный текст призван помочь вам начать изучение и использование практических инструментов для квантовых вычислений. Книга разбита на три части, причем каждая последующая опирается на предыдущую:

- в части I шадяще вводятся понятия, необходимые для описания *кубитов*, фундаментальной единицы квантового компьютера. В указанной части описывается способ симулирования кубитов на языке Python, который упрощает написание простых квантовых программ;
- в части II описывается использование Комплекта инструментов для квантовой разработки и язык программирования Q# для создания кубитов и выполнения квантовых алгоритмов, работа которых отличается от любых известных классических алгоритмов;
- в части III мы применяем инструменты и методы из предыдущих двух частей, чтобы научиться применять квантовые компьютеры к реально-прикладным задачам, таким как симулирование химических свойств.

Кроме того, имеется четыре приложения к книге. В приложении A содержатся все инструкции по инсталляции инструментов, которые мы используем в этой книге. Приложение B представляет собой краткий справочный раздел с квантовым глоссарием, напоминаниями о математической нотации и фрагментами исходного кода, которые могут оказаться полезными во время чтения книги. Приложение C представляет собой памятку, которая освежит вашу память по линейной алгебре, а приложение D является глубоким погружением в один из алгоритмов, который вы будете имплементировать.

Об исходном коде

Весь используемый в этой книге исходный код для книг издательства «ДМК Пресс» можно найти на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги. Полные инструкции

по инсталляции доступны в репозитории этой книги и в приложении А к книге.

Прилагаемые к книге образцы исходного кода также можно выполнить онлайн без какого-либо инсталлирования, используя службу mybinder.org. Для начала работы в указанной службе перейдите по ссылке https://bit.ly/qsharp-book-binder.

Дискуссионный форум liveBook

Покупка книги «Изучаем квантовые вычисления на Python и Q#» включает в себя бесплатный доступ к частному веб-форуму издательства Manning Publications, где вы можете комментировать книгу, задавать технические вопросы и получать помощь от авторов и других пользователей. В целях получения доступа к указанному форуму перейдите по ссылке https://livebook.manning.com/#!/book/learnquantum-computing-with-python-and-q-sharp/discussion. О форумах издательства Manning и правилах поведения можно узнать больше на веб-странице https://livebook.manning.com/#!/discussion.

Другие онлайновые ресурсы

Когда, прочитав эту книгу и поработав с предоставленными примерами исходного кода, вы начнете самостоятельное путешествие по квантовым вычислениям, вам окажутся полезными следующие онлайновые ресурсы:

- документация по Комплекту инструментов для квантовой разработки Quantum Development Kit (https://docs.microsoft.com/azure/quantum/) – концептуальная документация и полная ссылка на все, что касается языка Q#, включая изменения и дополнения с момента публикации этой книги;
- образцы Комплекта инструментов для квантовой разработки (https://github.com/microsoft/quantum) – полные примеры использования языка Q#, как самостоятельно, так и с главными программами на Python и в .NET, охватывающие широкий спектр различных приложений;
- QuTiP.org (http://qutip.org) – полное руководство пользователя для пакета QuTiP, который мы использовали, чтобы помочь с математикой в этой книге.

Кроме того, также имеется ряд замечательных сообществ как для экспертов по квантовым вычислениям, так и для новичков. Присоединение к сообществу квантовых разработчиков, подобному приведенным ниже, поможет решать вопросы, возникающие у вас на этом пути, а также позволит вам помогать другим в их путешествиях:

- qsharp.community (https://qsharp.community) – сообщество пользователей и разработчиков на языке Q#, в комплекте с чатом, блогом и репозиториями проектов;

- Quantum Computing Stack Exchange (Биржа по обмену информацией о квантовых вычислениях, https://quantumcomputing.stackexchange.com/) – отличное место для получения ответов на вопросы о квантовых вычислениях, включая любые вопросы по Q#, которые у вас могут возникнуть;
- Women in Quantum Computing and Applications (Женщины в квантовых вычислениях и приложениях, https://wiqca.dev) – инклюзивное сообщество для людей всех полов, чтобы прославлять квантовые вычисления и людей, которые делают их возможными;
- Quantum Open Source Foundation (Квантовый фонд открытых исходных кодов, https://qosf.org/) – сообщество, поддерживающее разработку и стандартизацию открытых инструментов для квантовых вычислений;
- Unitary Fund (Унитарный фонд, https://unitary.fund/) – некоммерческая организация, работающая над созданием экосистемы квантовых технологий, которая приносит пользу большинству людей.

Идем дальше

Квантовые вычисления – это увлекательная новая сфера технологий, которая предлагает новые способы мышления о вычислениях и новые инструменты для решения сложных задач. Эта книга поможет вам начать заниматься квантовыми вычислениями и обеспечит основу для продолжения своих занятий в этой сфере. Тем не менее данная книга не является учебником и не предназначена для того, чтобы подготовить вас к самостоятельным исследованиям квантовых вычислений. Как и в случае с классическими алгоритмами, разработка новых квантовых алгоритмов – это математическое искусство, как и все остальное; хотя мы и касаемся математики в этой книге и используем ее для объяснения алгоритмов, есть масса других учебников, которые помогут вам развить идеи, которые мы рассматриваем.

Если вы захотите продолжить свое путешествие по физике или математике после прочтения этой книги и приступили к квантовым вычислениям, мы предлагаем один из следующих ресурсов:

- Зоопарк сложности (https://complexityzoo.net/Complexity_Zoo);
- Зоопарк квантовых алгоритмов (http://quantumalgorithmzoo.org);
- «Теория сложности: современный подход» Санджива Ароры и Боаза Барака (*Complexity Theory: A Modern Approach*, Sanjeev Arora and Boaz Barak, Cambridge University Press, 2009);
- «Квантовые вычисления: щадящее введение» Элеоноры Г. Риффель и Вольфганга Х. Полака (*Quantum Computing: A Gentle Introduction*, Eleanor G. Rieffel and Wolfgang H. Polak, MIT Press, 2011);
- «Квантовые вычисления со времен Демокрита» Скотта Ааронсона (*Quantum Computing since Democritus*, Scott Aaronson, Cambridge University Press, 2013);
- «Квантовые вычисления и квантовая информация» Майкла А. Нильсена и Исаака Л. Чжуана (*Quantum Computation and Quantum Infor-*

mation, Michael A. Nielsen and Isaac L. Chuang, Cambridge University Press, 2000);
- «Системы квантовых процессов и информация» Бенджамина Шумахера и Майкла Уэстморленда (*Quantum Processes Systems, and Information*, Benjamin Schumacher and Michael Westmoreland, Cambridge University Press, 2010).

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторах

САРА КАЙЗЕР защитила докторскую диссертацию по физике (квантовая информация) в Институте квантовых вычислений Университета Ватерлоо. Она провела большую часть своей карьеры, разрабатывая новое квантовое оборудование в лаборатории, от создания спутников до взлома оборудования на основе квантовой криптографии. Донесение до интересующихся людей захватывающей информации о квантовых вычислениях является ее страстью. Она любит создавать новые демоверсии и инструменты, которые будут помогать расти квантовому сообществу. Когда она не сидит за своей механической клавиатурой, она любит кататься на байдарках и писать книги о науке для всех возрастов.

КРИС ГРАНАД защитил докторскую диссертацию по физике (квантовая информация) в Университете Института квантовых вычислений Ватерлоо и теперь трудится в группе квантовых систем Microsoft. Он работает над созданием стандартных библиотек для языка Q# и является экспертом в статистическом описании квантовых устройств на основе классических данных. Крис также помог Скотту Ааронсону подготовить его лекции в виде книги «Квантовые вычисления со времен Демокрита» (*Quantum Computing Since Democritus*, Cambridge University Press, 2013).

Об иллюстрации на обложке

Фигура на обложке книги «Изучаем квантовые вычисления на Python и Q#» озаглавлена «Hongroise», или венгерка. Иллюстрация взята из коллекции костюмов из разных стран Жака Грассе де Сен-Совера (1757–1810) под названием Costumes de Différents Pays, опубликованной во Франции в 1797 году. Каждая иллюстрация тонко прорисована и раскрашена вручную. Богатое разнообразие коллекций Грассе-де-Сен-Совера живо напоминает нам о том, насколько культурно обособленными были города и регионы мира всего 200 лет назад.

Изолированные друг от друга люди говорили на разных диалектах и языках. На улицах городов или в селах было легко определить местность, где люди живут, и их ремесло или положение в жизни, просто по их одежде. С тех пор наша одежда изменилась, и региональное разнообразие, столь богатое в то время, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных городах, регионах или странах. Возможно, мы променяли культурное разнообразие на более разнообразную личную жизнь – конечно же, на более разнообразную и быстро развивающуюся технологическую жизнь.

В то время когда трудно отличить одну компьютерную книгу от другой, издательство Manning прославляет изобретательность и инициативу компьютерного бизнеса с помощью обложек книг, основанных на богатом разнообразии региональной жизни двухвековой давности, оживленной картинами Грассе де Сен-Совера.

Часть I

Приступаем к работе с квантом

Эта часть книги поможет подготовить почву для остальной части нашего квантового путешествия. В главе 1 мы узнаем больше о квантовых вычислениях, о подходе к изучению квантовых вычислений в этой книге и о том, где мы можем их использовать для применения полученных навыков. В главе 2 мы начнем писать исходный код, разработав квантовый симулятор на языке Python. Затем мы применим этот симулятор для программирования квантового генератора случайных чисел. Далее, в главе 3, мы расширим симулятор для программирования криптографических приложений квантовой технологии, таких как протокол обмена квантовыми ключами BB84. В главе 4 мы применим нелокальные игры, чтобы узнать о запутанности, и снова расширим симулятор с целью поддержания нескольких кубитов. В главе 5 мы научимся использовать новый пакет Python для имплементирования квантовых стратегий для игры в нелокальные игры из главы 4. Наконец, в главе 6 мы в последний раз расширим наш симулятор, добавив новые квантовые операции, чтобы иметь возможность симулировать такие технические приемы, как *квантовая телепортация*, и практиковаться в перемещении данных в наших квантовых устройствах.

1 Введение в квантовые вычисления

Эта глава охватывает следующие ниже темы:

- почему люди восторгаются квантовыми вычислениями;
- что такое квантовый компьютер;
- что может и чего не может делать квантовый компьютер;
- как квантовые компьютеры соотносятся с классическим программированием.

В последние несколько лет квантовые вычисления становились все более популярной сферой исследований и источником шумихи. Используя квантовую физику для выполнения вычислений новыми и замечательными способами, *квантовые компьютеры* могут повлиять на общество, что придает нынешнему времени большую привлекательность в плане участия и изучения того, как программировать квантовые компьютеры и применять квантовые ресурсы для решения важных задач.

Однако во всей этой шумихе по поводу преимуществ квантовых вычислений легко упустить из виду *реальный* масштаб этих преимуществ. У нас есть интересный исторический прецедент того, что может произойти, когда обещания в отношении технологии опережают реальность. В 1970-х годах машинное обучение и искусственный интеллект пострадали от резкого сокращения финансирования, поскольку шумиха и ажиотаж вокруг ИИ превзошли его результаты; позже этот период будет назван «зимой ИИ». Интернет-компании столкнулись с той же опасностью аналогичным образом, пытаясь преодолеть крах доткомов.

Один из путей продвижения вперед состоит в том, чтобы критически понять обещания, предлагаемые квантовыми вычислениями, принцип работы квантовых компьютеров и что входит и не входит в сферу применения квантовых вычислений. В этой главе мы поможем вам развить это понимание, чтобы вы могли получить практический опыт и написать свои собственные квантовые программы в остальной части книги.

Однако, помимо всего этого, просто очень здорово узнать о совершенно новой вычислительной модели! Читая эту книгу, вы узнаете о том, как работают квантовые компьютеры, путем программирования симуляций, которые вы можете выполнять на своем ноутбуке уже сегодня. Эти симуляции покажут многие существенные элементы того, что мы ожидаем от реального коммерческого квантового программирования в то время, пока полезное коммерческое оборудование выходит в сеть. Эта книга предназначена для людей, которые имеют некоторый базовый опыт в программировании и линейной алгебре, но не имеют предварительных знаний о квантовой физике или квантовых вычислениях. Если у вас есть некоторые знания в этих сферах, то вы можете перейти к частям II и III, где мы займемся квантовым программированием и алгоритмами.

1.1 Почему квантовые вычисления имеют значение?

Вычислительные технологии развиваются поистине ошеломляющими темпами. Три десятилетия назад процессор 80486 позволял пользователям выполнять 50 MIPS (миллионов инструкций в секунду). Сегодня крохотные компьютеры, такие как Raspberry Pi, могут достигать 5000 MIPS, в то время как настольные процессоры могут легко достигать 50 000–300 000 MIPS. Если у нас есть исключительно сложная вычислительная задача, которую мы хотели бы решить, то очень разумная стратегия состоит в том, чтобы просто дождаться следующего поколения процессоров, которые сделают нашу жизнь проще, потоковое видео быстрее, а наши игры красочнее.

Однако в отношении многих волнующих нас задач нам не так повезло. Мы могли бы надеяться, что получение процессора, который работает в два раза быстрее, позволит нам решать задачи, которые в два раза больше, но, как и многое в жизни, «больше значит иначе». Предположим, мы отсортируем список из 10 миллионов чисел и обнаружим, что это занимает около 1 секунды. Позже, если мы захотим отсортировать список из 1 миллиарда чисел за 1 секунду, нам понадобится процессор, который должен будет работать не в 100 раз, а в 130 раз быстрее. При решении некоторых задач это становится еще хуже: в случае некоторых графических задач переход от 10 миллионов к 1 миллиарду точек займет в 13 000 раз больше времени.

Такие разнообразные задачи, как маршрутизация дорожного движения в городе и предсказывание химических реакций, усложняются *значительно* быстрее. Если бы квантовые вычисления всецело касались создания компьютера, который работает в 1000 раз быстрее, то мы едва ли смогли бы справиться с огромными задачами, которые мы хотим решать. К счастью, квантовые компьютеры *гораздо* интереснее. Мы ожидаем, что квантовые компьютеры будут работать намного *медленнее*, чем классические компьютеры, но что ресурсы, необходимые для решения многих задач, будут *масштабироваться* по-другому, вследствие чего, если мы обратимся к правильным типам задач, то мы сможем преодолеть правило «больше значит иначе». В то же время квантовые компьютеры не являются волшебной пулей – некоторые задачи останутся трудными. Например, хотя вполне вероятно, что квантовые компьютеры очень помогут нам в предсказании химических реакций, они едва помогут в решении других сложных задач.

Исследование применимости квантовых вычислений, т. е. в каких именно задачах мы можем получить квантовое преимущество, и разработка квантовых алгоритмов для них были в центре внимания исследований в сфере квантовых вычислений. До сих пор было очень трудно оценивать квантовые подходы в таком ключе, поскольку для написания квантовых алгоритмов требовались обширные математические навыки и понимание всех тонкостнй квантовой механики.

Однако по мере того, как промышленность начала разрабатывать платформы, помогающие подключать разработчиков к квантовым вычислениям, эта ситуация начала меняться. Используя весь Комплект инструментов для квантовой разработки компании Microsoft, мы можем абстрагироваться от большинства математических сложностей квантовых вычислений и начать реально *понимать* и *использовать* квантовые компьютеры. Описанные в этой книге инструменты и методы позволяют разработчикам изучить и понять то, на что будет похоже написание программ для этой новой аппаратной платформы.

Иными словами, квантовые вычисления никуда не денутся, поэтому понимание того, какие задачи мы можем решать с их помощью, и вправду имеет большое значение! Независимо от того, произойдет квантовая «революция» или нет, квантовые вычисления в значительной степени учитывали – и будут продолжать учитывать – технологические решения о том, как развивать вычислительные ресурсы в течение следующих нескольких десятилетий. Следующие ниже решения находятся под сильным влиянием квантовых вычислений:

- какие допущения являются разумными в области информационной безопасности?
- какие навыки полезны в образовательных программах?
- как оценивать рынок вычислительных решений?

Те из нас, кто работает в области технологий или смежных областях, все чаще должны принимать такие решения или вносить в них свой вклад. Мы несем ответственность за понимание того, чем являются квантовые

вычисления и, что, возможно, более важно, чем они не являются. Благодаря этому мы будем лучше подготовлены к активизации и внесению своего вклада в эти новые усилия и решения.

Помимо всего прочего, еще одна причина, по которой квантовые вычисления являются такой увлекательной темой, заключается в том, что они одновременно похожи и сильно отличаются от классических вычислений. Понимание сходств и различий между классическими и квантовыми вычислениями помогает нам понять фундаментальные компоненты в вычислениях в целом. Как классические, так и квантовые вычисления возникают из разных описаний физических законов, вследствие чего понимание вычислений поможет нам понять Вселенную по-новому.

Вместе с тем абсолютно важно понимать, что нет ни одной правильной или даже самой лучшей причины интересоваться квантовыми вычислениями. Что бы ни привело вас к исследованиям или приложениям в сфере квантовых вычислений, вы непременно узнаете что-то интересное на этом пути.

1.2 Что такое квантовый компьютер?

Давайте немного поговорим о том, из чего на самом деле состоит квантовый компьютер. В целях облегчения обсуждения этой темы кратко остановимся на смысле термина «компьютер».

ОПРЕДЕЛЕНИЕ *Компьютер* – это устройство, которое принимает данные на входе и выполняет какие-то операции с этими данными.

Существует масса примеров того, что мы называем *компьютером*; несколько примеров приведено на рис. 1.1.

Все, что показано ниже, объединено тем, что мы можем моделировать их с помощью классической физики, т. е. в терминах законов движения Ньютона, ньютоновской гравитации и электромагнетизма.

Это поможет нам различить типы компьютеров, к которым мы привыкли (например, ноутбуки, телефоны, хлебопечки, дома, автомобили и кардиостимуляторы), и компьютеры, о которых мы узнаем в этой книге. В целях проведения различия между обоими компьютерами те, которые можно описать классической физикой, мы будем называть *классическими компьютерами*. Это хорошо тем, что если мы заменим термин «классическая физика» на *квантовую физику*, то получим отличное определение квантового компьютера!

ОПРЕДЕЛЕНИЕ *Квантовый компьютер* – это устройство, которое принимает данные на входе и выполняет какие-то операции с этими данными с помощью процесса, который может быть описан только квантовой физикой.

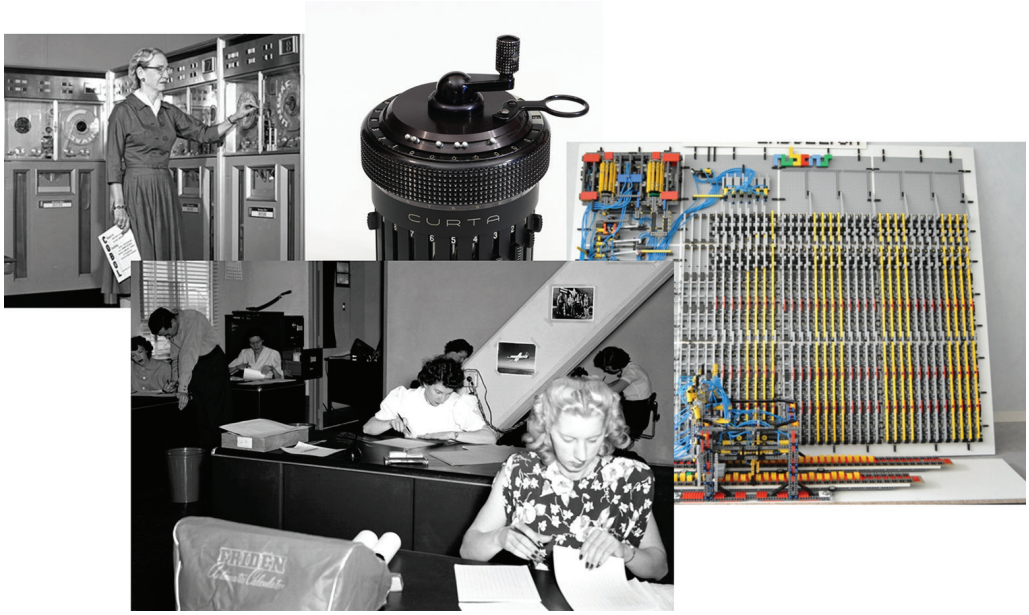


Рис. 1.1 Несколько примеров различных типов компьютеров, включая мейнфрейм UNIVAC, управляемый контр-адмиралом Хоппером, комнату с «человеческими компьютерами», работающими над полетными вычислениями, механический калькулятор и машину Тьюринга на основе LEGO. Каждый компьютер может быть описан той же математической моделью, что и такие компьютеры, как мобильные телефоны, ноутбуки и серверы. Источники: фотография «человеческих компьютеров» принадлежит NASA. Фотография машины Тьюринга из LEGO принадлежит Projet Rubens и используется в рамках CC BY 3.0 (https://creativecommons.org/licenses/by/3.0/**)**

Иными словами, различие между классическими и квантовыми компьютерами сводится к различию между классической и квантовой физикой. Мы рассмотрим это в книге подробнее чуть позже. Но первостепенное различие заключается в масштабе: наш повседневный опыт в основном связан с объектами, которые достаточно велики и достаточно горячи, чтобы, несмотря на то что квантовые эффекты все же существуют, они в среднем мало что делали. Хотя квантовая механика работает даже в масштабе повседневных объектов, таких как кофейные кружки, мешки с мукой и бейсбольные биты, оказывается, что мы можем очень хорошо описывать характер взаимодействия этих объектов, используя только классическую физику.

Если мы сосредоточимся на гораздо меньшем масштабе, где для описания наших систем необходима квантовая механика, то квантовые вычисления – это искусство использования малых, хорошо изолированных устройств для полезного преобразования данных методами, которые нельзя описать только в терминах классической физики. Один из подходов к строительству квантовых устройств состоит в использовании малых классических компьютеров, таких как цифровые сигнальные процессоры (DSP), чтобы контролировать свойства экзотических материалов.

Глубокое погружение: что случилось с теорией относительности?

Квантовая физика применима к объектам, которые очень малы и очень холодны либо хорошо изолированы. По аналогии, другая область физики, именуемая *теорией относительности*, описывает объекты, которые достаточно велики, чтобы гравитация играла важную роль, либо которые движутся очень быстро – почти со скоростью света. Многие компьютеры полагаются на релятивистские эффекты. И действительно, спутники глобального позиционирования критически зависят от теории относительности. До сих пор мы в основном сравнивали классическую и квантовую физику, так как же насчет теории относительности?

Как оказалось, все вычисления, имплементируемые с использованием релятивистских эффектов, также могут быть описаны с использованием чисто классических моделей вычислений, таких как машины Тьюринга. Напротив, квантовые вычисления невозможно описать как более быстрые классические вычисления, и они требуют другой математической модели. До сих пор не делалось никаких предложений о «гравитационном компьютере», который использовал бы теорию относительности в таком же ключе, и, следовательно, в этой книге мы можем с уверенностью отложить теорию относительности в сторону.

Физика и квантовые вычисления

Экзотические материалы, используемые для строительства квантовых компьютеров, имеют названия, которые могут показаться устрашающими, например *сверхпроводники* и *топологические изоляторы*. Однако мы можем найти утешение в том, как мы учимся понимать и использовать классические компьютеры.

Мы можем программировать классические компьютеры, не зная, что такое полупроводник. Проще говоря, физика, лежащая в основе того, как мы строим квантовые компьютеры, представляет собой увлекательный предмет исследования, но нам не обязательно учиться программировать и использовать квантовые устройства.

Квантовые устройства могут отличаться в деталях того, как они контролируются, но в конечном счете все квантовые устройства контролируются и считываются классическими компьютерами и какой-либо контрольной электроникой. В конце концов, нас интересуют классические данные, поэтому в конечном итоге должен существовать интерфейс с классическим миром.

ПРИМЕЧАНИЕ Большинство квантовых устройств должны храниться в очень холодном и хорошо изолированном состоянии, так как они могут быть чрезвычайно восприимчивы к шуму.

Применяя квантовые операции с использованием встроенного классического оборудования, мы можем манипулировать и преобразовывать

квантовые данные. И тогда мощь квантовых вычислений будет проистекать из тщательного выбора операций, которые следует применять для имплементирования полезного преобразования, решающего интересующую вас задачу.

1.3 Как мы будем использовать квантовые компьютеры?

Рис. 1.2 Способы, которыми мы хотели бы использовать квантовые компьютеры. Комикс используется с разрешения xkcd.com

Важно понимать потенциал и ограничения квантовых компьютеров, в особенности учитывая шумиху вокруг квантовых вычислений. Многие недоразумения, лежащие в основе этой шумихи, проистекают из экстраполяции аналогий за пределы того, где они имеют какой-либо смысл, – все аналогии имеют свои пределы, и квантовые вычисления ничем не отличаются. Симулирование действий квантовой программы на практике бывает отличным методом, который помогает проверять и уточнять понимание, обеспечиваемое аналогиями. Тем не менее в этой книге мы все равно будем использовать аналогии, поскольку они помогут дать представление о принципах работы квантовых вычислений.

> **ДЛЯ СПРАВКИ** Если вы когда-либо видели описания новых результатов в квантовых вычислениях, которые гласят, что, дескать, «мы можем телепортировать кошек, которые находятся в двух местах одновременно, используя силу бесконечного числа параллельных вселенных, работающих вместе, чтобы излечить от рака», то вы столкнулись с опасностью экстраполирования слишком далеко от того, где аналогии приносят пользу.

Одна особенно распространенная путаница в отношении квантовых вычислений заключается в том, как пользователи будут использовать квантовые компьютеры. Мы, как общество, пришли к пониманию сути *компьютера*: это то, что вы можете использовать для выполнения веб-приложений, написания документов и запуска симуляций. На самом

деле классические компьютеры делают так много разных вещей в нашей жизни, что мы даже не всегда замечаем, что является компьютером, а что нет. Кори Доктороу как-то заметил, что «ваш автомобиль – это компьютер, в котором вы сидите» (Ключевые тезисы из конференции DrupalCon-2014, Амстердам, www.youtube.com/watch?v=iaf3Sl2r3jE).

Однако квантовые компьютеры, скорее всего, будут гораздо более специализированными – мы ожидаем, что квантовые компьютеры будут бесполезны для отдельных задач. Отличная модель того, как квантовые вычисления будут вписываться в наш существующий стек классических вычислений, представлена графическими процессорами. Графические процессоры (GPU) – это специализированные аппаратные устройства, предназначенные для ускорения конкретных типов вычислений, таких как рисование графики, задачи машинного обучения, и всего того, что легко параллелизуется. Для этих конкретных задач вам нужен графический процессор, но, скорее всего, вы не захотите использовать его для всего спектра задач, поскольку у нас есть гораздо более гибкие процессоры для общих задач, таких как проверка электронной почты. Квантовые компьютеры будут точно такими же: они будут хороши для ускорения конкретных типов задач, но не будут пригодны для широкого использования.

ПРИМЕЧАНИЕ Программирование квантового компьютера сопряжено с некоторыми ограничениями, поэтому классические компьютеры будут предпочтительнее, когда нет никаких особых квантовых преимуществ.

Классические вычисления по-прежнему будут существовать и будут главенствующим способом общения и взаимодействия друг с другом, а также с нашим квантовым оборудованием. Даже для того, чтобы получить классический вычислительный ресурс для взаимодействия с квантовыми устройствами, в большинстве случаев нам также понадобится процессор цифроаналоговых сигналов, как показано на рис. 1.3.

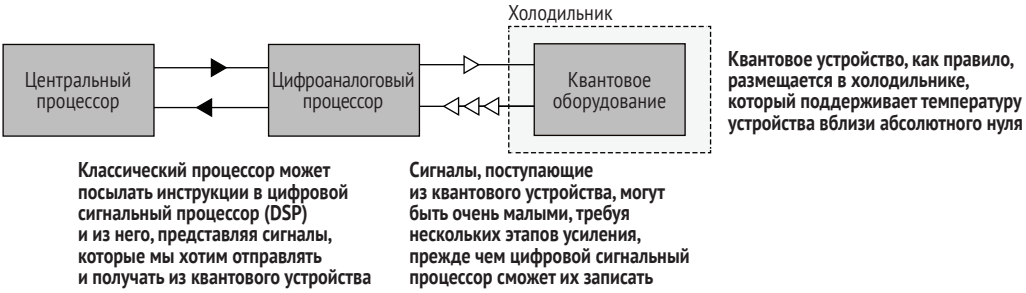


Рис. 1.3 Пример того, как квантовое устройство может взаимодействовать с классическим компьютером с помощью цифрового сигнального процессора (DSP). DSP посылает маломощные сигналы в квантовое устройство и усиливает очень маломощные сигналы, возвращающиеся в устройство

Более того, квантовая физика описывает вещи в очень малых масштабах (как по размеру, так и по энергии), которые хорошо изолированы от окружающей среды. Это накладывает некоторые жесткие ограничения на среду, в которой мы можем запускать квантовый компьютер. Одним из возможных решений является хранение квантовых устройств в криогенных холодильниках, часто при температуре вблизи абсолютного 0 K (–459.67 °F, или –273.15 °C). Хотя это не проблема в центре обработки данных, поддержание работы криогенного холодильника на самом деле не имеет смысла на рабочем столе, а тем более на ноутбуке или мобильном телефоне. По этой причине квантовые компьютеры, скорее всего, будут использоваться через облако, по крайней мере в течение довольно долгого времени после того, как они впервые станут коммерчески доступными.

Использование квантовых компьютеров в качестве облачной службы напоминает другие достижения в области специального вычислительного оборудования. Благодаря централизации экзотических вычислительных ресурсов, таких как приведенные ниже в центрах обработки данных, можно выполнять разведку вычислительных моделей, которые трудно развертывать локально всем, кроме крупнейших пользователей:

- специализированное игровое оборудование (PlayStation Now, Xbox One);
- кластеры высокопроизводительных вычислений с чрезвычайно низкой задержкой (например, Infiniband) для научных задач;
- массивные кластеры GPU;
- перепрограммируемое оборудование (например, Catapult/Brainwave);
- кластеры тензорных процессоров (TPU);
- архивное хранилище с высокой стабильностью и высокой задержкой (например, Amazon Glacier).

В будущем облачные службы, такие как Azure Quantum (https://azure.com/quantum), сделают возможности квантовых вычислений доступными во многом в таком же ключе.

Подобно тому, как высокоскоростное высокодоступное интернет-соединение сделало облачные вычисления доступными для большого числа пользователей, мы сможем использовать квантовые компьютеры, не покидая наш любимый пляж, или кафе с Wi-Fi-покрытием, или даже поезд, когда мы смотрим издали на величественные горные хребты.

1.3.1 Что квантовые компьютеры могут делать?

Если у нас есть конкретная задача, то как мы, квантовые программисты, узнаем, *имеет ли смысл ее решать с помощью квантового компьютера*?

Мы все еще познаем точную степень того, на что способны квантовые компьютеры, и поэтому у нас пока что нет никаких конкретных правил, чтобы ответить на этот вопрос. До настоящего времени мы нашли несколько примеров задач, в которых квантовые компьютеры предлагают

значительные преимущества по сравнению с наилучшими классическими подходами из известных на сегодняшний день. В каждом случае квантовые алгоритмы, которые, как выяснилось, решают эти задачи, задействуют квантовые эффекты для достижения преимуществ, иногда именуемых *квантовыми преимуществами*. Ниже приведены два полезных квантовых алгоритма:

- алгоритм Гровера (обсуждаемый в главе 11) выполняет поиск по списку из N элементов за $\sqrt{N}$ шагов;
- алгоритм Шора (глава 12) быстро вычисляет большие целые числа, которые, в частности, используются в криптографии для защиты приватных данных.

В этой книге мы увидим еще несколько алгоритмов, но Гровер и Шор являются хорошими примерами того, как работают квантовые алгоритмы: в каждом из них используются квантовые эффекты, чтобы отделять правильные ответы на вычислительные задачи от невалидных решений. Один из способов реализации квантового преимущества состоит в отыскании подходов к использованию квантовых эффектов для отделения правильных решений классических задач от неправильных.

Каковы квантовые превосходства?

Алгоритмы Гровера и Шора иллюстрируют два отличимых вида квантовых преимуществ. Выполнять разложение целых чисел на факторы, возможно, будет проще классически, чем мы подозреваем. Многие люди очень старались выполнить быструю факторизацию целых чисел и не преуспели в решении данной задачи, но это вовсе не значит, что мы можем *доказать*, что операция факторизации является вычислительно трудной. С другой стороны, мы можем доказать, что алгоритм Гровера работает быстрее *любого* классического алгоритма; здесь загвоздка в том, что на входе в нем используется другой вид данных.

Отыскание *доказуемого* преимущества для практической задачи является активной областью исследований в сфере квантовых вычислений. Тем не менее квантовые компьютеры могут быть мощными инструментами для решения задач, даже если мы не сможем доказать отсутствие какого-либо более производительного классического алгоритма. В конце концов, алгоритм Шора бросает вызов допущениям, лежащим в основе больших отраслей информационной безопасности, – математическое доказательство необходимо только потому, что мы еще не построили достаточно большой квантовый компьютер, чтобы иметь возможность выполнять алгоритм Шора.

Квантовые компьютеры также предлагают значительные преимущества для симулирования свойств квантовых систем, открывая приложения для квантовой химии и материаловедения. Например, квантовые компьютеры могли бы значительно облегчить понимание энергий основного состояния в химических системах. Указанные энергии основного состояния затем дают представление о скорости реакции, электрон-

ных конфигурациях, термодинамических свойствах и других свойствах, представляющих огромный интерес в химии.

На пути к разработке этих приложений мы также увидели значительные преимущества в побочных технологиях, таких как квантовое распределение ключей и квантовая метрология, некоторые из которых мы увидим в следующих нескольких главах. Учась контролировать и понимать квантовые устройства для целей вычислений, мы также усвоили ценные технические приемы визуализации, оценивания параметров, обеспечения безопасности и многие другие. Хотя они и не являются приложениями для квантовых вычислений в строгом смысле, они в значительной степени демонстрируют ценность *мышления* в терминах квантовых вычислений.

Конечно же, новые приложения квантовых компьютеров гораздо легче обнаруживать, когда у нас есть конкретное понимание механизмов работы квантовых алгоритмов и процесса строительства новых алгоритмов, исходя из базовых принципов. С этой точки зрения квантовое программирование представляет собой отличный ресурс для изучения методов открытия совершенно новых приложений.

1.3.2 Чего квантовые компьютеры не могут делать?

Как и другие формы специализированного вычислительного оборудования, квантовые компьютеры не будут хороши во всем. Для некоторых задач классические компьютеры просто подходят лучше. При разработке приложений для квантовых устройств полезно учитывать то, какие задания или задачи выходят за рамки квантовых вычислений.

Короткий ответ состоит в том, что у нас нет никаких жестких правил, позволяющих быстро определять задачи, которые лучше всего выполнять на классических компьютерах, и те, в которых могут использоваться преимущества квантовых компьютеров. Например, требования к хранилищу и пропускной способности для приложений больших данных очень трудно сопоставить с квантовыми устройствами, где у нас может быть только относительно малая квантовая система. Современные квантовые компьютеры могут регистировать входные данные не более чем в несколько десятков бит, и это ограничение будет становиться все более актуальным по мере того, как квантовые устройства будут использоваться для более требовательных задач. Хотя мы ожидаем, что в конечном итоге построим гораздо более крупные квантовые системы, чем сейчас, классические вычисления, вероятно, всегда будут предпочтительнее для задач, решения которых требуют больших объемов ввода-вывода.

Аналогичным образом задачи машинного обучения, которые в значительной степени зависят от произвольного доступа к крупным наборам классических входных данных, концептуально трудно решать с помощью квантовых вычислений. Тем не менее *могут* существовать и другие задачи машинного обучения, которые гораздо более естественно соотносятся с квантовыми вычислениями. Исследовательские усилия по поиску наилучших способов применения квантовых ресурсов для

решения задач машинного обучения все еще продолжаются. В общем случае задачи, которые имеют малые размеры входных и выходных данных, но требуют больших объемов вычислений для перехода от входных данных к выходным, являются хорошими кандидатами для квантовых компьютеров.

В свете этих проблем может возникнуть соблазн прийти к заключению, что квантовые компьютеры *всегда* преуспевают в задачах, которые имеют малые входы и выходы и очень интенсивное взаимодействие между ними. Такие понятия, как *квантовый параллелизм*, популярны в средствах массовой информации, и квантовые компьютеры иногда даже описываются как использующие в вычислениях параллельные вселенные.

> **ПРИМЕЧАНИЕ** Понятие «параллельные вселенные» является отличным примером аналогии, которая помогает делать квантовые концепции понятными, но может привести к бессмысленности, если довести его до крайности. Иногда полезно думать, что разные части квантового вычисления находятся в разных вселенных, которые не могут влиять друг на друга, но это описание затрудняет размышления о некоторых эффектах, о которых мы узнаем в этой книге, таких как интерференция. Если зайти слишком далеко, то аналогия с параллельными вселенными также дает основания думать о квантовых вычислениях в терминах, которые ближе к особенно «мясистому» и забавному эпизоду научно-фантастического кинофильма «Звездный путь», чем к реальности.

Однако оно не способно передать то, что не всегда является очевидным, т. е. как использовать квантовые эффекты для извлечения полезных ответов из квантового устройства, даже если состояние квантового устройства, по всей видимости, содержит желаемый результат. Например, один из способов факторизации целого числа N с помощью классического компьютера состоит в перечислении каждого *потенциального* фактора и проверке того, является он на самом деле фактором (множителем/делителем) или нет:

1. Пусть $i = 2$.
2. Проверить на равенство нулю остатка от N/i.
   – Если равен нулю, то вернуть, что i факторизует N.
   – Если не равен нулю, то увеличить i на единицу и повторить цикл.

Мы можем ускорить этот классический алгоритм, задействовав большое число разных классических компьютеров, по одному для каждого потенциального фактора, который мы хотим попробовать. То есть эту задачу можно легко параллелизовать. Квантовый компьютер может попробовать каждый потенциальный фактор в одном и том же устройстве, но, как оказалось, для факторизации целых чисел быстрее классического подхода этого *еще* недостаточно. Если мы используем этот подход на квантовом компьютере, то на выходе будет один из потенциальных факторов, выбранных случайным образом. Фактические правильные

факторы будут возникать с вероятностью около $1/\sqrt{N}$, что не лучше, чем в случае классического алгоритма.

Однако, как мы увидим в главе 12, с помощью квантового компьютера для факторизации данных мы можем применять другие квантовые эффекты быстрее, чем самые лучшие классические алгоритмы факторизации из известных на сегодняшний день. Бóльшая часть тяжелой работы, выполняемой алгоритмом Шора, заключается в обеспечении того, чтобы вероятность измерить правильный фактор в конце была намного больше, чем вероятность измерить неправильный фактор. Бóльшая часть искусства квантового программирования заключается во взаимопогашении неправильных ответов в таком ключе; это нелегко или даже невозможно сделать для всех задач, которые мы, вероятно, захотим решить.

В целях более четкого понимания того, что квантовые компьютеры способны и не способны делать, и как делать классные вещи с квантовыми компьютерами, несмотря на эти проблемы, целесообразно использовать более конкретный подход. Итак, давайте рассмотрим суть квантовой программы, чтобы иметь возможность написать свою собственную.

1.4 Что такое программа?

На протяжении всей этой книги мы часто будем прибегать к объяснению квантовой концепции, сначала просматривая ее классический аналог. В частности, давайте сделаем шаг назад и проинспектируем понятие классической программы.

> **ОПРЕДЕЛЕНИЕ** *Программа* – это последовательность инструкций, которые могут интерпретироваться классическим компьютером для выполнения желаемой задачи. Налоговые формы, направления движения на местности, рецепты и скрипты на языке Python – все это примеры программ.

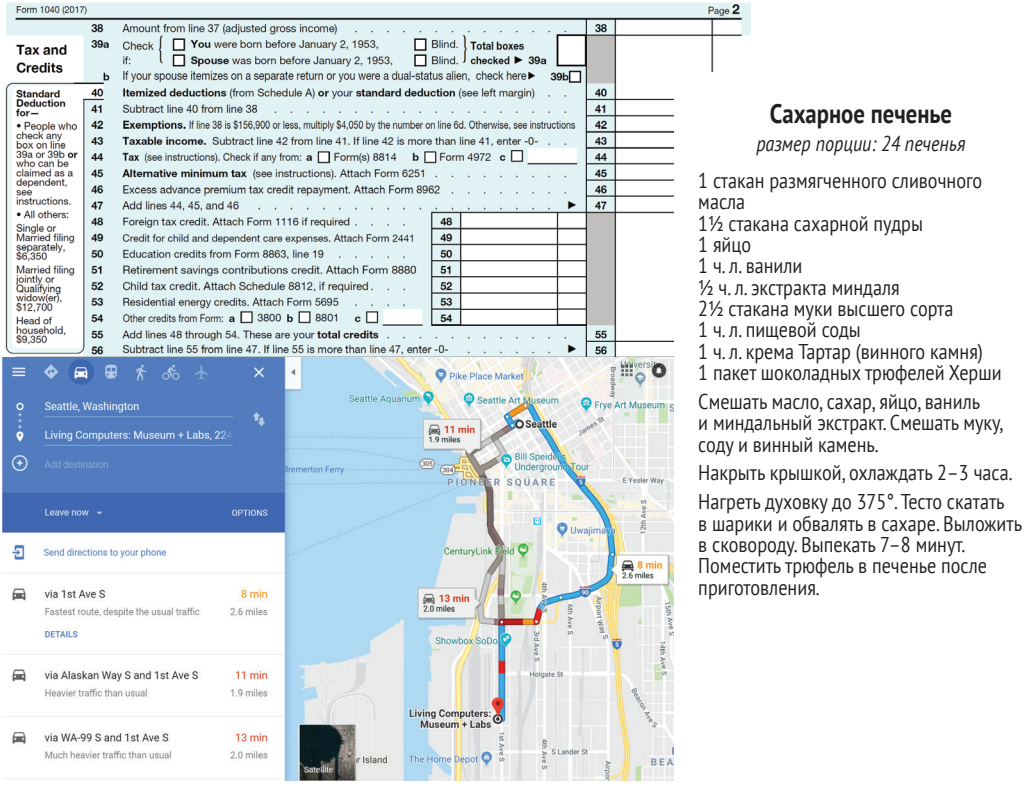
Мы можем писать классические программы, разбивая широкий спектр разных задач для их интерпретирования на самых разных компьютерах. Некоторые примеры программ приведены на рис. 1.4.

Давайте посмотрим, как может выглядеть простая программа «Привет, Мир!» на Python:

```
>>> def hello():
...     print("Привет, Мир!")
...
>>> hello()
Привет, мир!
```

По своей сути, эту программу можно рассматривать как последовательность инструкций, данных *интерпретатору* Python, который затем

выполняет каждую инструкцию по очереди, чтобы добиться некоторого эффекта – в данном случае печати сообщения на экране. То есть программа представляет собой *описание* задачи, которая затем *интерпретируется* языком Python и, в свою очередь, нашим процессором для достижения нашей цели. Это взаимодействие между описанием и интерпретацией мотивирует называть Python, C и другие подобные языки программирования именно *языками*, подчеркивая, что программирование есть то, как мы общаемся с нашими компьютерами.



Сахарное печенье

размер порции: 24 печенья

1 стакан размягченного сливочного масла
1½ стакана сахарной пудры
1 яйцо
1 ч. л. ванили
½ ч. л. экстракта миндаля
2½ стакана муки высшего сорта
1 ч. л. пищевой соды
1 ч. л. крема Тартар (винного камня)
1 пакет шоколадных трюфелей Херши

Смешать масло, сахар, яйцо, ваниль и миндальный экстракт. Смешать муку, соду и винный камень.

Накрыть крышкой, охлаждать 2–3 часа.

Нагреть духовку до 375°. Тесто скатать в шарики и обвалять в сахаре. Выложить в сковороду. Выпекать 7–8 минут. Поместить трюфель в печенье после приготовления.

Рис. 1.4 Примеры классических программ. Налоговые формы, направления на карте местности и рецепты – все это примеры, в которых последовательность инструкций интерпретируется классическим компьютером, таким как человек. Они могут выглядеть очень по-разному, но каждый из них использует список шагов для передачи информации о процедуре

В примере использования языка Python для печати сообщения «Привет, Мир!» мы практически общаемся с Гвидо ван Россумом, основателем и разработчиком языка Python. Затем Гвидо практически взаимодействует от нашего имени с дизайнерами используемой нами операционной системы. Эти дизайнеры, в свою очередь, общаются от нашего имени с Intel, AMD, ARM или любой другой компанией, разработавшей используемый нами процессор, и т. д.

1.4.1 Что такое квантовая программа?

Как и классические программы, квантовые программы состоят из последовательностей инструкций, которые интерпретируются классическими компьютерами в целях выполнения той или иной задачи. Разница, однако, заключается в том, что в квантовой программе задача, которую мы хотим выполнить, предусматривает контролирование квантовой системы для выполнения вычислений.

Как следствие инструкции, используемые в классических и квантовых программах, также различаются. Классическая программа может описывать такую задачу, как скачивание некоторых изображений кошек из Интернета, в терминах инструкций для стека сетевой обработки и в конечном счете в терминах ассемблерных инструкций, таких как mov (переместить). Напротив, квантовые языки, такие как Q#, позволяют программистам выражать квантовые задачи в терминах инструкций, таких как `M` (измерить). Во время выполнения с использованием квантового оборудования эти программы могут давать инструкции цифровому сигнальному процессору посылать микроволны, радиоволны или лазеры в квантовое устройство и усиливать сигналы, выходящие из устройства.

На протяжении остальной части этой книги мы увидим массу примеров того, с какими задачами сталкивается квантовая программа, решая или, по крайней мере, обращаясь к ним, и какие виды классических инструментов мы можем использовать в целях упрощения квантового программирования. Например, на рис. 1.5 показан пример написания квантовой программы в классической интегрированной среде разработки Visual Studio Code.

Мы будем надстраивать концепции, необходимые для написания квантовых программ, от главы к главе; на рис. 1.6 показана дорожная карта. В следующей главе мы начнем с изучения базовых строительных блоков, из которых состоит квантовый компьютер, и использования их для написания нашей первой квантовой программы.

Резюме

- Квантовые вычисления важны, потому что квантовые компьютеры потенциально позволят нам решать задачи, которые слишком трудно решить с помощью обычных компьютеров.
- Квантовые компьютеры могут обеспечивать преимущества перед классическими компьютерами для некоторых видов задач, таких как факторизация крупных чисел.
- Квантовые компьютеры – это устройства, в которых для обработки данных используется квантовая физика.
- Программы – это последовательности инструкций, которые могут интерпретироваться классическим компьютером для выполнения задач.
- Квантовые программы – это программы, которые выполняют вычисления, посылая инструкции квантовым устройствам.

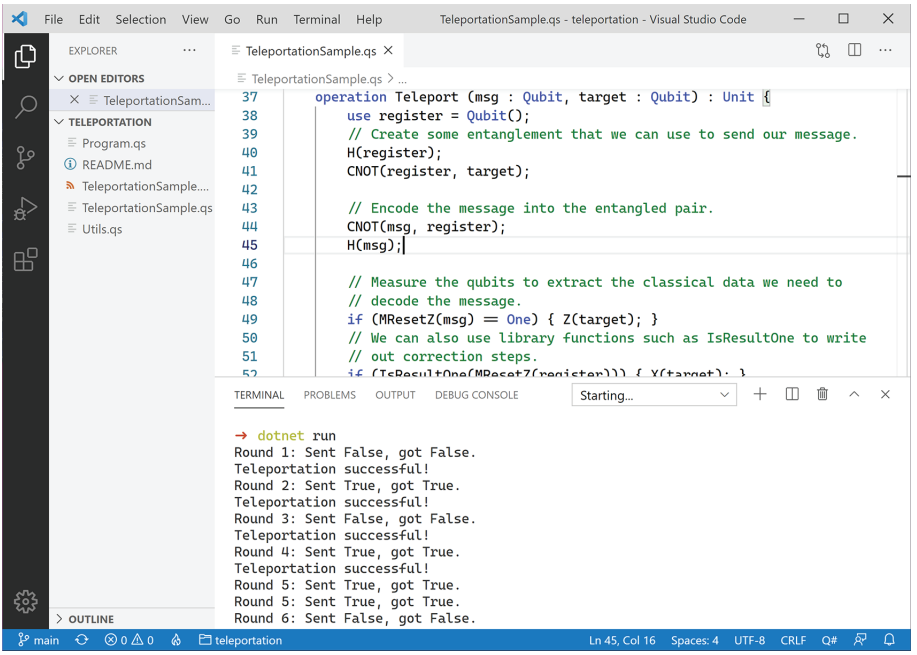


Рис. 1.5 Написание квантовой программы с помощью Комплекта инструментов для квантовой разработки и редактора Visual Studio Code. Мы рассмотрим содержание этой программы в главе 7, но сейчас вы можете окинуть ее взором и увидеть, что она похожа на другие программные проекты, над которыми вы, возможно, работали

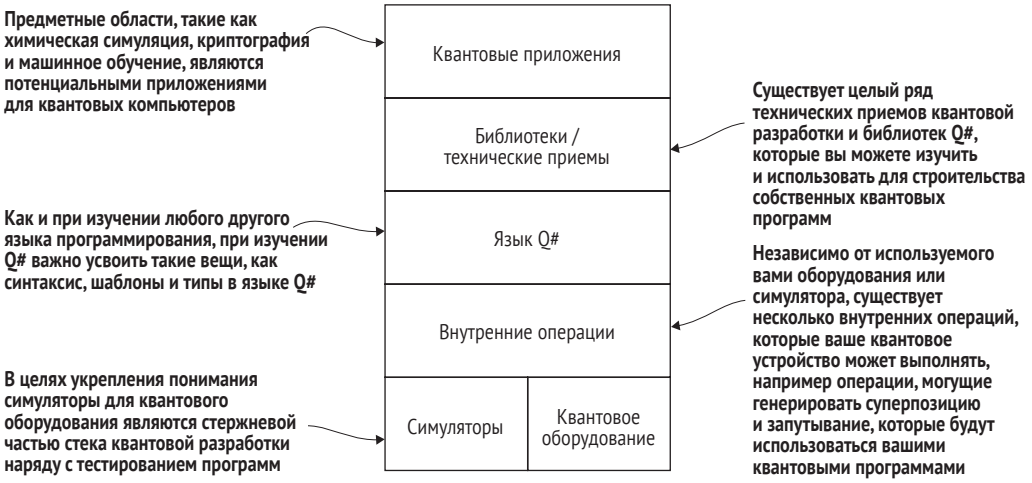


Рис. 1.6 В этой книге излагаются концепции, необходимые для написания квантовых программ. В части I мы начнем с описания симуляторов более низкого уровня и внутренних операций (например, аппаратного API), создав наш собственный симулятор на Python. В части II рассматриваются язык Q# и технические приемы квантовой разработки, которые помогут нам разрабатывать собственные приложения. В части III показано несколько известных приложений для квантовых вычислений, а также приведены проблемы и возможности, которые мы имеем в связи с развитием этой технологии

2 Кубиты: строительные блоки

Эта глава охватывает следующие ниже темы:

- почему случайные числа – это важный ресурс;
- что такое кубит;
- какие базовые операции можно выполнять на кубите;
- программирование квантового генератора случайных чисел на языке Python.

В этой главе мы начнем знакомиться с несколькими концепциями квантового программирования. Главная концепция, которую мы рассмотрим, – это *кубит*, квантовый аналог классического бита. Мы будем использовать кубиты в качестве абстракции или модели для описания новых видов вычислений, которые возможны с помощью квантовой физики. На рис. 2.1 показана модель применения квантового компьютера, а также конфигурация симулятора, который мы используем в этой книге. Реальные либо симулированные кубиты будут обитать на целевой машине и взаимодействовать с квантовыми программами, которые мы будем писать! Эти квантовые программы могут отправляться различными главными программами (или хост-программами), которые затем ожидают получения результатов из квантовой программы.

Для того чтобы понять, что такое кубиты и как мы с ними взаимодействуем, мы приведем пример их использования в наши дни: генерирование случайных чисел. Хотя из кубитов можно создавать гораздо более интересные устройства, простой пример *квантового генератора случай-*

ных чисел (*quantum random number generator*, аббр. *QRNG*) предлагает хороший способ познакомиться с кубитом.

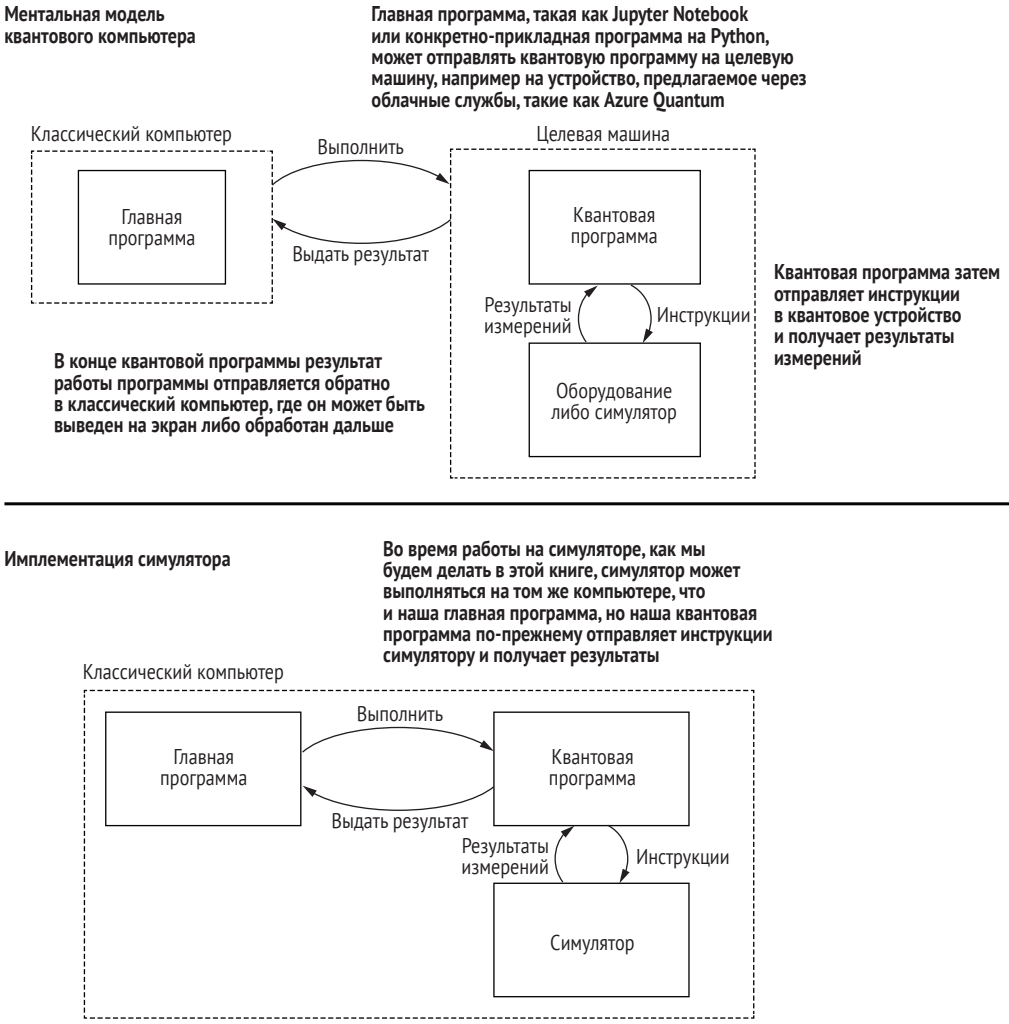


Рис. 2.1 Ментальная модель возможного использования квантового компьютера. Верхняя половина рисунка является общей моделью квантового компьютера. В этой книге мы будем использовать локальные симуляторы. Нижняя половина показывает то, что мы будем строить и использовать

2.1 Зачем нужны случайные числа?

Люди любят определенность. Нам нравится, когда нажатие клавиши на клавиатуре всегда делает одно и то же. Однако в некоторых контекстах мы хотим задействовать случайность:

- играя в игры;
- симулируя сложные системы (такие как фондовый рынок);
- подбирая безопасные секреты (например, пароли и криптографические ключи).

Во всех подобных ситуациях, когда нам нужна случайность, можно описывать шансы каждого исхода. Говоря о случайных событиях, описание шансов – это все, что мы можем сказать о ситуации до тех пор, пока не будет брошен кубик (либо не будет подброшена монета, либо пароль не будет использован повторно). Когда мы описываем шансы каждого примера, мы говорим примерно следующее:

- *если* я брошу этот кубик, *то* получу шестерку *с вероятностью* 1 из 6;
- *если* я подброшу эту монету, *то* получу орла *с вероятностью* 1 из 2.

Мы также можем описывать случаи, когда для каждого исхода вероятности не одинаковы. На «Колесе фортуны» (рис. 2.2) вероятность того, что после вращения колеса мы получим приз в размере 1 000 000 долларов, намного меньше, чем вероятность того, что после его вращения мы обанкротимся.

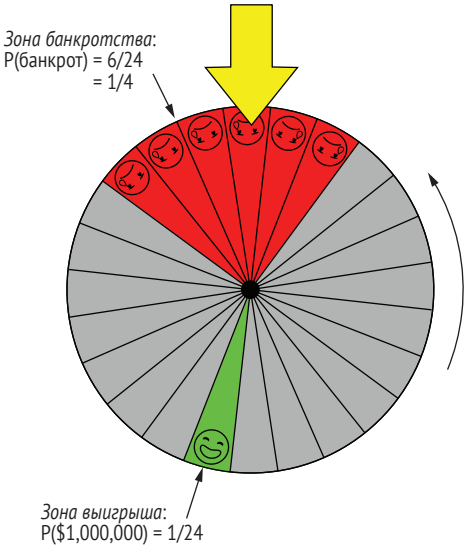


Рис. 2.2 Вероятности 1 000 000 долларов и банкротства на «Колесе фортуны». Прежде чем вращать колесо, мы не знаем точно, где оно остановится, но, глядя на колесо, мы знаем, что вероятность обанкротиться намного больше, чем вероятность крупного выигрыша

Как и в игровых шоу, в информатике существует масса контекстов, где случайность имеет решающее значение, в особенности когда требуется безопасность. Если мы хотим хранить некоторую информацию приватной, то криптография позволяет нам это делать, комбинируя наши данные со случайными числами различными способами. Если наш генератор случайных чисел не очень хорош – т. е. если злоумышленник может предсказывать числа, которые мы используем для защиты наших приватных данных, – тогда криптография нам поможет мало. Мы также можем представить себе использование плохого генератора случайных чисел для проведения розыгрыша или лотереи; злоумышленник, кото-

рый выяснит принцип генерирования наших случайных чисел, может привести нас прямо в банк.

Каковы шансы?

Если использовать случайные числа, которые наши злоумышленники умеют предсказывать, то можно потерять много денег. Спросите у продюсеров популярного игрового шоу 1980-х годов «Нажмите на удачу!» (*Press Your Luck*!).

Один из его участников обнаружил, что он может предсказывать место остановки нового электронного «колеса» игры, и это позволило ему выиграть более 250 000 долларов в сегодняшних деньгах. Об этом можно узнать больше, прочитав статью «Человек, у которого не было неудач» (*The Man Who Got No Whammies*) Закари Крокетта по адресу https://priceonomics.com/the-man-who-got-no-whammies.

Глубокое погружение: вычислительная безопасность и теоретико-информационная безопасность

Некоторые способы защиты приватной информации основаны на допущениях о том, какие задачи легко или трудно решать злоумышленнику. Например, широко используемый алгоритм шифрования под названием RSA основан на трудности поиска простых факторов для крупных чисел. Алгоритм RSA используется в интернете и в других контекстах для защиты пользовательских данных, исходя из допущения, что злоумышленники не способны легко факторизовывать очень крупные числа. До сих пор это допущение доказывало свою действенность, но вполне возможно, что будет обнаружен новый алгоритм факторизации, подрывающий безопасность RSA. Как мы увидим в главе 11, квантовый алгоритм под названием *алгоритм Шора* позволяет решать некоторые виды криптографических задач намного быстрее, чем это могут делать классические компьютеры, бросая вызов допущениям, широко используемым для обеспечения вычислительной безопасности.

Напротив, если злоумышленник способен угадывать секреты только случайно, даже с очень большими вычислительными мощностями, то система безопасности обеспечивает гораздо более высокие гарантии своей способности защищать приватную информацию. Такие системы считаются *информационно безопасными*. Позже в этой главе мы увидим, что генерирование случайных чисел в труднопредсказываемом стиле позволяет нам имплементировать информационно безопасную процедуру, именуемую одноразовым шифровальным блокнотом (либо просто *одноразовым блокнотом*, от англ. *one-time pad*).

Как оказалось, квантовая механика позволяет нам строить действительно уникальные источники информации. Если мы построим их правильно, то случайность наших результатов будет гарантирована *физикой*, а не допущением о том, сколько времени потребуется компьютеру для решения сложной задачи. Это означает, что хакеру или злоумышленнику

придется нарушить законы физики, чтобы нарушить безопасность! Это не означает, что мы должны использовать квантовые случайные числа для всего; люди по-прежнему являются самым слабым звеном в инфраструктуре безопасности 😈.

Это дает нам некоторую уверенность в том, что мы можем использовать квантовые случайные числа для жизненно важных задач, таких как защита приватных данных, проведение лотерей и игра в *Dungeons & Dragons* («Подземелья и драконы»). Симулирование работы квантовых генераторов случайных чисел позволит нам усвоить многие базовые концепции, лежащие в основе квантовой механики, поэтому давайте сразу приступим к работе!

Как упоминалось ранее, один из отличных способов начать работу состоит в том, чтобы посмотреть на пример квантовой программы, генерирующей случайные числа: квантовый генератор случайных чисел (QRNG). Не волнуйтесь, если следующий ниже алгоритм (также показанный на рис. 2.3) прямо сейчас не вполне для вас понятен – мы будем объяснять различные его части по мере прохождения остальной части главы.

1. Попросить квантовое устройство выделить кубит.
2. Применить к кубиту инструкцию, именуемую *инструкцией Адамара*; мы узна́ем о ней позже в этой главе.
3. Измерить кубит и вернуть результат.

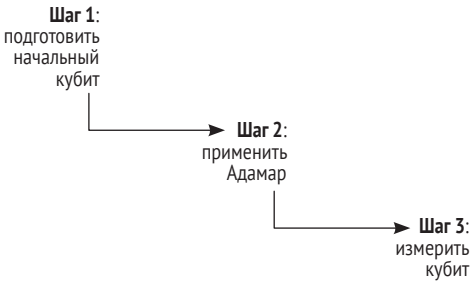


Рис. 2.3 Алгоритм квантового генератора случайных чисел. В целях отбора случайных чисел с помощью квантового компьютера наша программа подготовит свежий кубит, а затем применит инструкцию Адамара, чтобы подготовить нужную нам суперпозицию. В конце мы можем измерить и вернуть случайный результат

В остальной части главы мы разработаем класс Python под названием `QuantumDevice` (Квантовое устройство), который позволит нам писать программы, имплементирующие алгоритмы, подобные этому. После того как у нас появится класс `QuantumDevice`, мы сможем написать QRNG-генератор в виде программы на языке Python, аналогичной классическим программам, к которым мы привыкли.

ПРИМЕЧАНИЕ Инструкции по настройке среды Python на вашем устройстве для выполнения квантовых программ см. в приложении А к книге.

Обратите внимание, что следующий ниже пример не будет работать до тех пор, пока вы не напишете симулятор в этой главе 😉.

Листинг 2.1 qrng.py: квантовая программа, генерирующая случайные числа

```
def qrng(device : QuantumDevice) -> bool:      (1)
    with device.using_qubit() as q:            (2)
        q.h()                                  (3)
        return q.measure()                     (4)
```

(1) Квантовые программы пишутся точно так же, как и классические программы. В данном случае мы используем язык Python, поэтому наша квантовая программа – это функция qrng на Python, которая имплементирует QRNG-генератор.

(2) Квантовые программы работают, запрашивая у квантово-вычислительного оборудования кубиты – квантовые аналоги битов, которые мы можем использовать для выполнения вычислений.

(3) В целях получения данных из наших кубитов мы их измеряем. Здесь в половине случаев наша операция измерения будет возвращать истину, а в другой половине – ложь.

(4) Получив кубит, мы можем выдавать этому кубиту инструкции. Как и в ассемблерных языках, эти инструкции часто обозначаются короткими аббревиатурами; мы увидим смысл функции h() позже в этой главе.

Вот так! Четыре шага, и мы только что создали нашу первую квантовую программу. Приведенный выше QRNG-генератор возвращает истину либо ложь. В терминах языка Python это означает, что мы получаем 1 или 0 всякий раз, когда выполняем `qrng`. Это не очень сложный генератор случайных чисел, но число, которое он возвращает, и в самом деле является случайным.

В целях запуска программы `qrng` нам нужно предоставить функции квантовое устройство, `QuantumDevice`, которое обеспечивает доступ к кубитам и имплементирует различные инструкции, которые мы можем отправлять кубитам. Хотя нам нужен только один кубит, для начала мы создадим собственный симулятор квантового компьютера. Для этой скромной задачи может использоваться существующее оборудование, но то, что мы рассмотрим позже, выйдет за рамки доступного оборудования. Оно будет работать локально на ноутбуке или настольном компьютере и вести себя как настоящее квантовое оборудование. В оставшейся части главы мы построим разные части, необходимые для написания нашего собственного симулятора и выполнения программы `qrng`.

2.2 Что такое классические биты?

При изучении концепций квантовой механики часто бывает полезно сделать шаг назад и пересмотреть *классические* концепции, чтобы провести параллель с тем, как они выражаются в квантовых вычислениях. Имея это в виду, давайте еще раз взглянем на то, что такое *биты*.

Предположим, что мы хотели бы отправить нашей дорогой подруге Еве важное сообщение, например «💔». Каким образом представить наше сообщение так, чтобы его можно было легко отправлять?

Мы могли бы начать с составления списка всех букв и символов, которые мы могли бы использовать для написания сообщений. К счастью, Консорциум Юникода (https://unicode.org) уже сделал это за нас и назначил *коды* широкому спектру символов, используемых для связи по всему миру. Например, символу *I* назначен код 0049, тогда как 🏵 обозначается как A66E, 🛎 как 2E0E и 💔 как 1F496. На первый взгляд эти коды могут показаться бесполезными, но они являются полезными рецептами для отправки каждого символа в виде сообщения. Если мы знаем, как отправлять два сообщения (назовем их «0» и «1»), то эти рецепты позволяют нам создавать более сложные сообщения, такие как «🏵», «🛎» и «💔», как последовательности сообщений «0» и «1»:

0	0000	**8**	1000
1	0001	**9**	1001
2	0010	**A**	1010
3	0011	**B**	1011
4	0100	**C**	1100
5	0101	**D**	1101
6	0110	**E**	1110
7	0111	**F**	1111

Теперь мы можем отправлять все, что захотим, если знаем, как отправлять Еве всего два сообщения: сообщение «0» и сообщение «1». Используя эти рецепты, наше сообщение «💔» становится «0001 1111 0100 1001 0110», или в Юникоде 1F496.

ДЛЯ СПРАВКИ Не отправьте по ошибке «0001 1111 0100 1001 0100», иначе Ева получит от вас письмо 💔!

Каждое из двух сообщений «0» и «1» называется *битом*.

ПРИМЕЧАНИЕ Для того чтобы отличить биты от квантовых битов, которые мы будем встречать в остальной части книги, мы часто будем подчеркивать, что говорим о *классических битах*.

Когда мы используем слово «бит», то обычно имеем в виду одну из двух вещей:

- любую физическую систему, которая может быть полностью описана, отвечая на один двоичный вопрос «истина–ложь»;
- информацию, хранящуюся в такой физической системе.

Например, навесные замки, выключатели света, транзисторы, левый либо правый спин на криволинейном шаре и вино в бокалах можно рассматривать как биты, поскольку мы можем использовать их все для отправки или регистрации сообщений (см. табл. 2.1).

Таблица 2.1 Примеры битов

Метка	Замок	Выключатель	Транзистор	Бокал вина	Шар
0	Открыт	Выкл.	Низкое напряжение	Содержит белое вино	Поворачивается влево
1	Закрыт	Вкл.	Высокое напряжение	Содержит красное вино	Поворачивается вправо

Все эти примеры являются битами, потому что мы можем полностью их описать кому-то другому, отвечая на один вопрос «истина–ложь». Иными словами, каждый пример позволяет нам отправлять либо сообщение 0, либо сообщение 1. Как и все концептуальные модели, бит имеет ограничения – как бы мы описали розовое вино, например?

Даже с учетом этого бит является полезным инструментом, потому что мы можем описывать способы взаимодействия с битом, которые не зависят от того, как мы на самом деле строим бит.

2.2.1 Что можно делать с классическими битами?

Теперь, когда у нас есть способ описания и передачи классической информации, что мы можем делать, чтобы ее обрабатывать и модифицировать? Мы описываем способы обработки информации в терминах *операций*, которые определяем как способы описания того, как модель может быть изменена или использована в работе.

В целях визуализации операции NOT давайте представим, что две точки помечены как 0 и 1, как показано на рис. 2.4. Операция NOT – это любое преобразование, которое превращает биты 0 в биты 1 и наоборот.

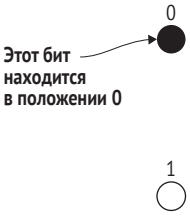


Рис. 2.4 Классический бит может находиться в одном из двух разных состояний, обычно именуемых 0 и 1. Мы можем изобразить классический бит в виде черной точки в положении 0 либо в положении 1

В классических запоминающих устройствах, таких как жесткие диски, вентиль NOT переворачивает магнитное поле, в котором хранится наше битовое значение. Как показано на рис. 2.5, мы можем рассматривать NOT как имплементацию поворота на 180° между точками 0 и 1, которые мы нарисовали на рис. 2.4.

Визуализация классических битов в таком ключе также позволяет нам немного расширить понятие битов, чтобы включить способ описания *случайных битов* (который будет полезен позже). Если у нас есть *справедливая монета* (т. е. монета, которая в половине случаев приземляется орлами, а в другой половине – решками), то было бы неправильно называть эту монету 0 или 1. Мы будем знать, какое битовое значение имеет бит нашей монеты, только если положим ее той или иной стороной вверх; мы также можем ее подбрасывать, чтобы получать значение случайного бита. Всякий раз, когда мы подбрасываем монету, мы знаем, что в конце

концов она приземлится, и мы получим орлы либо решки. То, как она приземляется – орлами либо решками, – зависит от вероятности, именуемой смещением (или искаженностью) монеты. Для описания смещения мы должны выбрать сторону монеты, что легко сформулировать как вопрос типа «какова вероятность того, что монета будет приземляться орлами?». Так или иначе, справедливая монета имеет смещение, равное 50 %, потому что она приземляется орлами в половине случаев, что соответствует битовому значению 0 на рис. 2.6.

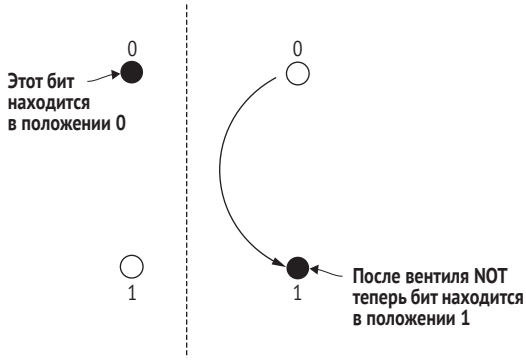


Рис. 2.5 Классическая операция NOT переворачивает классический бит между 0 и 1. Например, если бит начинается в состоянии 0, NOT переворачивает его в состояние 1

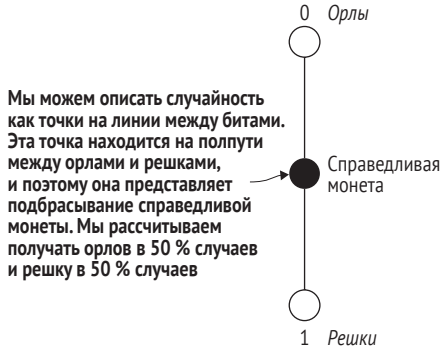


Рис. 2.6 Мы можем использовать тот же рисунок, что и раньше, чтобы расширить нашу концепцию бита для описания монеты. В отличие от бита, монета имеет вероятность быть либо 0, либо 1 всякий раз, когда она подбрасывается. Мы графически представляем эту вероятность в виде точки между 0 и 1

Используя эту визуализацию, мы можем взять наши предыдущие две точки, обозначающие битовые значения 0 и 1, и соединить их линией, на которой мы можем построить смещение нашей монеты. При этом становится проще увидеть, что операция NOT (которая по-прежнему работает с нашим новым вероятностным битом) ничего со справедливой монетой не делает. Если 0 и 1 появляются с одинаковой вероятностью, то не имеет значения, поворачиваем ли мы 0 в 1 либо 1 в 0: мы все равно получим 0 и 1 с одинаковой вероятностью.

Что делать, если наше смещение не находится посередине? Если мы знаем, что кто-то пытается сжульничать, используя взвешенную или мо-

дифицированную монету, которая почти всегда приземляется орлами, то мы можем сказать, что смещение монеты составляет 90 %, и построить ее на нашей линии, нарисовав точку гораздо ближе к 0, чем к 1.

ОПРЕДЕЛЕНИЕ Точка на линии, где мы рисуем каждый классический бит, есть *состояние* этого бита.

Давайте рассмотрим сценарий. Скажем, я хочу отправить вам кучу битов, хранящихся с использованием висячих замков. Какой самый дешевый способ это сделать?

Один из подходов состоит в том, чтобы отправить по почте коробку, содержащую большое число замков, которые либо открыты, либо закрыты, и надеяться, что они прибудут в том же состоянии, в котором я их отправил. С другой стороны, мы можем договориться, что все замки начинаются в состоянии 0 (открыт), и я могу отправить вам инструкции о том, какие замки закрывать. Таким образом, вы можете купить свои собственные замки, и мне нужно только отправить *описание* того, как подготовить эти замки, используя классические вентили NOT. Отправить листок бумаги или электронное письмо намного дешевле, чем отправлять коробку с висячими замками!

Это служит иллюстрацией принципа, на который мы будем опираться на протяжении всей книги: *состояние физической системы также может описываться в терминах инструкций по подготовке этого состояния*. Следовательно, разрешенные в физической системе операции определяют и то, какие состояния возможны.

Есть еще одна вещь, которую мы можем делать с классическими битами, хотя она и может показаться совершенно тривиальной, и она будет критически важной для понимания квантовых вычислений: мы можем на них смотреть. Если я посмотрю на замок и заключу: «Ага! Этот замок не заперт», – то теперь я могу думать о своем мозге как об особенно хлюпком бите. Сообщение 0 сохраняется в моем мозгу при мысли «Ага! Этот замок не заперт», тогда как сообщение 1 будет сохранено при мысли «Так, хорошо, этот замок заперт». По сути, глядя на классический бит, я *скопировал* его в свой мозг. Мы говорим, что акт *измерения* классического бита копирует этот бит[1].

В более общем случае современная жизнь строится вокруг легкости, с которой мы копируем классические биты, глядя на них. Мы копируем классические биты с поистине безрассудным самозабвением, измеряя многие миллиарды классических битов каждую секунду, когда мы копируем данные с наших игровых компьютеров на наши телевизоры.

С другой стороны, если бит хранится в виде монеты, то операция измерения включает в себя его переворачивание. Измерение не совсем ко-

---

[1] Измерение (measurement) в квантовой механике – это концепция, описывающая возможность получения информации о состоянии системы путем проведения физического эксперимента. Подробнее см. https://en.wikipedia.org/wiki/Measurement_in_quantum_mechanics. – *Прим. перев.*

пирует монету, так как при следующем подбрасывании я могу получить другой результат измерения. Если у меня есть только один результат измерения монеты, то я не могу сделать вывод о вероятности получить орлов или решек. У нас не было этой двусмысленности с замковыми битами, потому что мы знали, что состояние замков равно либо 0, либо 1. Если бы я измерил замок и обнаружил, что он находится в состоянии 0, то я бы знал, что он всегда будет находиться в состоянии 0, если только я что-то не сделаю с замком.

В квантовых вычислениях, как мы увидим позже в этой главе, ситуация не совсем такая же. Хотя измерение классической информации обходится достаточно дешево, чтобы жаловаться на точное количество миллиардов бит, которое нам позволяет измерять кабель стоимостью 5 долларов, мы должны быть гораздо более осторожны с тем, как мы подходим к квантовым измерениям.

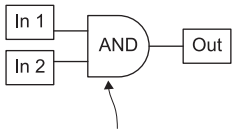
2.2.2 Абстракции – наши друзья

Независимо от того, как мы строим бит физически, мы (к счастью) можем представлять их одинаково как в математике, так и в исходном коде. Например, Python предоставляет тип `bool` (сокращение от Boolean, в честь логика Джорджа Буля), который имеет два валидных значения: `True` и `False`. Мы можем представлять преобразования на битах, такие как NOT и OR, как операции, воздействующие на булевы переменные. Важно отметить, что мы можем указывать классическую операцию путем описания того, как эта операция преобразовывает каждое возможное входное значение, и это описание часто именуется *таблицей истинности*.

ОПРЕДЕЛЕНИЕ *Таблица истинности* – это таблица, описывающая результат классической операции на выходе для каждой возможной комбинации данных на входе. Например, на рис. 2.7 показана таблица истинности для операции AND.

Таблицы истинности – это один из способов показать, что происходит с классическими битами в функциях или логических схемах

Вход	Выход
0 0	0
0 1	0
1 0	0
1 1	1



Этот логический вентиль генерирует те же выходные биты, что и таблица истинности слева

Рис. 2.7 Таблица истинности для логической операции AND. Если мы знаем всю таблицу истинности для логической операции, то мы знаем, что конкретно эта операция делает для любого возможного значения на входе

Мы можем найти таблицу истинности для операции NAND (сокращение от NOT-AND) на Python, перебрав комбинации `True` и `False`.

Листинг 2.2 Использование Python для печати таблицы истинности для NAND

```
>>> from itertools import product
>>> for inputs in product([False, True], repeat=2):
...     output = not (inputs[0] and inputs[1])
...     print(f"{inputs[0]}\t{inputs[1]}\t->\t{output}")
False   False   ->      True
False   True    ->      True
True    False   ->      True
True    True    ->      False
```

ПРИМЕЧАНИЕ Описание операции как таблицы истинности соблюдается и для более сложных операций. В принципе, даже такая операция, как сложение между двумя 64-битовыми целыми числами, может быть записана в виде таблицы истинности. Однако это не очень практично, так как таблица истинности для двух 64-битовых входных значений будет иметь $2^{128} \approx \times 10^{38}$ элементов и для записи потребует 10^{40} бит. Для сравнения, по последним оценкам, размер всего интернета приближается к 10^{27} бит.

Бóльшая часть искусства в классической логике и дизайне оборудования заключается в изготовлении *схем*, которые могут обеспечивать очень компактные представления классических операций, не полагаясь на потенциально массивные таблицы истинности. В квантовых вычислениях мы используем название «*унитарные операторы*» для аналогичных таблиц истинности, служащих для квантовых битов, на которых мы будем подробно останавливаться по мере продвижения.

Подытоживая:

- классические биты – это физические системы, которые могут находиться в одном из двух разных *состояний*;
- классическими битами можно манипулировать с помощью *операций*, чтобы обрабатывать информацию;
- акт *измерения* классического бита создает копию информации, содержащейся в состоянии.

ПРИМЕЧАНИЕ В следующем далее разделе мы будем использовать линейную алгебру, чтобы узнать о *кубитах*, базовой единице информации в квантовом компьютере. Если вам нужно освежить свои знания по линейной алгебре, то сейчас самое время, чтобы сделать крюк к приложению С книги. Мы будем ссылаться на аналогию из этого приложения на протяжении всей книги в местах, где будем думать о векторах как о направлениях на карте местности. Когда вы вернетесь назад, мы будем находиться тут!

2.3 Кубиты: состояния и операции

Подобно тому, как классические биты являются самой базовой единицей информации в классическом компьютере, *кубиты* являются базовой единицей информации в квантовом компьютере. Они могут быть физически имплементированы системами, имеющими два состояния, точно так же, как и классические биты, но они ведут себя в соответствии с законами квантовой механики, что допускает некоторые виды поведения, на которые классические биты не способны. Давайте отнесемся к кубитам так же, как к любой другой забавной новой компьютерной детали: подключим ее и посмотрим, что произойдет!

Симулированные кубиты

Во всей книге мы не будем использовать настоящие кубиты. Вместо этого мы будем использовать *классические симуляции* кубитов. Это позволит нам узнать принцип работы квантовых компьютеров и начать программировать небольшие экземпляры задач, которые квантовые компьютеры могут решать, даже если у нас еще нет доступа к квантовому оборудованию, необходимому для решения практических задач.

Проблема с этим подходом заключается в том, что для симулирования кубитов на классических компьютерах требуется экспоненциальный объем классических ресурсов в числе кубитов. Самые мощные классические вычислительные службы могут симулировать примерно до 40 кубит, после чего придется упрощать или сокращать типы выполняемых квантовых программ. Для сравнения, на момент написания этой книги текущее коммерческое оборудование достигало примерно 70 кубит. Устройства с таким числом кубитов чрезвычайно трудно симулировать с помощью классических компьютеров, вместе с тем доступные в настоящее время устройства все еще слишком шумны, чтобы иметь возможность выполнять большинство полезных вычислительных задач.

Представьте себе, что вам нужно написать классическую программу, работая лишь с 40 классическими битами! Хотя 40 бит – это довольно мало по сравнению с гигабайтами, с которыми мы привыкли работать в классическом программировании, тем не менее всего с 40 кубитами можно делать некоторые действительно интересные вещи, которые помогут нам строить прототипы того, как может выглядеть реальное квантовое преимущество.

2.3.1 Состояние кубита

В целях имплементирования нашего QRNG-генератора нам нужно придумать, как описать наш кубит. Мы использовали замки, мячи и другие классические системы для представления значений *классических битов*, 0 либо 1. Мы можем использовать многие физические системы, которые будут выступать в качестве кубита, при этом *состояния* будут «значениями», которые наш кубит может иметь.

Подобно состояниям 0 и 1 классических битов, мы можем написать метки для квантовых состояний. Состояния кубита, наиболее похожие на классические 0 и 1, обозначаются как |0⟩ и |1⟩, как показано на рис. 2.8. Они носят названия соответственно *кет 0* и *кет 1*.

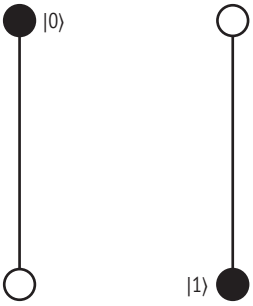


Рис. 2.8 Используя (бра-кетную) нотацию Дирака для кубитов, мы можем графически представить состояния |0⟩ и |1⟩ кубитов так же, как представляли состояния 0 и 1 классических битов. В частности, мы будем рисовать состояния |0⟩ и |1⟩ как противоположные точки вдоль оси, как показано выше

Кет?

Термин «кет» происходит от одного причудливого названия в квантовых вычислениях, которое обязано своей историей чрезвычайно глупому каламбуру. Как мы увидим, когда посмотрим на измерения, есть еще один вид объекта, именуемый «бра», который записывается как ⟨0|. Когда мы соединяем бра и кет, мы получаем пару скобок: ⟨⟩.

Использование бра и кет при написании математики для квантовой механики часто называют *нотацией Дирака* в честь Поля Дирака, который изобрел как саму нотацию, так и вызывающий вздох сожаления каламбур, с которым мы теперь вынуждены иметь дело. Мы будем встречать этот причудливый стиль на протяжении всей книги.

Однако следует помнить, что состояние – это удобная модель, используемая для предсказания поведения кубита, а не иманентное свойство кубита. Это различие становится особенно важным, когда мы будем рассматривать квантовое измерение позже в этой главе – как мы увидим, мы не можем измерять состояние кубита непосредственно.

ПРЕДУПРЕЖДЕНИЕ В реальных системах мы никогда не сможем извлечь или в совершенстве узнать состояние кубита, имея конечное число копий.

Не волнуйтесь, если все это еще не вполне для вас понятно; мы увидим массу примеров, когда будем читать книгу. Сейчас важно иметь в виду, что кубиты – это не состояния.

Если мы хотим просимулировать движение бейсбольного мяча после броска, то мы могли бы начать с регистрации его текущего местоположения, скорости и направления его движения, стороны вращения и т. д. Этот список чисел помогает нам представлять бейсбольный мяч на листе бумаги либо в компьютере, чтобы предсказывать, что конкретно этот

бейсбольный мяч будет делать, но мы бы не сказали, что бейсбольный мяч – это список чисел. Приступая к симулированию, нам нужно взять интересующий нас бейсбольный мяч и *измерить*, где он находится, как быстро он движется и т. д.

Мы говорим, что полный набор данных, необходимый для точного симулирования поведения бейсбольного мяча, заключен в *состоянии* этого бейсбольного мяча. Аналогичным образом, состояние кубита – это весь набор данных, необходимый для его симулирования и предсказания результатов, которые мы получим при его измерении. Подобно тому, как мы нуждаемся в обновлении состояния летящего бейсбольного мяча в нашем симуляторе, мы будем обновлять состояние кубита, когда применяем к нему операции.

> **ДЛЯ СПРАВКИ** Запомнить это тонкое различие помогает следующее утверждение: кубит *описывается* состоянием, но неверно, что кубит *является* состоянием.

Все становится немного утонченнее, когда речь заходит об измерении. Хотя мы можем измерить бейсбольный мяч, не делая с ним ничего, кроме копирования некоторой классической информации, как мы увидим в остальной части книги, мы не можем идеально скопировать квантовую информацию, хранящуюся в кубите, – когда мы измеряем кубит, мы оказываем влияние на его состояние. Это иногда сбивает с толку, так как мы регистрируем полное состояние кубита, когда его симулируем, вследствие чего мы можем заглядывать в память нашего симулятора всякий раз, когда захотим. Ничто из того, что мы можем делать с реальными кубитами, не позволяет нам смотреть на их состояние, поэтому если мы будем «жульничать», заглядывая в память симулятора, то не сможем выполнить нашу программу на реальном оборудовании.

Иными словами, хотя, глядя на состояния, мы помогаем себе в отладке классических симуляторов при их строительстве, мы должны обеспечивать написание алгоритмов только на основе информации, которую мы могли бы правдоподобно узнавать из реального оборудования.

Жульничество с закрытыми глазами

При использовании квантового симулятора он должен хранить состояние наших кубитов внутри – вот почему симулирование квантовых систем бывает таким сложным. Каждый кубит в принципе может коррелироваться с любым другим кубитом, поэтому нам нужны экспоненциальные ресурсы в общем случае, чтобы записывать состояние в симуляторе (подробнее об этом мы поговорим в главе 4).

Если мы «жульничаем», глядя непосредственно на сохраненное симулятором состояние, то мы можем выполнять нашу программу только на симуляторе, а не на реальном оборудовании. В последующих главах мы увидим, как можно жульничать безопаснее, используя инструкции подтверждения истинности `assert` и делая обман ненаблюдаемым 😈.

2.3.2 Игра в операции

Теперь, когда для этих состояний у нас есть названия, давайте покажем, как представлять информацию, которую они содержат. С помощью классических битов мы можем регистрировать содержащуюся в бите информацию в любое время в виде простого значения в битовой строке: 0 или 1. Это работает, потому что единственные операции, которые мы можем выполнять, состоят из переворотов (или поворотов на 180°) на этой линии. Квантовая механика позволяет нам применять к кубитам больше видов операций, включая поворот менее чем на 180°. То есть кубиты отличаются от классических битов видами операций, которые мы можем с ними выполнять.

ПРИМЕЧАНИЕ В то время как операции с классическими битами являются логическими операциями, которые могут выполняться путем сочетания операций NOT, AND и OR различными способами, квантовые операции состоят из *поворотов*.

Например, если мы хотим перевести состояние кубита из |0⟩ в |1⟩ и наоборот, т. е. применить квантовый аналог операции NOT, то мы поворачиваем кубит по часовой стрелке на 180°, как показано на рис. 2.9.

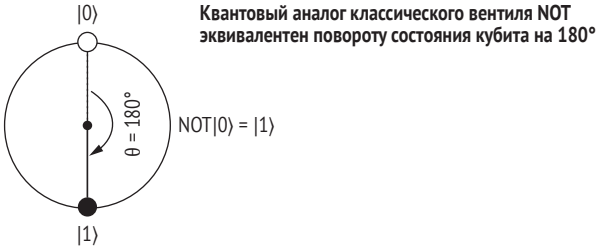


Рис. 2.9 Визуализация квантового эквивалента операции NOT, оперирующей на кубите в состоянии |0⟩, оставляя кубит в состоянии |1⟩. Мы можем рассматривать эту операцию как поворот на 180° вокруг центра линии, соединяющей состояния |0⟩ и |1⟩

Мы видели, как поворот на 180° является аналогом вентиля NOT. Какие еще повороты можно делать?

Обратимость

Когда мы поворачиваем квантовое состояние, мы всегда можем вернуться в то же состояние, в котором начали, повернув назад. Это свойство, именуемое *обратимостью*, оказывается для квантовых вычислений фундаментальным. За исключением измерения, о котором мы подробнее узнаем позже в этой главе, все квантовые операции должны быть обратимыми.

Однако не все классические операции, к которым мы привыкли, обратимы. Такие операции, как AND и OR, не являются обратимыми в том, как они обыч-

но записываются, поэтому они не могут быть имплементированы как квантовые операции без дополнительной работы. Мы увидим, как это сделать, в главе 8, когда введем трюк с «откатом вычисления» (uncompute) для выражения других классических операций в виде поворотов.

С другой стороны, классические операции, такие как XOR, можно легко сделать обратимыми, поэтому мы можем записывать их как повороты, используя квантовую операцию, именуемую *контролируемым-NOT*, как мы увидим в главе 8.

Если мы повернем кубит в состоянии |0⟩ по часовой стрелке на 90° вместо 180°, то получим квантовую операцию, которую можно рассматривать как квадратный корень из операции NOT, как показано на рис. 2.10.

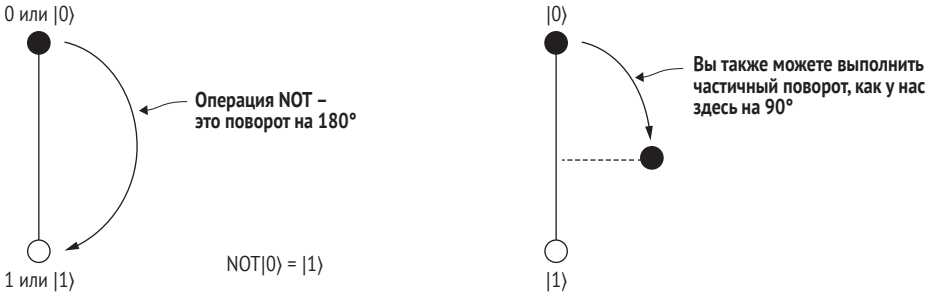


Рис. 2.10 Мы также можем повернуть состояние менее чем на 180 градусов. Поступая в таком ключе, мы получаем состояние, которое не является ни |0⟩, ни |1⟩, но находится на полпути по окружности между ними

Подобно тому, как мы ранее определили квадратный корень $\sqrt{x}$ из числа x как число y, такое, что $y^2 = x$, мы можем определить квадратный корень квантовой операции. Если мы дважды применим поворот на 90°, то вернемся к операции NOT, поэтому мы можем рассуждать о повороте на 90° как о квадратном корне из NOT.

Половины и полуотрицания

В каждой области есть свои камни преткновения. Например, спросите графического программиста, что означает положительное *y*: направление «вверх» или же направление «вниз». В квантовых вычислениях богатая история и междисциплинарный характер этой сферы иногда воспринимаются как обоюдоострый меч в том смысле, что каждый отдельный образ мышления о квантовых вычислениях связан с соглашениями и обозначениями.

Это проявляется, в частности, в легкости, с которой можно ошибиться в том, куда поместить факторы числа два. В этой книге мы решили следовать соглашению, используемому в языке Q# компании Microsoft.

Теперь у нас есть новое состояние, которое не является ни |0⟩, ни |1⟩, но представляет собой равную комбинацию их обоих. Точно в том же смысле, в каком мы можем описать «северо-восток», сложив направления «север» и «восток», мы можем записать это новое состояние, как показано на рис. 2.11.

Состояние кубита можно представить в виде точки на окружности, имеющей два помеченных состояния на полюсах: |0⟩ и |1⟩. В более общем случае мы будем изображать повороты с использованием произвольных углов θ между состояниями кубита, как показано на рис. 2.12.

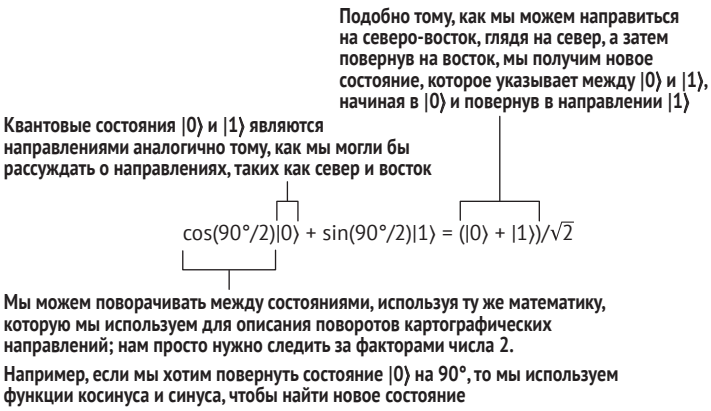


Рис. 2.11 Мы можем записать состояние, которое мы получаем при повороте на 90°, рассуждая о состояниях |0⟩ и |1⟩ как о направлениях. Для этого, используя немного тригонометрии, мы получаем, что поворачивание состояния |0⟩ на 90° дает нам новое состояние, (|0⟩ + |1⟩)/√2 . Более подробно о том, как записывать такого рода повороты математически, можно узнать из приложения B к книге, которое поможет освежить ваши знания по линейной алгебре

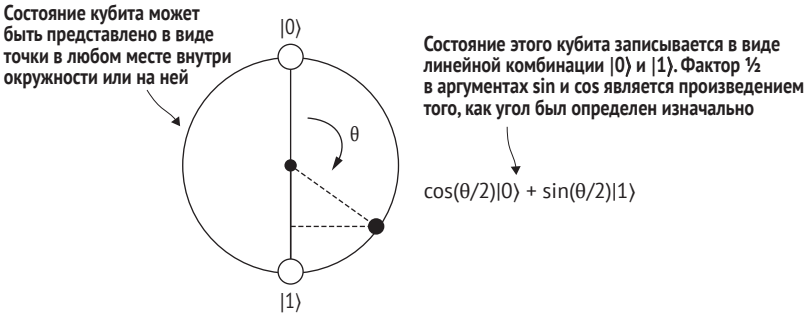


Рис. 2.12 Если мы повернем состояние |0⟩ на угол, отличный от 90° или 180°, то результирующее состояние можно представить в виде точки на окружности, которая имеет |0⟩ и |1⟩ в качестве верхнего и нижнего полюсов. Это дает нам возможность визуализировать возможные состояния, в которых может находиться один кубит

|+⟩, |−⟩ и суперпозиция

Мы называем состояние, представляющее собой равную комбинацию |0⟩ и |1⟩, состоянием $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ (из-за знака между членами). Мы говорим, что состояние |+⟩ является *суперпозицией* |0⟩ и |1⟩.

Если поворот равен −90° (против часовой стрелки), то мы называем полученное состояние $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$. Попробуйте записать эти повороты, используя −90°, чтобы увидеть, что мы получим |−⟩!

Математически мы можем записать состояние любой точки на окружности, представляющей наш кубит, как $\cos(\theta/2)\,|0\rangle + \sin(\theta/2)\,|1\rangle$, где |0⟩ и |1⟩ – это разные способы записи соответственно векторов [[1], [0]] и [[0], [1]].

ДЛЯ СПРАВКИ Думая о кетной нотации, следует помнить, что она дает *имена* векторам, которые мы широко используем. Когда мы пишем |0⟩ = [[1], [0]], мы говорим, что вектор [[1], [0]] имеет достаточную важность, чтобы мы назвали его в честь 0. Аналогичным образом, когда мы пишем $|+\rangle = [[1], [1]]/\sqrt{2}$, мы даем имя векторному представлению состояния, которое будем использовать на протяжении всей книги.

Это можно выразить и по-другому, сказав, что кубит, как правило, представляет собой *линейную комбинацию* векторов |0⟩ и |1⟩ с коэффициентами, описывающими угол, на который |0⟩ должен быть повернут, чтобы попасть в это состояние. В целях практической пригодности для программирования мы можем записывать то, как поворачивание состояния влияет на каждое из двух состояний |0⟩ и |1⟩, как показано на рис. 2.13.

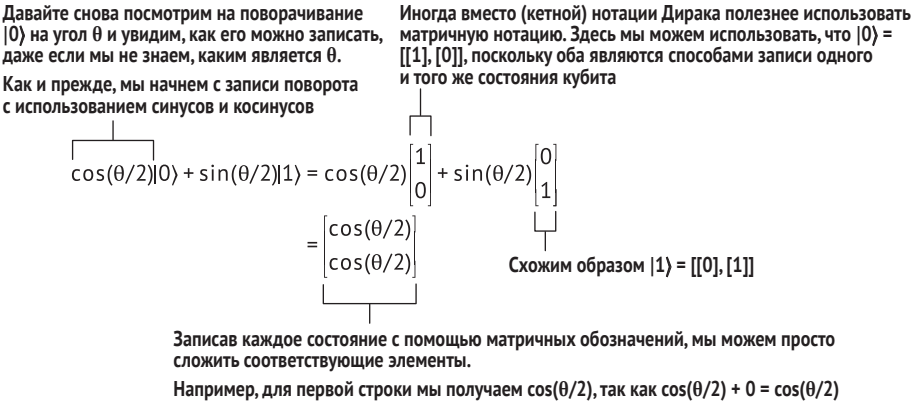


Рис. 2.13 Используя линейную алгебру, мы можем описать состояние одного кубита как двухэлементный вектор. В этом уравнении мы показываем, как этот образ мышления о состояниях кубитов связан с нашим ранее упомянутым использованием (бра-кетной) нотации Дирака. В частности, мы показываем окончательное состояние после поворачивания состояния |0⟩ на произвольный угол θ, используя как векторное, так и дираковское обозначения; оба они будут полезны в разных точках нашего квантового путешествия

ДЛЯ СПРАВКИ Это в точности соответствует тому, как мы ранее использовали базис векторов для представления линейной функции в виде матрицы (см. приложение С к книге).

В этой книге мы узнаем и о других квантовых операциях, но их легче всего визуализировать в виде поворотов. В табл. 2.2 обобщены состояния, которые мы научились создавать в результате этих поворотов.

Таблица 2.2 Метки состояний, разложения в нотации Дирака и представления в виде векторов

Метка состояния	Нотация Дирака	Векторное представление
$\|0\rangle$	$\|0\rangle$	[[1], [0]]
$\|1\rangle$	$\|1\rangle$	[[0], [1]]
$\|+\rangle$	$(\|0\rangle + \|1\rangle)/\sqrt{2}$	$[[1/\sqrt{2}], [1/\sqrt{2}]]$
$\|-\rangle$	$(\|0\rangle - \|1\rangle)/\sqrt{2}$	$[[1/\sqrt{2}], [-1/\sqrt{2}]]$

Полный рот математики

На первый взгляд, что-то вроде $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ ужасно трудно произносить вслух, делая эту нотацию бесполезной в разговоре. Однако на практике квантовые программисты часто прибегают к нескольким сокращениям, когда говорят вслух или рисуют что-то на доске.

Например, часть «$\sqrt{2}$» всегда должна быть там, так как векторы, представляющие квантовые состояния, всегда должны иметь длину один; это означает, что мы иногда можем пренебрегать правилами нотации и писать «$|+\rangle = |0\rangle + |1\rangle$», полагаясь на то, что наша аудитория не забудет разделить на $\sqrt{2}$. Если мы выступаем с докладом или обсуждаем квантовые вычисления за чашкой чая, то мы можем сказать: «кет плюс – это кет 0 плюс кет 1», но повторное использование слова «плюс» немного сбивает с толку без помощи бра и кет. Для того чтобы подчеркнуть в устной форме, что сложение позволяет нам представлять суперпозицию, мы вполне можем вместо этого сказать, что «плюсовое состояние – это равная суперпозиция нуля и единицы».

2.3.3 Измерение кубитов

Когда мы хотим получить информацию, хранящуюся в кубите, нам нужно измерить кубит. В идеале мы хотели бы иметь измерительное устройство, которое позволяет нам напрямую считывать всю информацию о состоянии сразу. Как оказалось, это невозможно по законам квантовой механики, как мы увидим в главах 3 и 4. Тем не менее измерение *позволяет* нам узнавать информацию о состоянии относительно конкретных направлений в системе. Например, если у нас кубит находится в состоянии $|0\rangle$ и мы смотрим, не находится ли он в состоянии $|0\rangle$, то мы всегда в результате получим, что это так. С другой стороны, если у нас кубит находится в состоянии $|+\rangle$ и мы смотрим, не находится ли он в состоянии $|0\rangle$, то мы получим результат 0 с вероятностью 50 %. Как показано

на рис. 2.14, это происходит потому, что состояние |+⟩ одинаково перекрывается с состояниями |0⟩ и |1⟩, вследствие чего мы будем получать оба результата с одинаковой вероятностью.

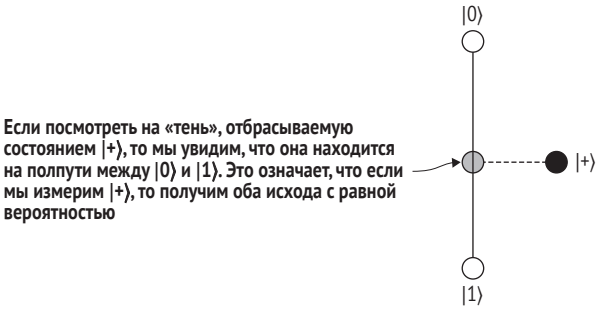


Рис. 2.14 Состояние |+⟩ одинаково перекрывается как с |0⟩, так и с |1⟩, потому что «тень», которую оно отбрасывает, находится точно посередине. Отсюда, когда мы смотрим на наш кубит, чтобы увидеть, не находится ли он в состоянии |0⟩ или |1⟩, мы получим оба результата с равной вероятностью, если наш кубит начал в состоянии |+⟩. Мы можем думать о тени, которую состояние |+⟩ отбрасывает на линию между состояниями |0⟩ и |1⟩, как о своего рода монете

ДЛЯ СПРАВКИ Результаты измерения кубитов *всегда* являются классическими битовыми значениями! Иными словами, независимо от того, что мы измеряем, классический бит либо кубит, наш результат всегда будет классическим битом.

В большинстве случаев мы будем измерять главным образом то, есть ли у нас |0⟩ или |1⟩; т. е. мы будем измерять вдоль линии между |0⟩ и |1⟩. Для удобства мы даем этой оси название: ось *Z*. Мы можем представить это визуально, *спроецировав* наш вектор состояния на ось *Z* (см. рис. 2.15), используя внутреннее произведение.

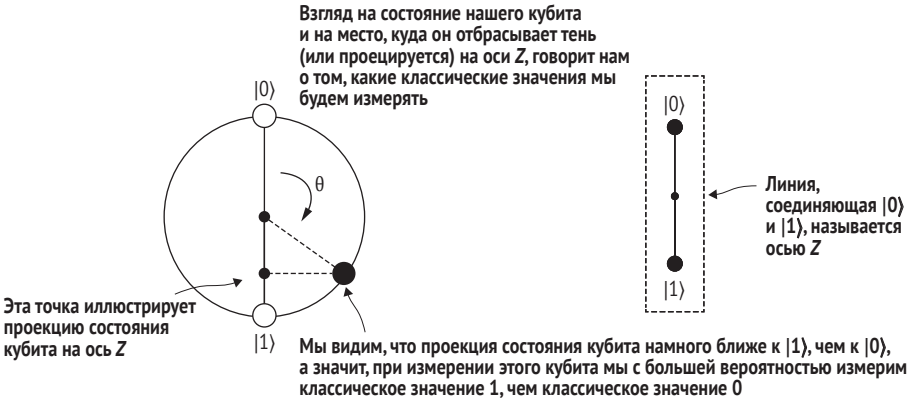


Рис. 2.15 Визуализация квантового измерения, которое можно рассматривать как проецирование состояния в том или ином направлении. Например, если кубит повернут так, что его состояние близко к состоянию |1⟩, то его измерение с большей вероятностью вернет 1, чем 0

ДЛЯ СПРАВКИ Если вам нужно освежить свои знания по теме внутренних произведений, то рекомендуем обратиться к приложению С к книге

Подумайте о подсветке фонариком с того места, откуда мы рисуем состояние кубита назад на ось Z; вероятность получить результат 0 либо 1 определяется тенью, которую состояние оставляет на оси Z.

Глубокое погружение: почему квантовое измерение не линейно?

После того как мы придали такое большое значение линейности квантовой механики, может показаться странным, что мы сразу же вводим измерение как нелинейное. Если нам разрешены нелинейные операции, такие как измерение, можем ли мы также имплементировать другие нелинейные операции, такие как клонирование кубитов?

Короткий ответ таков – хотя все согласны с математикой, лежащей в основе операции измерения, все еще есть изрядное количество философских дискуссий о том, как лучше всего понимать, почему квантовое измерение действует так, как оно действует. Эти дискуссии подпадают под название «*квантовые основы*» и пытаются сделать больше, чем просто понять, что такое квантовая механика и что конкретно она предсказывает, также разбираясь в причинах и отвечая на вопрос «*почему*». По большей части основы разведывают разные способы *интерприрования* квантовой механики. Подобно тому, как мы способны понять классическую вероятность, обращаясь к контринтуитивным мысленным экспериментам, таким как стратегии игровых шоу или то, как казино могут выигрывать даже в тех играх, которые, кажется, приносят одни убытки, квантовые основы сконцентрированы на разработке новых интерпретаций с помощью небольших мысленных экспериментов, которые зондируют разные аспекты квантовой механики. Например, некоторые результаты квантовых основ помогают нам понять смысл измерений.

В частности, одно критическое наблюдение состоит в том, что мы всегда можем снова сделать квантовые измерения линейными, включив в наше описание состояние измерительного аппарата; мы увидим некоторые необходимые для этого математические инструменты в главах 4 и 6. Если это наблюдение довести до крайности, то оно приводит к таким интерпретациям, как *многомировая интерпретация*. Многомировая интерпретация решает интерпретацию измерения, настаивая на том, что мы рассматриваем только те состояния, которые включают в себя измерительные устройства, вследствие чего кажущаяся нелинейность измерения на самом деле не существует.

В другой крайности мы можем интерпретировать квантовое измерение, отметив, что нелинейность в квантовых измерениях точно такая же, как и в отрасли статистики, именуемой байесовым выводом. Отсюда квантовая механика кажется нелинейной только тогда, когда мы забываем включить в рассуждения тот факт, что агент выполняет измерение и учится на каждом результате. Это наблюдение приводит к тому, что квантовая механика рассматривается не как описание мира, а как описание того, что мы знаем о мире.

Хотя эти два вида интерпретаций расходятся на философском уровне, оба предлагают разные способы урегулирования того, как линейная теория, такая как квантовая механика, может иногда казаться нелинейной. Независимо от того, какая интерпретация помогает вам понять взаимодействие между измерением и остальной частью квантовой механики, вы можете утешиться тем, что результаты измерений всегда описываются одной и той же математикой и симуляциями. И действительно, полагаться на симуляцию (иногда саркастически именуемую интерпретацией «заткнись и калькулируй») – самая старая и самая знаменитая из всех интерпретаций.

Квадрат длины каждой проекции *представляет* вероятность того, что измеряемое нами состояние будет найдено в этом направлении. Если у нас есть кубит в состоянии $|0\rangle$ и мы попытаемся измерить его вдоль направления состояния $|1\rangle$, то мы получим вероятность нуля, потому что состояния противоположны друг другу, когда мы рисуем их на окружности. Думая в терминах рисунков, состояние $|0\rangle$ не имеет проекции на состояние $|1\rangle$ – в смысле рис. 2.15, $|0\rangle$ не оставляет тени на $|1\rangle$.

ДЛЯ СПРАВКИ Если некое событие происходит с вероятностью 1, то это событие происходит *всегда*. Если некое событие происходит с вероятностью 0, то это событие *невозможно*. Например, вероятность того, что типичный шестигранный кубик выпадет гранью 7, равна нулю, поскольку это выпадение невозможно. Схожим образом, если кубит находится в состоянии $|0\rangle$, то получение результата 1 от измерения по оси Z невозможно, так как $|0\rangle$ не имеет проекции на $|1\rangle$.

Однако если у нас есть $|0\rangle$ и мы попытаемся измерить его вдоль направления $|0\rangle$, то мы получим вероятность 1, потому что состояния параллельны (и по определению имеют длину 1). Давайте пройдемся по операции измерения состояния, которое не является ни параллельным, ни перпендикулярным.

Пример

Скажем, у нас есть кубит в состоянии $(|0\rangle + |1\rangle)/\sqrt{2}$ (то же самое, что и $|+\rangle$ из табл. 2.1), и мы хотим измерить его или спроецировать вдоль оси Z. Тогда мы можем найти вероятность того, что классический результат будет равен 1 путем проецирования $|+\rangle$ на $|1\rangle$.

Мы можем найти проекцию одного состояния на другое, используя *внутреннее произведение* между их векторными представлениями. В этом случае мы записываем внутреннее произведение $|+\rangle$ и $|1\rangle$ как $\langle 1 \mid +\rangle$, где $\langle 1|$ – это транспозиция $|1\rangle$ и где стыковка двух вертикальных черт друг против друга означает взятие внутреннего произведения. Позже мы увидим, что $\langle 1|$ является конъюгатной транспозицией $|1\rangle$ и называется «бра».

Мы можем записать это следующим образом:

В целях вычисления проекции мы начинаем с записи «бра», на который мы хотим спроецировать.

$\langle 1| \, (|0\rangle + |1\rangle)/\sqrt{2}$

Далее мы распределяем бра.

$= (1/\sqrt{2})(\langle 1|0\rangle + \langle 1|1\rangle)$

Мы можем записать каждую бра-кетную пару как внутреннее произведение между двумя векторами.

$= (1/\sqrt{2})([[0], [1] \cdot [[1], [0]] + [[0], [1]] \cdot [[0], [1]])$

Вычисление каждого внутреннего произведения делает все намного проще!

$= (1/\sqrt{2})(0 + 1)$

Теперь у нас есть наложение между $|0\rangle$ и $|1\rangle$.

$= 1/\sqrt{2}$.

Для превращения этой проекции в вероятность мы возводим ее в квадрат, получив, что вероятность наблюдать исход 1 при подготовке состояния $|+\rangle$ равна 1/2.

Мы часто проецируем на ось *Z*, потому что это удобно во многих реальных экспериментах, но мы также можем измерять вдоль оси *X*, чтобы увидеть, есть ли у нас состояние $|+\rangle$ либо $|-\rangle$. Измеряя вдоль оси *X*, мы с уверенностью получаем $|+\rangle$ и никогда не получаем $|-\rangle$, как показано на рис. 2.16.

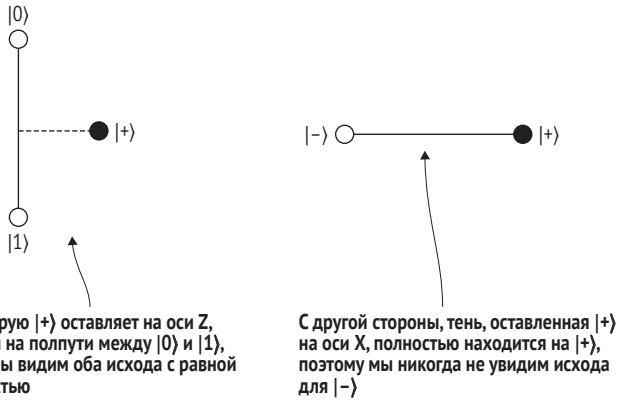


Рис. 2.16 Измерение $|+\rangle$ вдоль оси X всегда приводит к $|+\rangle$. Для того чтобы это увидеть, обратите внимание, что «тень», оставленная состоянием $|+\rangle$ на оси X (т. е. линия между состояниями $|-\rangle$ и $|+\rangle$), является как раз самим состоянием $|+\rangle$

ПРИМЕЧАНИЕ Мы можем получить полностью определенный результат измерения *только* потому, что в этом случае мы заранее знаем «правильное» направление измерения – если же нам передано состояние без какой-либо информации о том, каким является «правильное» направление измерения, то мы не сможем идеально предсказать какой-либо результат измерения.

2.3.4 Обобщение измерения: независимость от базиса

Иногда мы можем не знать о том, как наш кубит был подготовлен, поэтому мы не знаем, как измерять биты надлежащим образом. В более общем случае любая пара состояний, которые не перекрываются (являясь противоположными полюсами), определяет измерение одинаково. Фактическим результатом измерения является значение классического бита, которое указывает на то, с каким полюсом состояние выровнено при выполнении измерения.

Еще более общие измерения

Квантовая механика допускает гораздо более общие виды измерений – мы увидим некоторые из них по ходу работы, но в этой книге мы в основном сосредоточимся на случае проверки между двумя противоположными полюсами. Этот выбор является довольно удобным видом контролирования большинства квантовых устройств и может использоваться практически в любой из коммерческих платформ квантовых вычислений, доступных в настоящее время.

Математически мы используем обозначения ⟨измерение|состояние⟩ для представления операции измерения кубита. Левый компонент, ⟨измерение|, называется *бра*, и мы уже встречали компонент *кет* справа. Вместе они называются *бракетом*!

Компоненты бра очень похожи на компоненты кет, за исключением того, что для переключения с одного на другой мы должны выполнить транспонирование (превратить строки в столбцы и наоборот) бра или кета, которые у нас есть:

$$|0\rangle^T = [[1], [0]]^T = [[1, 0]].$$

На это можно взглянуть и по-другому, как на то, что транспонирование превращает векторы-столбцы (кет-векторы) в векторы-строки (бра-векторы).

ПРИМЕЧАНИЕ Поскольку сейчас мы работаем только с действительными числами, нам не нужно будет делать ничего другого, чтобы перейти от кетов к бра. Но когда мы будем работать с комплексными числами в следующей главе, нам также понадобится комплексный конъюгат.

Бра позволяют нам записывать измерения. Но для того чтобы увидеть, что на самом деле *делают* измерения, нам нужна еще одна вещь в нашем распоряжении: правило о том, как использовать бра и кеты вместе, чтобы получать *вероятность* увидеть этот результат измерения. В квантовой механике вероятности измерения отыскиваются, глядя на длину проекции или тени, которую кет для состояния оставляет на бра для измерения. По опыту мы знаем, что можем найти проекции и длины, используя внутренние произведения. В нотации Дирака внутреннее произведение бра и кета записывается как ⟨измерение|состояние⟩, давая нам именно то правило, которое нам нужно.

Например, если мы подготовили состояние $|+\rangle$ и хотим знать вероятность того, что мы наблюдаем 1 при измерении в Z-базисе, то, проецируя, как показано на рис. 2.15, мы можем найти нужную нам длину. Проекция $|+\rangle$ на $\langle 1|$ говорит нам о том, что мы видим результат 1 с вероятностью $\Pr(1|+) = |\langle 1|+\rangle|^2 = |\langle 1|0\rangle + \langle 1|1\rangle|^2/2 = |0 + 1|^2/2 = 1/2$. Таким образом, в 50 % случаев мы получим исход 1. В остальных 50 % случаев мы получим исход 0.

Правило Борна

Если у нас есть квантовое состояние |состояние⟩ и мы выполняем измерение вдоль направления ⟨измерение|, то мы можем записать вероятность того, что будем наблюдать *измерение* в качестве нашего результата, как

$\Pr(\text{измерение}|\text{состояние}) = |\langle \text{измерение}|\text{состояние}\rangle|^2$.

Другими словами, вероятность – это квадрат магнитуды внутреннего произведения бра измерения и кета состояния.

Это выражение называется *правилом Борна*.

В табл. 2.3 мы перечислили несколько других примеров использования правила Борна для предсказания того, какие классические биты мы получим при измерении кубитов.

Таблица 2.3 Примеры использования правила Борна для отыскания вероятностей измерений

Если мы подготавливаем...	...и измеряем...	...то мы увидим этот исход с этой вероятностью
$\|0\rangle$	$\langle 0\|$	$\|\langle 0\|0\rangle\|^2 = 1$
$\|0\rangle$	$\langle 1\|$	$\|\langle 1\|0\rangle\|^2 = 0$
$\|0\rangle$	$\langle +\|$	$\|\langle +\|0\rangle\|^2 = \|(\langle 0\| + \langle 1\|)\|0\rangle/\sqrt{2}\|^2 = (1/\sqrt{2} + 0)^2 = 1/2$
$\|+\rangle$	$\langle +\|$	$\|\langle +\|+\rangle\|^2 = \|(\langle 0\| + \langle 1\|)(\|0\rangle + \|1\rangle)/2\|^2 = \|\langle 0\|0\rangle + \langle 1\|0\rangle + \langle 0\|1\rangle + \langle 1\|1\rangle\|^2/4 = \|1 + 0 + 0 + 1\|^2/4 = 2^2/4 = 1$
$\|+\rangle$	$\langle -\|$	$\|\langle -\|+\rangle\|^2 = 0$
$-\|0\rangle$	$\langle 0\|$	$\|-\langle 0\|0\rangle\|^2 = \|-1\|^2 = 1^2$
$-\|+\rangle$	$\langle -\|$	$\|-\langle +\|-\rangle\|^2 = \|(-\langle 0\| - \langle 1\|)(\|0\rangle - \|1\rangle)/2\|^2 = \|-\langle 0\|0\rangle - \langle 1\|0\rangle + \langle 0\|1\rangle + \langle 1\|1\rangle\|^2/4 = \|-1 - 0 + 0 + 1\|^2/4 = 0^2/4 = 0$

ДЛЯ СПРАВКИ В табл. 2.3 мы используем тот факт, что $\langle 0|0\rangle = \langle 1|1\rangle = 1$ и $\langle 0|1\rangle = \langle 1|0\rangle = 0$. (Попробуйте проверить это сами!) Когда два состояния имеют внутреннее произведение, равное нулю, мы говорим, что они *ортогональны* (или *перпендикулярны*). Тот факт, что $|0\rangle$ и $|1\rangle$ ортогональны, облегчает быстрое выполнение многих вычислений.

Теперь мы рассмотрели все, что нам нужно знать о кубитах, чтобы иметь возможность их симулировать! Давайте рассмотрим требования, которые нам нужно было удовлетворить, чтобы обеспечить наличие рабочих кубитов.

Кубит

Кубит – это любая физическая система, удовлетворяющая трем свойствам:

- система может быть идеально просимулирована при наличии сведений о векторе чисел (состоянии);
- система может быть преобразована с помощью квантовых операций (например, поворотов);
- любое измерение системы производит один классический бит информации, следуя правилу Борна.

Всякий раз, когда у нас есть кубит (система с тремя приведенными выше свойствами), мы можем его описать, используя тот же математический аппарат или симуляционный код, без дополнительной ссылки на то, с какой системой мы работаем. Это похоже на ситуацию, когда нам не нужно знать о том, чем определяется бит: направлением движения шарика пинбола или напряжением в транзисторе, чтобы написать вентили NOT и AND, или написать программное обеспечение, в котором эти вентили используются для выполнения интересных вычислений.

ПРИМЕЧАНИЕ По аналогии с тем, как мы используем слово «бит», одновременно обозначая и физическую систему, хранящую информацию, и информацию, хранящуюся в бите, мы также используем и слово «кубит», одновременно обозначая и квантовое устройство, и квантовую информацию, хранящуюся в этом устройстве.

Фаза

В последних двух строках табл. 2.3 мы увидели, что умножение состояния на фазу –1 не влияет на вероятности измерений. Это не является простым совпадением и скорее указывает на одну из самых интересных вещей о кубитах. Поскольку правило Борна заботится только о квадрате абсолютного значения внутреннего произведения состояния и измерения, умножение числа на (–1) не влияет на его абсолютное значение. Мы называем такие числа,

как +1 или −1, абсолютное значение которых равно 1, *фазами*. В следующей главе мы узнаем о фазах гораздо больше, когда будем больше работать с комплексными числами.

А пока мы будем говорить, что умножение всего вектора на −1 является примером применения *глобальной* фазы, тогда как переход из $|+\rangle$ в $|-\rangle$ является примером применения *относительной* фазы между $|0\rangle$ и $|1\rangle$. Хотя глобальные фазы никогда не влияют на результаты измерений, существует большая разница между состояниями $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ и $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$: коэффициенты перед $|0\rangle$ и $|1\rangle$ одинаковы в $|+\rangle$ и отличаются на фазу (-1) в $|-\rangle$. Мы увидим гораздо больше различий между этими двумя концепциями в главах 3, 4, 6 и 7.

2.3.5 Симулирование кубитов в исходном коде

Теперь, когда мы рассмотрели способы описания состояний, операций и измерений кубитов, самое время посмотреть на то, как представлять все эти понятия в исходном коде. Мы будем использовать сценарий с участием нашей подруги Евы, чтобы мотивировать исходный код, который мы пишем.

Предположим, мы хотели бы сохранить наши 💔 отношения с Евой в секрете, чтобы никто другой не узнал. Как зашифровать наше сообщение Еве так, чтобы только она могла его прочитать?

Мы разведаем это приложение подробнее в следующей главе, но самый базовый шаг, необходимый для любого хорошего *алгоритма шифрования*, – иметь источник случайных чисел, которые трудно предсказывать. Давайте напишем точно, как мы будем совмещать наши секретные и случайные биты, чтобы создать безопасное сообщение для отправки Еве. Рисунок 2.17 показывает, что если вы и Ева знаете одну и ту же секретную последовательность случайных классических битов, то мы можем использовать эту последовательность для безопасного общения. В начале главы мы увидели, как писать сообщение, или *открытый текст*, который мы хотим отправить Еве (в данном случае «💔»), в виде строки классических битов. Одноразовый блокнот – это последовательность случайных классических битов, которая действует как способ скремблирования или шифрования нашего сообщения. Указанное скремблирование выполняется путем взятия битового XOR сообщения и битов одноразового блокнота для каждой позиции в последовательности. В результате производится последовательность классических битов, именуемая *шифротекстом*. Для любого, кто попытается прочитать наше сообщение, шифротекст будет выглядеть как случайные биты. Например, невозможно установить, чему равен бит в шифротексте, 1 или 0, и все из-за открытого текста и одноразового блокнота.

Вы, возможно, спросите, как мы получаем строки случайных битов для нашего одноразового блокнота. А мы можем сделать наш собственный QRNG-генератор с помощью кубитов! Это, возможно, покажется стран-

ным, но, чтобы сделать наш QRNG-генератор, мы будем симулировать кубиты классическими битами. Случайные числа, которые он генерирует, будут не более безопасными, чем компьютер, который мы используем для симуляции, но этот подход дает нам хорошее начало в понимании принципа работы кубитов.

Рис. 2.17 Пример того, как использовать случайные биты для шифрования секретов, даже через интернет или другую ненадежную сеть. Здесь мы пытаемся безопасно отправить сообщение «♥». Если мы с Евой начнем с совместного секрета «✉», то мы сможем использовать его как одноразовый блокнот для защиты нашего сообщения

Давайте отправим Еве наше сообщение! Подобно тому, как классический бит может быть представлен в исходном коде значениями `True` и `False`, как мы увидели, мы можем представить два состояния кубита $|0\rangle$ и $|1\rangle$ в виде *векторов*. То есть состояния кубита представляются в коде в форме списков *списков чисел*.

Листинг 2.3 Представление кубитов в исходном коде с помощью NumPy

```
>>> import numpy as np    ❶
>>> ket0 = np.array(      ❷
...     [[1], [0]]
... )
>>> ket0
array([[1],               ❸
       [0]])
>>> ket1 = np.array(
...     [[0], [1]]
... )
>>> ket1
array([[0],
       [1]])
```

❶ Мы используем Python'овский пакет NumPy для представления векторов, так как NumPy сильно оптимизирована и сделает нашу жизнь намного проще.
❷ Мы называем нашу переменную ket0 в честь нотации |0⟩, в которой мы помечаем кубитовые состояния кетной половиной скобок ⟨|.
❸ NumPy напечатает векторы 2×1 в виде столбцов.

Как мы увидели ранее, мы можем сконструировать другие состояния, такие как |+⟩, используя линейные комбинации |0⟩ и |1⟩. Точно в том же смысле мы можем использовать NumPy для сложения векторных представлений |0⟩ и |1⟩ для конструирования векторного представления |+⟩.

Листинг 2.4 Векторное представление |+⟩

```
>>> ket_plus = (ket0 + ket1) / np.sqrt(2)    ❶
>>> ket_plus
array([[0.70710678+0.j],                     ❷
       [0.70710678+0.j]])
```

❶ В NumPy используются векторы для хранения состояния |+⟩, т. е. линейная комбинация |0⟩ и |1⟩.
❷ В этой книге мы будем встречать число 0.70710678 много раз, так как оно является хорошей аппроксимацией $\sqrt{2}$, длины вектора [[1], [1]].

Линейные операторы и квантовые операции

Описывать квантовые операции как линейные операторы является хорошим началом, но не все линейные операторы являются валидными квантовыми операциями! Если бы мы могли имплементировать операцию, описываемую линейным оператором, таким как $2\times\mathbb{1}$ (т. е. вдвое больше оператора тождественности), то мы могли бы нарушить правило, что вероятности всегда являются числами между нулем и единицей. Мы также требуем, чтобы все квантовые операции, отличные от операции измерения, были *обратимыми*, поскольку это является фундаментальным свойством квантовой механики.

Оказывается, что операции, реализуемые в квантовой механике, описываются матрицами U, инверсии U^{-1} которых можно вычислить, взяв конъюгатную транспозицию, $U^{-1} = U^{+}$. Такие матрицы называются *унитарными матрицами*.

Визуализация типов валидных квантовых операций. Все унитарные операторы являются линейными, но не все линейные операторы являются унитарными. Обратимые квантовые операции (т. е. кроме операции измерения) представлены операторами, которые являются не просто линейными, но еще и унитарными

Если бы мы захотели просимулировать то, как операция трансформирует список битов, в классической логике, то мы могли бы использовать таблицу истинности. Схожим образом, поскольку квантовые операции, отличные от операции измерения, всегда линейны, для симулирования того, как операция преобразовывает состояние кубита, мы можем использовать матрицу, которая сообщает нам о том, как преобразовывается каждое состояние.

Одной особенно важной квантовой операцией является *операция Адамара*, которая преобразовывает $|0\rangle$ в $|+\rangle$ и $|1\rangle$ в $|-\rangle$. Как мы видели ранее, измерение $|+\rangle$ вдоль оси Z в результате дает либо 0, либо 1 с равной вероятностью. Поскольку для отправки секретных сообщений нам нужны случайные биты, операция Адамара действительно полезна для создания нашего QRNG-генератора.

Используя векторы и матрицы, мы можем определить операцию Адамара, составив таблицу того, как она действует на состояния $|0\rangle$ и $|1\rangle$, как показано в табл. 2.4.

Поскольку квантовая механика линейна, это описание операции Адамара является полным!

Таблица 2.4 Представление операции Адамара в виде таблицы

Входное состояние	Выходное состояние
$\|0\rangle$	$\|+\rangle = (\|0\rangle + \|1\rangle)/\sqrt{2}$
$\|1\rangle$	$\|-\rangle = (\|0\rangle - \|1\rangle)/\sqrt{2}$

В матричной форме мы записываем табл. 2.4 как `H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)`.

Листинг 2.5 Определение операции Адамара

```
>>> H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)      ❶
>>> H @ ket0
array([[0.70710678],
       [0.70710678]])
>>> H @ ket1
array([[ 0.70710678],
       [-0.70710678]])
```

❶ Мы определяем переменную *H* для хранения матричного представления *H* операции Адамара, которое мы видели в табл. 2.4. Нам понадобится *H* на протяжении остальной части этой главы, поэтому полезно определить ее здесь.

Операция Адамара

Операция Адамара – это квантовая операция, которую можно просимулировать с помощью приведенного ниже линейного преобразования:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Любая операция с квантовыми данными может быть записана в форме матрицы в таком ключе. Если мы хотим преобразовать |0⟩ в |1⟩ и наоборот (квантовое обобщение классической операции NOT, которую мы встречали ранее, соответствующей повороту на 180°), то мы делаем то же самое, что мы делали для определения операции Адамара.

Листинг 2.6 Представление квантового вентиля NOT

```
>>> X = np.array([[0, 1], [1, 0]])    ❶
>>> X @ ket0
array([[0],
       [1]])
>>> (X @ ket0 == ket1).all()          ❷
True
>>> X @ H @ ket0                      ❸
array([[0.70710678],
       [0.70710678]])
```

❶ Квантовая операция, соответствующая классической операции NOT, обычно называется операцией *x*; мы представляем матрицу для *x* Python'овской переменной *X*.

❷ Мы можем подтвердить, что *X* преобразовывает |0⟩ в |1⟩. Метод all() пакета NumPy возвращает True, если каждый элемент *X* @ ket0 == ket1 истинен: т. е. если каждый элемент массива *X* @ ket0 равен соответствующему элементу в ket1.

❸ Оператор *x* ничего не делает с $H|0\rangle$. Мы можем это подтвердить, снова используя оператор @, чтобы умножить *X* на значение, представляющее состояние $|+\rangle = H|0\rangle$. Мы можем выразить это значение как *H* @ ket0.

Операция *x* на последнем входном значении ничего не делает, потому что операция *x* меняет местами |0⟩ и |1⟩. Состояние $H|0\rangle$, также именуемое |+⟩, уже является суммой двух кетов: $(|0\rangle + |1\rangle)/\sqrt{2} = (|1\rangle + |0\rangle)/\sqrt{2}$, и поэтому своп в операции *x* ничего не делает.

Вспоминая аналогию с картой местности из приложения С к книге, мы можем рассматривать матрицу *H* как *отражение* направления ⤢, как показано на рис. 2.18.

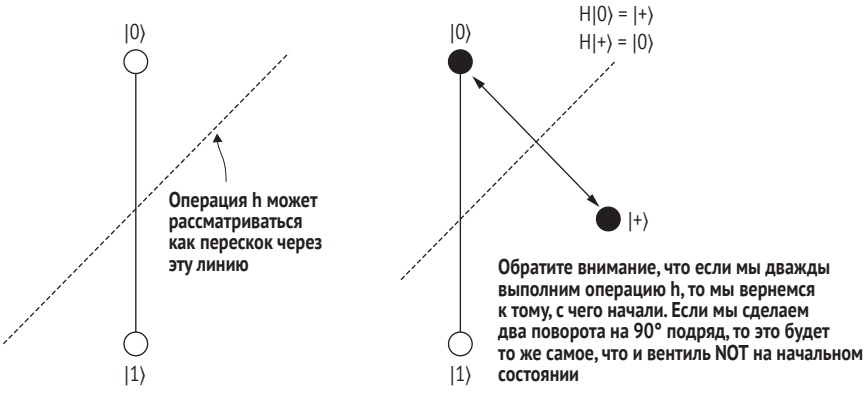
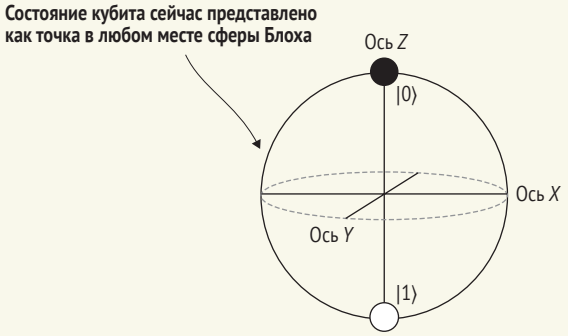


Рис. 2.18 Операция h как отражение или перескок через ⤢. В отличие от поворота на 90°, двухкратное применение h возвращает кубит в состояние, в котором он начал. На h можно взглянуть и по-другому, как на то, что отражение ⤢ меняет роль осей *X* и *Z*

Нас ждет третья размерность!

В отношении кубитов аналогия с картой местности из приложения С помогает нам понять, как писать состояния одиночных кубитов и ими манипулировать. Однако до сих пор мы рассматривали только те состояния, которые могут быть записаны с использованием действительных чисел. В общем случае в квантовых состояниях могут использоваться комплексные числа. Если мы перестроим нашу карту местности и сделаем ее трехмерной, то мы сможем без каких-либо проблем включать комплексные числа. Этот образ мышления о кубитах называется *сферой Блоха* и бывает очень полезен для рассмотрения квантовых операций, таких как повороты и отражения, как мы увидим в главе 6.



В более общем случае мы можем визуализировать состояния кубита как точки на сфере, а не только на окружности. Для этого, однако, требуется использовать комплексные числа – мы узнаем об этом больше в главе 6

Глубокое погружение: бесконечно много состояний?

Из рисунка на приведенной выше панели может показаться, что существует бесконечно много разных состояний кубита. Для любых двух разных точек на сфере мы всегда можем найти точку, которая находится «между» ними. Хотя это утверждение является истинным, оно также слегка вводит в заблуждение. Если на мгновение задуматься о классической ситуации, то монета, которая приземляется орлами в 90 % случаев, отличается от монеты, которая приземляется орлами в 90.0000000001 % случаев. На самом деле мы всегда можем сделать монету, смещение которой находится «между» смещением двух других монет. Однако подбрасывание монеты может давать нам только один классический бит информации. В среднем потребовалось бы около 10^{23} подбрасываний, чтобы отличить монету, которая приземляется орлами в 90 % случаев, от той, которая приземляется орлами в 90.0000000001 % случаев. Мы можем рассматривать эти две монеты как идентичные, потому что не можем провести эксперимент, который бы надежно их различал. По аналогии, для квантовых вычислений существуют пределы нашей способности различать бесконечное множество разных квантовых состояний, которые мы видим из рисунка сферы Блоха.

Тот факт, что кубит имеет бесконечно много состояний, не делает его уникальным. Иногда люди говорят, что квантовая система может находиться «в бесконечно многих состояниях одновременно», и по этой причине квантовые компьютеры могут предложить ускорение. *Это неправда*! Как указывалось ранее, мы не можем различать состояния, которые находятся очень близко друг к другу, поэтому часть утверждения, где говорится о «бесконечности», не может быть тем, что дает квантовым вычислениям преимущество. Мы еще поговорим о той части, где говорится об «одновременности» в следующих главах, но достаточно сказать, что вовсе не число состояний, в которых наш кубит может находиться, делает квантовые компьютеры крутыми!

2.4 Программирование рабочего QRNG-генератора

Теперь, когда у нас есть несколько квантовых концепций, с которыми можно играть, давайте применим то, что мы узнали, для программирования QRNG-генератора, чтобы иметь возможность отправлять 💔 сообщения без каких-либо забот. Мы собираемся построить QRNG-генератор, который возвращает либо 0, либо 1.

Случайные биты или случайные числа?

То, что наш QRNG-генератор может выдавать только одно из двух чисел, 0 либо 1, может показаться ограничением. Напротив: этого достаточно для генерирования случайных чисел в диапазоне от 0 до N для любого положительного целого числа N. Проще всего это увидеть, начав с особого случая, когда N равно $2^n - 1$ для некоторого положительного целого числа n, и в этом случае мы просто записываем наши случайные числа в виде n-битовых строк. Например, мы можем создавать случайные числа от 0 до 7, генерируя три случайных бита r_0, r_1 и r_2, а затем возвращая $4r_2 + 2r_1 + r_0$.

Ситуация немного сложнее, если N не задается степенью двойки, поскольку у нас есть «оставшиеся» возможности, с которыми нам нужно как-то работать. Например, если нам нужно бросить шестигранный кубик, но у нас есть только восьмигранный кубик (возможно, вчера вечером мы были друидом в ролевой игре), тогда нам нужно решить, что делать, когда этот кубик повернется гранью 7 или 8. Лучшее, что можно сделать, если мы хотим получить справедливый шестигранный кубик, – просто бросить его повторно, когда это произойдет. Используя этот подход, мы можем строить произвольные справедливые кубики из подбрасываний монет – удобно для любой игры, в которую мы хотим играть. Короче говоря, мы не ограничены лишь двумя исходами из нашего QRNG-генератора!

Как и в любой квантовой программе, наша программа QRNG будет представлять собой последовательность инструкций для устройства, которое выполняет операции с кубитом (см. рис. 2.19). В псевдокоде квантовая программа для имплементирования QRNG-генератора состоит из трех инструкций:

1. Подготовить кубит в состоянии $|0\rangle$.
2. Применить операцию Адамара к нашему кубиту, чтобы он находился в состоянии $|+\rangle = H|0\rangle$.
3. Измерить кубит, чтобы получить результат 0 либо 1 с вероятностью 50/50.

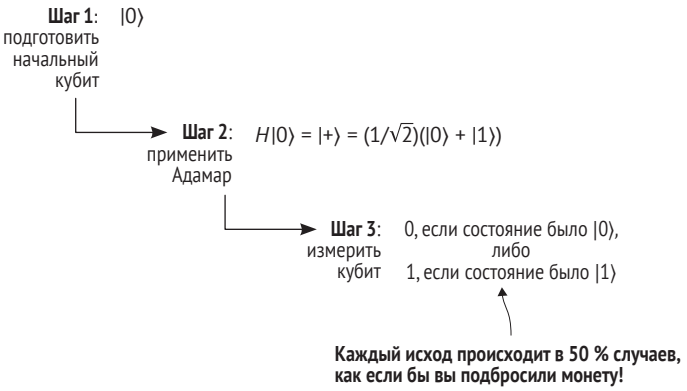


Рис. 2.19 Шаги для написания программы QRNG, которую мы хотим протестировать. Возвращаясь к рис. 2.3, мы можем использовать то, что мы узнали до сих пор, чтобы записывать состояние нашего кубита после каждого шага алгоритма QRNG

То есть нам нужна программа, которая выглядит примерно так.

Листинг 2.7 Пример псевдокода программы QRNG

```
def qrng():
    q = Qubit()
    H(q)
    return measure(q)
```

Задействуя матричное умножение, мы можем использовать классический компьютер, такой как ноутбук, чтобы просимулировать поведение `qrng()` на идеальном квантовом устройстве. Наша программа `qrng` вызывает программный стек (см. рис. 2.20), который абстрагируется от того, что конкретно мы используем: классический симулятор или же реальное квантовое устройство.

В стеке много частей, но не волнуйтесь – мы поговорим о них по ходу дела. А пока мы сосредоточимся на верхнем разделе (обозначенном на рисунке как «Классический компьютер») и начнем с написания исходного кода для квантовой программы, а также симуляторного бэкенда на Python.

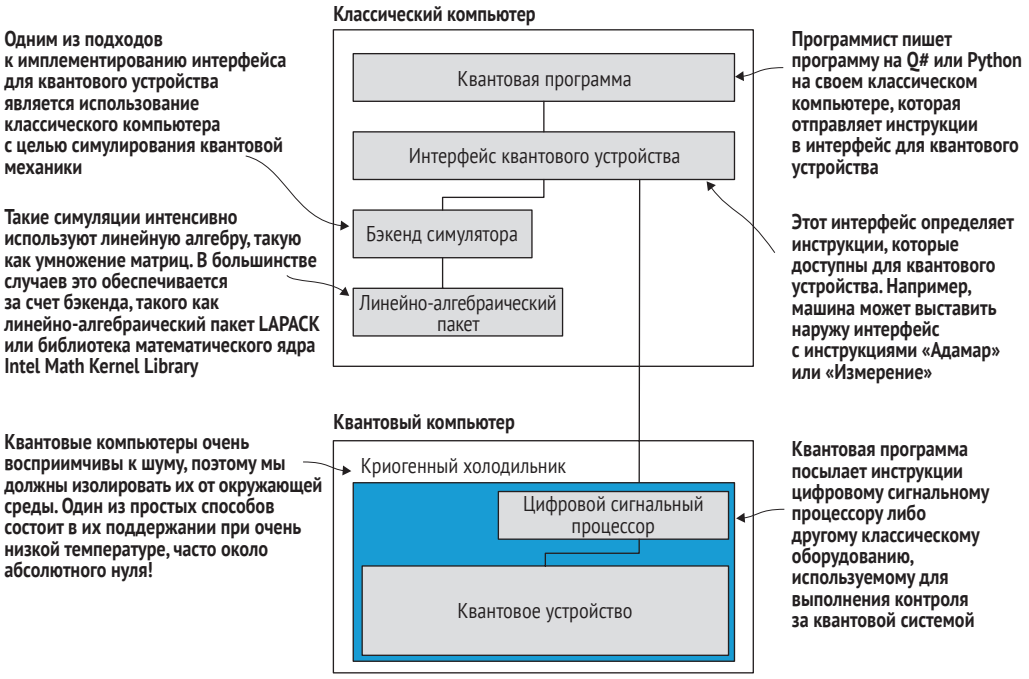


Рис. 2.20 Пример того, как может выглядеть программный стек для квантовой программы

ПРИМЕЧАНИЕ В главе 7 мы перейдем к использованию симуляторного бэкенда, поставляемого вместе с Комплектом инструментов для квантовой разработки от компании Microsoft.

С учетом этого представления о программном стеке мы можем написать нашу симуляцию QRNG-генератора, сначала написав класс `QuantumDevice` с абстрактными методами выделения кубитов, выполнения операций и измерения кубитов. Затем мы можем имплементировать этот класс с симулятором и вызвать этот симулятор из `qrng()`.

В целях дизайна интерфейса для нашего симулятора так, как показано на рис. 2.20, давайте перечислим все, что наше квантовое устройство должно делать. Прежде всего пользователи должны иметь возможность выделять и возвращать кубиты.

Листинг 2.8 interface.py: интерфейс с квантовым устройством как абстрактные методы

```
class QuantumDevice(metaclass=ABCMeta):
    @abstractmethod
    def allocate_qubit(self) -> Qubit:          ❶
        pass

    @abstractmethod
    def deallocate_qubit(self, qubit: Qubit):   ❷
        pass
```

```
    @contextmanager
    def using_qubit(self):                          ❸
        qubit = self.allocate_qubit()
        try:
            yield qubit
        finally:
            qubit.reset()                           ❹
            self.deallocate_qubit(qubit)
```

❶ Любая имплементация квантового устройства должна имплементировать этот метод, позволяя пользователям получать кубиты.
❷ Когда пользователи закончат работу с кубитом, имплементация метода deallocate_qubit позволит пользователям возвращать кубит в устройство.
❸ Мы можем предоставить Python'овский контекстный менеджер, чтобы упростить безопасное выделение и высвобождение кубитов.
❹ Контекстный менеджер обеспечивает, чтобы каждый кубит сбрасывался и высвобождался перед возвращением в классический компьютер независимо от того, какие исключения возникают.

Сами кубиты могут затем выставлять наружу фактические преобразования, которые нам нужны:

- пользователи должны иметь возможность выполнять операции Адамара с кубитами;
- пользователи должны иметь возможность измерять кубиты, чтобы получать классические данные.

Листинг 2.9 interface.py: интерфейс с кубитами на квантовом устройстве

```
from abc import ABCMeta, abstractmethod
from contextlib import contextmanager
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass                   ❶

    @abstractmethod                     ❷
    def measure(self) -> bool: pass

    @abstractmethod
    def reset(self): pass               ❸
```

❶ Метод h преобразовывает кубит прямо на месте (не делая копию), используя операцию Адамара np.array([[1, 1], [1, -1]]) / np.sqrt(2).
❷ Метод measure позволяет пользователям измерять кубиты и извлекать классические данные.
❸ Метод reset облегчает пользователям подготовку кубита заново с нуля.

Имея все на своих местах, мы можем вернуться к нашему определению qrng, используя эти новые классы.

Листинг 2.10 qrng.py: определение устройства qrng

```
def qrng(device: QuantumDevice) -> bool:
    with device.using_qubit() as q:
        q.h()
        return q.measure()
```

Если имплементировать интерфейс `QuantumDevice` с классом `SingleQubitSimulator`, то мы можем передавать его `qrng` для выполнения нашей имплементации QRNG-генератора на симуляторе.

Листинг 2.11 qrng.py: определение функции main для qrng.py

```
if __name__ == "__main__":
    qsim = SingleQubitSimulator()
    for idx_sample in range(10):
        random_sample = qrng(qsim)
        print(f"QRNG-генератор вернул {random_sample}.")
```

Теперь у нас есть все, чтобы написать класс `SingleQubitSimulator`. Мы начнем с определения набора констант для вектора |0⟩ и матричного представления операции Адамара.

Листинг 2.12 simulator.py: определение полезных констант

```
KET_0 = np.array([
    [1],
    [0]
], dtype=complex)                    ❶
H = np.array([
    [1,  1],
    [1, -1]
], dtype=complex) / np.sqrt(2)       ❷
```

❶ Поскольку мы будем использовать |0⟩ в симуляторе часто, мы определяем для него константу.
❷ Аналогичным образом мы будем использовать матрицу Адамара *H* для определения того, как операция Адамара преобразовывает состояния, и, значит, мы определяем константу и для нее.

Далее мы определим то, как выглядит просимулированный кубит. С точки зрения симулятора, кубит обертывает вектор, в котором хранится текущее состояние кубита. Для представления состояния нашего кубита мы используем массив NumPy.

Листинг 2.13 simulator.py: определение класса для представления кубитов в нашем устройстве

```
class SimulatedQubit(Qubit):
    def __init__(self):                  ❶
        self.reset()

    def h(self):                         ❷
        self.state = H @ self.state

def measure(self) -> bool:
    pr0 = np.abs(self.state[0, 0]) ** 2  ❸
    sample = np.random.random() <= pr0   ❹
    return bool(0 if sample else 1)      ❺

def reset(self):
    self.state = KET_0.copy()
```

❶ Операция Адамара может быть просимулирована, применив матричное представление *H* к состоянию, которое мы храним в данный момент, а затем обновив его до нового состояния.
❷ В рамках интерфейса Qubit мы обеспечиваем, чтобы метод reset подготавливал наш кубит, приводя его в состояние |0⟩. Мы можем использовать это при создании кубита, обеспечивая, чтобы кубиты всегда начинались в правильном состоянии.
❸ Мы сохранили состояние нашего кубита в виде вектора, поэтому мы знаем, что внутреннее произведение с |0⟩ является просто первым элементом этого вектора. Например, если состояние равно np.array([[a], [b]]) для некоторых чисел *a* и *b*, то вероятность наблюдать исход 0 равна $|a|^2$. Мы можем найти это с помощью np.abs(a) ** 2. Это даст нам вероятность того, что измерение нашего кубита вернет 0.
❹ Для того чтобы превратить вероятность получить 0 в результат измерения, мы генерируем случайное число между 0 и 1, используя np.random.random, и проверяем, что оно меньше pr0.
❺ Наконец, мы возвращаем источнику вызова 0, если у нас 0, и 1, если у нас 1.

Какое случайное число пришло первым: 0 или 1?

При создании этого QRNG-генератора мы должны вызывать *классический* генератор случайных чисел. Такая конфигурация, возможно, покажется чуть-чуть зацикленной, но это происходит потому, что наша классическая симуляция – это именно симуляция. Симуляция QRNG-генератора не случайнее, чем случайность в аппаратном и программном обеспечении, которое мы используем для имплементирования этого симулятора.

С учетом этого квантовая программа qrng.py сама по себе не нуждается в вызове классического генератора случайных чисел, но обращается к симулятору. Если бы мы выполняли qrng.py на реальном квантовом устройстве, то симулятор и классический генератор случайных чисел были бы заменены операциями с фактическим кубитом. В этот момент у нас был бы поток случайных чисел, которые невозможно было бы предсказать вследствие законов квантовой механики.

Выполнив нашу программу, мы теперь получим случайные числа, которые мы и ожидали!

```
$ python qrng.py
QRNG-генератор вернул False.
QRNG-генератор вернул True.
QRNG-генератор вернул True.
QRNG-генератор вернул False.
QRNG-генератор вернул False.
QRNG-генератор вернул True.
QRNG-генератор вернул False.
QRNG-генератор вернул False.
QRNG-генератор вернул False.
QRNG-генератор вернул True.
```

Поздравляем! Мы написали не только нашу первую квантовую программу, но и симуляционный бэкенд и использовали его для выполнения указанной квантовой программы точно так же, как бы мы выполняли ее на реальном квантовом компьютере.

Глубокое погружение: кот Шредингера

Возможно, вы уже встречали или слышали об этой квантовой программе под совсем другим названием. Программа QRNG часто описывается в терминах мысленного эксперимента с котом Шредингера. Предположим, кот находится в закрытой коробке с флаконом яда, который будет выпущен, если некая случайная частица распадется. Прежде чем мы откроем коробку, чтобы это проверить, каким образом можно узнать, жив кот или мертв?

> [Состояние] всей системы выразило бы это, имея в ней живого и мертвого кота, (простите за выражение) перемешанного или размазанного в равных частях.
>
> *– Эрвин Шредингер*

Исторически Шредингер предложил это описание в 1935 году, чтобы выразить свое мнение о том, что некоторые выводы квантовой механики «вызывают смех» с помощью мысленного эксперимента, который подчеркивает, насколько эти выводы противоречат интуиции. Такие мысленные эксперименты, известные как *gedankenexperiment*, являются в физике знаменитой традицией и помогают нам понимать или критиковать разные теории, доводя их до крайних или абсурдных пределов.

Однако, читая о коте Шредингера почти столетие спустя, полезно вспомнить все, что произошло за прошедшие годы. С момента его первого письма мир увидел:

- войну в невиданных ранее масштабах;
- первые шаги, которые человечество предприняло для разведывания пространства за пределами нашей планеты;
- рост числа коммерческих полетов на реактивных самолетах;
- понимание и первые последствия антропогенного изменения климата;
- фундаментальный сдвиг в том, как мы общаемся (от телевидения до интернета);
- широкую доступность дешевых вычислительных устройств;
- открытие удивительного разнообразия субатомных частиц.

Мир, в котором мы живем, – это не тот мир, в котором Шредингер пытался разобраться в квантовой механике! У нас есть много преимуществ в попытках понять, и не последнее из них заключается в том, что мы можем быстро получать доступ к квантовой механике, программируя симуляции с использованием классических компьютеров. Например, инструкция `h`, которую мы увидели ранее, ставит наш кубит в ситуацию, аналогичную ситуации с котом в мысленном эксперименте, но с тем преимуществом, что теперь гораздо легче экспериментировать с нашей программой, чем с мысленным экспериментом. В остальной части книги мы будем использовать квантовые программы для усвоения частей квантовой механики, необходимых для написания квантовых алгоритмов.

Резюме

- Случайные числа помогают в широком спектре приложений, таких как игры, симулирование сложных систем и защита данных.
- Классические биты могут находиться в одном из двух состояний, которые мы традиционно называем 0 и 1.
- Квантовые аналоги классических битов, именуемые *кубитами*, могут находиться либо в состоянии $|0\rangle$, либо в состоянии $|1\rangle$, либо в суперпозициях состояний $|0\rangle$ и $|1\rangle$, например $|+\rangle = 1/\sqrt{2}(|0\rangle + |1\rangle)$.
- Используя операцию Адамара, мы можем подготавливать кубиты, приводя их в состояние $|+\rangle$; затем мы можем измерять кубиты, чтобы генерировать числа, которые гарантированно являются случайными по законам квантовой механики.

3 Обмен секретами с помощью квантового распределения ключей

Эта глава охватывает следующие ниже темы:

- осознание последствий использования квантовых ресурсов для обеспечения безопасности;
- программирование симулятора на языке Python для протокола квантового распределения ключей;
- имплементирование операции квантового NOT.

В предыдущей главе мы начали играть с кубитами и использовали их для строительства квантового генератора случайных чисел с симулятором, который мы собрали на Python. В этой главе мы увидим, что кубиты помогают нам с шифрованием (или другими криптографическими задачами), позволяя нам безопасно *распределять* секретные ключи. Существуют классические методы совместного использования случайных ключей (например, RSA), но они имеют разные гарантии безопасности совместного использования.

3.1 В любви и шифровании все средства хороши

У нас есть квантовый генератор случайных чисел из главы 2, но это только половина того, что нам нужно, чтобы делиться секретами с нашими друзьями. Нам необходимо делиться этими случайными числами с на-

шими друзьями, если мы хотим использовать квантовые случайные числа для безопасного общения с ними. Указанные случайные числа (часто именуемые *ключом*) можно использовать с *алгоритмами шифрования*, в которых случайность ключа сочетается с информацией, которую люди хотят сохранить в секрете таким образом, чтобы кто-то другой с ключом мог прочитать эту информацию. На рис. 3.1 мы видим, как два человека используют ключ (здесь строку случайных битов) для шифровки и дешифровки сообщений между собой.

Рис. 3.1 Ментальная модель того, как мы с Евой используем шифрование для секретного общения, даже через интернет или другую ненадежную сеть

Если мы хотим задействовать кубиты, то можем показать, что использование квантового распределения ключей (QKD) является *доказуемо* безопасным, тогда как классические методы распределения ключей часто являются лишь *вычислительно* безопасными.

ОПРЕДЕЛЕНИЕ *Квантовое распределение ключей* (Quantum key distribution, аббр. QKD) – это протокол обмена информацией, который позволяет пользователям обмениваться квантовыми случайными числами путем обмена кубитами и аутентифицированной классической информацией.

Эта разница не имеет значения для большинства случаев использования. Но если мы являемся группой активистов, правительственной, банковской, журналистской, шпионской или любой другой организацией, в которой обеспечение информационной безопасности является вопросом жизни и смерти, то эта разница на самом деле очень важна.

Когда мы делимся ключом, используя QKD, это не гарантирует, что ключ попадет к партнеру. Это обусловлено тем, что кто-то всегда может совершить DoS-атаку (например, перерезать оптическое волокно между отправителем и получателем), как и в случае любого другого классического протокола. Хорошая аналогия для того, что может обещать QKD, похожа на защищенную от взлома печать на пищевых продуктах. Когда производитель арахисового масла хочет обеспечить, чтобы при открытии нами банки оно было точно таким же, каким было, когда покинуло заводской цех, компания ставит на контейнер защитную от взлома печать. Компания обещает, что если печать попадет к нам (потребителю) в целости и сохранности, то арахисовое масло будет хорошим, и никакая

третья сторона ничего с ним не сделает. Передача криптографического ключа с помощью протокола QKD похожа на установку защищенной от несанкционированного доступа печати на транзитные биты. Если кто-то попытается вскрыть транзитный ключ, то получатель об этом узнает и не будет использовать этот ключ. Однако запечатывание транзитных битов *не* гарантирует, что биты попадут в приемник.

Вычислительная безопасность против доказуемой безопасности

Доказуемая безопасность криптографических протоколов является мечтой. Метод или протокол для криптографической задачи является *доказуемо безопасным*, если мы можем написать доказательство, показывающее, что он безопасен, не используя никаких допущений о злоумышленнике: т. е. что у него может иметься все время и вычислительная мощь Вселенной, а наш протокол останется по-прежнему безопасным! Бóльшая часть нашей текущей криптографической инфраструктуры является *вычислительно безопасной*, которая гарантирует безопасность метода или протокола с разумными допущениями о возможностях злоумышленника. Разработчик или пользователь протокола может выбрать пороговые значения для того, как выглядят конечные компьютерные ресурсы (например, самый большой современный суперкомпьютер или все компьютеры на планете) и каким является разумное время (100 лет, 10 000 лет, возраст Вселенной).

Для имплементирования общей схемы QKD можно использовать целый ряд протоколов. В этой главе мы будем работать с одним из наиболее распространенных протоколов QKD – BB84, однако есть и много других, на которые у нас не будет времени. На протяжении всей этой главы мы будем постепенно к нему приближаться, но на рис. 3.2 показаны шаги к протоколу BB84.

QKD является примером квантовой программы, в которой используется один-единственный кубит и побочная технология из квантовых вычислений. Привлекательным для разработки его делает то, что у нас есть оборудование для его имплементирования уже сегодня! Ряд компаний уже около 15 лет продают оборудование QKD на рынке, но последующие важные шаги этой технологии включают проверку безопасности оборудования и программного обеспечения этих систем.

ПРЕДУПРЕЖДЕНИЕ Примеры, которые мы имплементируем и используем в этой книге, *симулируют* доказуемо безопасные протоколы. Учитывая, что мы не выполняем эти примеры на квантовых устройствах, они *не* являются доказуемо безопасными. Даже при имплементировании этих протоколов с реальным квантовым оборудованием эти доказательства безопасности ничего не делают, чтобы остановить атаки по побочным каналам или социальную инженерию по отделению нас от нашего ключа 😈. Мы подробнее поговорим об упомянутых доказательствах позже в этой главе, когда обсудим теорему о запрете клонирования.

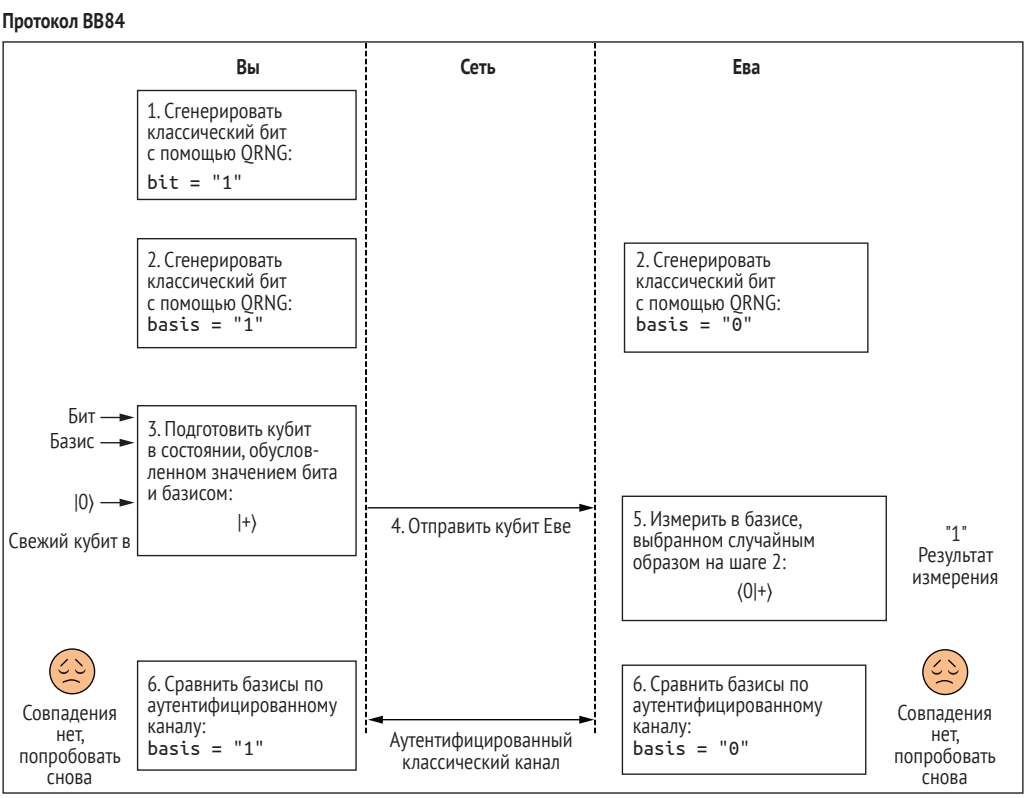


Рис. 3.2 Пошаговая диаграмма протокола BB84, варианта протокола QKD

Давайте сразу приступим к делу и посмотрим на то, как работает QKD! Для наших целей предположим, что мы и Ева являемся двумя людьми из предыдущей главы, которые хотят обменяться ключом, чтобы иметь возможность отправлять секретные сообщения 😍. Сценарий выглядит следующим образом:

> *Мы хотим отправить секретное сообщение своему другу. Используя квантовый генератор случайных чисел из главы 2, протокол QKD BB84 и шифрование на основе одноразового (шифровального) блокнота, мы хотим сконструировать программу для отправки сообщений, которые могут быть доказуемо безопасными.*

Мы можем визуализировать этот сценарий как своего рода пошаговую диаграмму, как показано на рис. 3.3.

Обратите внимание, что ключ, который нам нужно отправить, представляет собой строку классических битов. Каким образом можно использовать кубиты для отправки этих классических битов? Мы начнем с понимания того, как кодировать классическую информацию в кубитах, а затем усвоим конкретные шаги протокола BB84. В следующем разделе

мы рассмотрим новую квантовую операцию, которая поможет нам кодировать классические биты с кубитами.

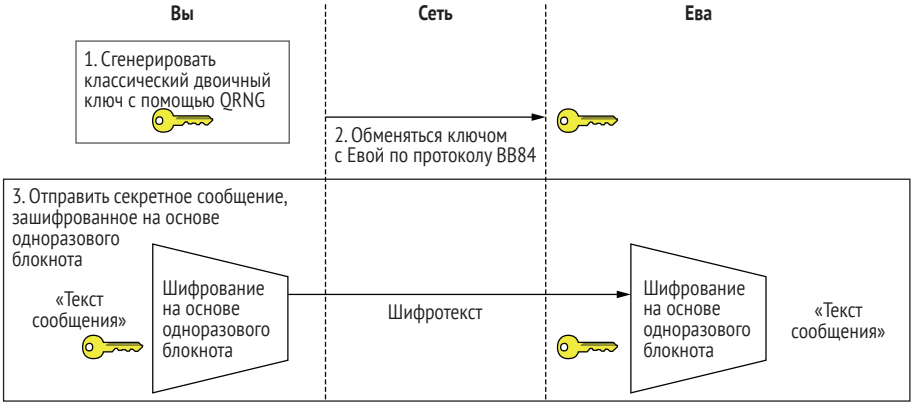


Рис. 3.3 Сценарий этой главы: отправка секретного сообщения Еве с помощью BB84 и шифрования на основе одноразового блокнота. Сначала мы должны обменяться секретным ключом с Евой, чтобы использовать его для шифрования сообщения, которое мы хотим отправить. Мы можем использовать кубиты и суперпозиционные состояния, которые будут помогать на шаге обмена ключами!

3.1.1 Квантовые операции NOT

Если у нас есть некоторая классическая информация, скажем один двоичный бит, то каким образом можно закодировать ее с квантовым ресурсом, таким как кубит? Рассмотрим следующий ниже алгоритм отправки строки случайных классических битов, закодированных в кубитах.

1. Использовать квантовый генератор случайных чисел для генерирования случайного бита ключа для отправки.
2. Начать с кубита в состоянии $|0\rangle$, а затем подготовить его в состоянии, которое представляет это битовое значение из шага 1. Здесь использовать $|0\rangle$, если классический бит был равен 0, и $|1\rangle$, если классический бит был равен 1.
3. Этот подготовленный кубит отправляется Еве, которая затем его измеряет и записывает значение классического бита.
4. Повторять шаги 1–3 до тех пор, пока у нас с Евой не будет столько ключей, сколько мы хотим (обычно это диктуется криптографическим протоколом, который мы хотим использовать после этого).

На рис. 3.4 показана пошаговая диаграмма указанного алгоритма.

Теперь, чтобы переключить кубит с $|0\rangle$ на $|1\rangle$, нам в нашем наборе инструментов понадобится еще одна квантовая операция. На шаге 2 мы можем использовать операцию *квантового* NOT – похожую на классическую операцию NOT, – которая поворачивает кубит из $|0\rangle$ в $|1\rangle$ (см. рис. 2.9).

Мы называем эту операцию квантового NOT операцией x.

ОПРЕДЕЛЕНИЕ Операция x, или *квантового NOT*, переводит кубит из состояния |0⟩ в состояние |1⟩ и наоборот.

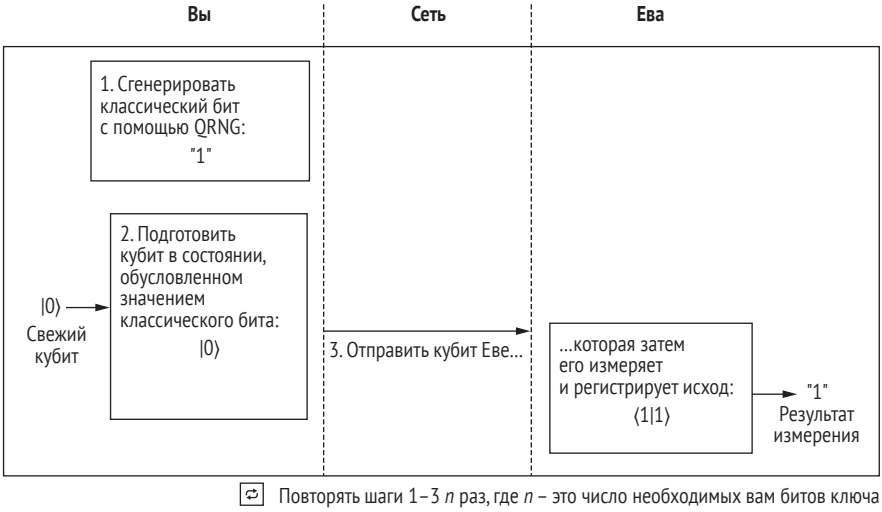


Рис. 3.4 Визуализация алгоритма отправки строки классических битов с кубитами. Мы начинаем с использования нашего QRNG для генерирования значения классического бита, кодируем его на новом кубите, а затем отправляем его Еве. Затем она может его измерить и зарегистрировать классический результат измерения

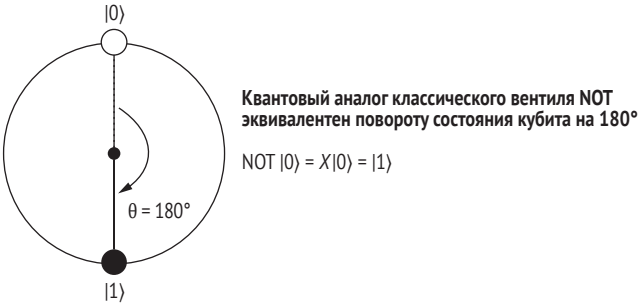


Рис. 3.5 Визуализация квантового эквивалента операции NOT, работающей с кубитом в состоянии |0⟩, оставляя кубит в состоянии |1⟩

Шаг 2 можно переписать следующим образом:

1 Если наш классический бит из шага 1 был равен 0, то ничего не делать. Если он был равен 1, то применить операцию квантового NOT (она же операция x) к нашему кубиту.

Этот алгоритм работает в 100 % случаев, потому что при измерении Евой получаемого ею кубита состояния |0⟩ и |1⟩ можно идеально различить с помощью измерения по оси *Z*. Может показаться, что мы с Евой проделали всю эту большую работу только для того, чтобы поделиться

несколькими классическими случайными битами, но мы увидим, как добавление в этот базовый протокол нескольких форм квантового поведения сделает его полезнее! Давайте посмотрим, как это можно было бы имплементировать в исходном коде.

Листинг 3.1 qkd.py: обмен классическими битами через кубиты

```
def prepare_classical_message(bit: bool, q: Qubit) -> None:          ❶
    if bit:
        q.x()                                                        ❷

def eve_measure(q: Qubit) -> bool:
    return q.measure()                                               ❸

def send_classical_bit(device: QuantumDevice, bit: bool) -> None:
    with device.using_qubit() as q:
       prepare_classical_message(bit, q)
       result = eve_measure(q)
       q.reset()
    assert result == bit                                             ❹
```

❶ В целях подготовки нашего кубита с классическим битом, который мы хотим отправить, нам нужно иметь на входе значение бита и кубит. Эта функция ничего не возвращает, потому что последствия операций, которые мы применяем к нашему кубиту, отслеживаются в однокубитовом симуляторе.

❷ Если мы отправляем 1, то мы можем использовать NOT-операцию x для подготовки q в состоянии $|1\rangle$, потому что операция x будет поворачивать $|0\rangle$ в $|1\rangle$ и наоборот.

❸ Кажется, что глупо выносить измерение в другую функцию, учитывая, что она является однострочной. Но в будущем мы изменим метод, которым Ева измеряет кубит, так что такая конфигурация является полезной.

❹ Мы можем проверить, что измерение q дает тот же классический бит, который мы отправили.

Симулятор, который мы написали в предыдущей главе, имеет *почти* все, что нам нужно для имплементирования всего описанного выше. Нам просто нужно добавить инструкцию, соответствующую операции x. Инструкция *x* может быть представлена матрицей *X*, подобно тому, как мы представили инструкцию h с использованием матрицы *H*. Мы можем записать матрицу *X* точно так же, как записали *H* в главе 2, следующим образом:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Упражнение 3.1: таблицы истинности и матрицы

В главе 2 мы увидели, что унитарные матрицы играют ту же роль в квантовых вычислениях, что и *таблицы истинности* в классических вычислениях. Мы можем использовать этот факт, чтобы выяснить, как должна выглядеть матрица *X*, чтобы представлять операцию квантового NOT, x. Давайте начнем

с составления таблицы того, что матрица X должна делать с каждым входным состоянием, чтобы представлять, что делает инструкция x.

Вход	Выход
$\|0\rangle$	$\|1\rangle$
$\|1\rangle$	$\|0\rangle$

Эта таблица говорит нам о том, что если мы умножим матрицу X на вектор $|0\rangle$, то нам нужно получить $|1\rangle$, и аналогичным образом, что $X|1\rangle = |0\rangle$.

С помощью пакета NumPy либо вручную проверьте, что матрица

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

соответствует тому, что мы имеем в приведенной выше таблице истинности.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

Давайте продолжим и добавим функциональность, необходимую нашему симулятору для выполнения листинга 3.1. Мы будем работать с симулятором, который мы написали в предыдущей главе, но если вам нужно освежить свои знания, то вы можете найти исходный код в репозитории GitHub этой книги: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Прежде всего нам нужно обновить интерфейс квантового устройства, добавив новый метод, которым наш кубит должен обладать.

Листинг 3.2 interface.py: добавление x в интерфейс кубита

```
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass

    @abstractmethod
    def x(self): pass                    ❶

    @abstractmethod
    def measure(self) -> bool: pass

    @abstractmethod
    def reset(self): pass
```

❶ Мы можем смоделировать, имплементировав операции квантового NOT по образцу операции h из главы 1.

Теперь, когда наш интерфейс кубита знает, что нам нужна имплементация операции x, давайте добавим эту имплементацию!

Листинг 3.3 simulator.py: добавление x в симулятор кубита

```
KET_0 = np.array([
    [1],
    [0]
], dtype=complex)
H = np.array([
    [1,  1],
    [1, -1]
], dtype=complex) / np.sqrt(2)
X = np.array([                                  ❶
    [0, 1],
    [1, 0]
], dtype=complex) / np.sqrt(2)

class SimulatedQubit(Qubit):
    def __init__(self):
        self.reset()

    def h(self):
        self.state = H @ self.state

    def x(self):                                ❷
        self.state = X @ self.state

    def measure(self) -> bool:
    pr0 = np.abs(self.state[0, 0]) ** 2
    sample = np.random.random() <= pr0
    return bool(0 if sample else 1)

def reset(self):
    self.state = KET_0.copy()
```

❶ Добавляет переменную X для хранения матрицы *X*, которая нам нужна для представления операции x.

❷ Так же, как и функция h, мы хотим имплементировать квантовую операцию x, применив матрицу, хранящуюся в X, к вектору состояния.

3.1.2 Обмен классических битов с кубитами

Великолепно! Давайте попробуем применить наш обновленный Python'овский симулятор кубита, чтобы поделиться секретным классическим битом с кубитом.

ПРИМЕЧАНИЕ В исходном коде, который мы напишем в этой главе, кубиты, которыми мы делимся с Евой, обитают в одном и том же симулируемом устройстве. Это может сделать неловкой мысль об *отправке* кубитов друг другу, если мы оба используем одно и то же устройство! В реальности наши устройства будут ис-

пользовать в качестве кубитов *фотоны* (отдельные частицы света), и их действительно легко отправлять по оптическим волокнам или по воздуху с помощью телескопов.

Все это еще не совсем то же самое, что и протокол квантового распределения ключей, но оно служит хорошей основой для типов функций и шагов, которые имеет наш целевой протокол QKD.

Откройте сеанс IPython, в котором мы держим исходный код симулятора, запустив ipython в терминале. После импортирования файлов Python создайте экземпляр однокубитового симулятора и сгенерируйте случайный бит для отправки в качестве классического бита. (Хорошо, что у нас есть квантовый генератор случайных чисел!) Используя свежий кубит, подготовьте его на основе значения классического бита, которое мы хотим отправить Еве. Затем Ева измеряет кубит, и мы увидим, имеем ли мы оба одинаковое значение классического бита.

Листинг 3.4 Отправка классических битов с помощью однокубитового симулятора

```
>>> qrng_simulator = SingleQubitSimulator()          ❶

>>> key_bit = int(qrng(qrng_simulator))              ❷

>>> qkd_simulator = SingleQubitSimulator()           ❸

>>> with qkd_simulator.using_qubit() as q:
...     prepare_classical_message(key_bit, q)        ❹
...     print(f"Вы подготовили классический бит ключа: {key_bit}")
...     eve_measurement = int(eve_measure(q))        ❺
...     print(f"Ева измерила классический бит ключа: {eve_measurement}")
...

Вы подготовили классический бит ключа: 1
Ева измерила классический бит ключа: 1
```

❶ Здесь мы будем использовать новый экземпляр симулятора кубита для обмена ключами, но, строго говоря, мы не обязаны это делать. В главе 4 мы рассмотрим тему расширения симулятора для работы с несколькими кубитами.
❷ Нам нужен симулированный кубит для использования в QRNG.
❸ Повторно используя функцию qrng, которую мы написали в главе 2, мы можем сгенерировать случайный классический бит для использования в качестве ключа.
❹ Мы кодируем классический бит в кубите, предоставленном экземпляром qkd_simulator. Если классический бит был равен 0, то мы ничего с экземпляром qkd_simulator не делаем; и если классический бит был равен 1, то мы используем метод x, чтобы перевести кубит в состояние |1⟩.
❺ Ева измеряет кубит из экземпляра qkd_simulator и затем сохраняет значение бита как eve_measurement.

Наш пример обмена секретами с кубитами должен быть детерминированным, т. е. всякий раз, когда мы готовим и отправляем бит, Ева будет правильно измерять одинаковое значение. Насколько это безопасно? Если вы подозреваете, что это небезопасно, то вы определенно на что-

то наткнулись. В следующем разделе мы обсудим безопасность нашей прототипной схемы обмена секретами и рассмотрим способы ее усовершенствования.

3.2 Сказка о двух базисах

Теперь у нас с Евой есть способ отправлять классические биты, используя кубиты, но что произойдет, если злоумышленник завладеет этим кубитом? Он мог бы использовать инструкцию measure, чтобы получить те же классические данные, что и Ева. Это огромная проблема, и это разумно заставляет нас задаться вопросом, почему кто-то вообще использует кубиты для обмена ключами.

К счастью, квантовая механика предлагает способ сделать этот обмен более безопасным! Какие модификации мы могли бы внести в наш протокол? Например, мы могли бы представить классическое сообщение «0» кубитом в состоянии $|+\rangle$ и сообщение «1» кубитом в состоянии $|-\rangle$.

Листинг 3.5 qkd.py: кодирование сообщения состояниями $|+\rangle$ / $|-\rangle$

```
def prepare_classical_message_plusminus(bit: bool, q: Qubit) -> None:
    if bit:
        q.x()
    q.h()                    ❶

def eve_measure_plusminus(q: Qubit) -> bool:
    q.h()                    ❷
    return q.measure()

def send_classical_bit_plusminus(device: QuantumDevice, bit: bool) -> None:
    with device.using_qubit() as q:
        prepare_classical_message_plusminus(bit, q)
        result = eve_measure_plusminus(q)
        assert result == bit
```

❶ Все перед строкой prepare_classical_message_plusminus является таким же, как и раньше с prepare_classical_message. Применение вентиля Адамара в этой точке поворачивает состояния $|0\rangle/|1\rangle$ в состояния $|+\rangle/|-\rangle$.

❷ Использует операцию h для поворота наших состояний $|+\rangle/|-\rangle$ обратно в состояния $|0\rangle/|1\rangle$, потому что наша операция measure определена только для правильного измерения состояний $|0\rangle/|1\rangle$.

ДЛЯ СПРАВКИ На измерение в листинге 3.5 можно взглянуть и по-другому, как на поворачивание измерения, чтобы соответствовать базису, в котором мы в настоящий момент работаем ($|+\rangle/|-\rangle$). Все дело в том, под каким углом зрения смотреть!

Теперь у нас есть два разных способа отправки кубитов, которые мы и Ева могли бы использовать при отправке кубитов (краткое описание см. в табл. 3.1). Мы называем эти два разных способа отправки сообщений *базисами*, и каждый из них содержит два полностью различимых

(ортогональных) состояния. Это похоже на приложение С к книге, где мы рассматриваем направления на карте местности (например, север и запад), которые определяют удобный *базис* для описания направлений.

Таблица 3.1 Разные классические сообщения, которые мы хотим отправить, и то, как их кодировать в базисах Z и X

	Сообщение «0»	Сообщение «1»
«0»- (или *Z*-базис)	$\|0\rangle$	$\|1\rangle = X\|0\rangle$
«1»- (или *X*-базис)	$\|+\rangle = H\|0\rangle$	$\|-\rangle = H\|1\rangle = HX\|0\rangle$

ДЛЯ СПРАВКИ Освежить свои знания по базисам можно, обратившись к приложению С книги.

Мы использовали состояния $|0\rangle$ и $|1\rangle$ в качестве одного базиса (именуемого *Z*-базисом) и $|+\rangle$ и $|-\rangle$ в качестве другого (именуемого *X*-базисом). Названия этих базисов относятся к оси, вдоль которой мы можем идеально различать состояния (см. рис. 3.6).

Рис. 3.6 Теперь, в дополнение к использованию *Z*-базиса для кодирования классического бита на кубите, мы можем использовать *X*-базис

ПРИМЕЧАНИЕ В квантовых вычислениях никогда не существует по-настоящему *правильного* базиса, поскольку существуют удобные базисы, которые мы решаем использовать по соглашению.

Если ни мы, ни Ева не знаем, какой способ отправки мы используем для конкретного бита, то у нас обоих проблемы. Что произойдет, если мы смешаем отправку наших сообщений в *Z*-базисе и *X*-базисе? Хорошая новость состоит в том, что мы можем использовать наш симулятор, чтобы попробовать и посмотреть, что произойдет.

Листинг 3.6 Обмен битами, но без использования одинакового базиса

```
def prepare_classical_message(bit: bool, q: Qubit) -> None:      ❶
    if bit:
        q.x()
```

```
def eve_measure_plusminus(q: Qubit) -> bool:
    q.h()                                                    ❷
    return q.measure()

def prepare_classical_message(bit: bool, q: Qubit) -> None:  ❶
    if bit:
        q.x()

def eve_measure_plusminus(q: Qubit) -> bool:
    q.h()                                                    ❷
    return q.measure()

def send_classical_bit_wrong_basis(device: QuantumDevice, bit: bool) -> None:
    with device.using_qubit() as q:
        prepare_classical_message(bit, q)
        result = eve_measure_plusminus(q)
        assert result == bit, "Две стороны не имеют одинаковое битовое значение" ❸
```

❶ Использует метод, который мы встречали ранее, чтобы подготовить кубит в *Z*-базисе с помощью метода h.
❷ Ева измеряет в *X*-базисе, потому что перед измерением она выполняет вентиль Адамара на своем кубите.
❸ Функция ничего не возвращает, поэтому если мы и Ева в конечном итоге получим биты ключа, которые не совпадают, то это вызовет ошибку.

Выполнив приведенный выше исходный код, мы видим, что если мы отправим в *Z*-базисе и Ева измеряет в *X*-базисе, то в конце мы, возможно, не получим совпадающие классические биты.

Листинг 3.7 Отправка в *Z*-базисе; измерение в *X*-базисе

```
>>> qsim = SingleQubitSimulator()
>>> send_classical_bit_wrong_basis(qsim, 0)     ❶
AssertionError: Две стороны не имеют одинаковое битовое значение
```

❶ Мы выбрали битовое значение равным 0. Вам, возможно, придется выполнить эту строку кода несколько раз, прежде чем вы получите ошибку.

Вы можете попробовать это экспериментально. Вы обнаружите, что примерно в половине случаев получаете ошибку `AssertionError` (сбой обмена ключами). Почему это происходит? Начнем с того, что Ева измеряет в *X*-базисе, поэтому она может точно различать только $|+\rangle$ и $|-\rangle$. Что она будет измерять, если ей не будет дано состояние, идеально различимое для ее базиса (как в этом случае, когда ей дается $|0\rangle$)? Мы можем записать состояние $|0\rangle$ в *X*-базисе как

$$|0\rangle = (|+\rangle + |-\rangle)/\sqrt{2}.$$

Напомним, что в главе 2 мы определили $|+\rangle$ аналогичным образом, сложив $|0\rangle$ и $|1\rangle$. Состояние $|+\rangle$ также называется *суперпозицией* состояний $|0\rangle$ и $|1\rangle$.

ПРИМЕЧАНИЕ Всякий раз, когда состояние может быть записано в виде подобного рода линейной комбинации состояний, оно считается суперпозицией состояний, которые складываются вместе.

Упражнение 3.2: проверка того, что |0⟩ является суперпозицией |+⟩ и |−⟩

Попробуйте использовать то, что вы узнали о векторах в предыдущей главе, чтобы убедиться, что $|0\rangle = (|+\rangle + |-\rangle)/\sqrt{2}$, вручную либо с помощью Python. *Подсказка*: вспомните, что $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ и что $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$.

Теперь что касается вычисления фактического измерения с помощью правила Борна из главы 2. Напомним, что мы можем вычислить вероятность результата измерения, измерив конкретное состояние следующим выражением:

$$\Pr(\text{измерение}|\text{состояние}) = |\langle \text{измерение}|\text{состояние}\rangle|^2.$$

Записывая измерение состояния |0⟩ в X-базисе, мы видим, что будем получать 0 (или |+⟩) в половине случаев и 1 (или |−⟩) в другой половине:

$$\Pr(\langle +|0\rangle) = |\langle +|0\rangle|^2 = ((\langle +|+\rangle + \langle +|-\rangle)/\sqrt{2})^2 = (1+0)^2/2 = 1/2.$$

Упражнение 3.3: измерение кубитов в разных базисах

Использование предыдущего примера в качестве руководства:

1. Вычислите вероятность получить результат измерения |−⟩ при измерении состояния |0⟩ в направлении |−⟩.
2. Также вычислите вероятность получить результат измерения |−⟩ с входным состоянием |1⟩.

Это говорит нам о том, что если Ева не знает правильного базиса, в котором нужно измерять, то измерения, которые она делает, не отличаются от случайной догадки. Это обусловлено тем, что в неправильном базисе кубит находится в суперпозиции двух состояний, определяющих базис. Одним из «ключей» к тому, как работает QKD, является то, что без правильной дополнительной информации (базиса, в котором кубит закодирован) любое измерение кубита, в принципе, бесполезно. В целях обеспечения нашей безопасности мы должны сделать так, чтобы злоумышленнику было трудно усвоить эту дополнительную информацию, чтобы знать правильный базис, в котором нужно измерять. Протокол QKD, который мы рассмотрим далее, имеет решение для этой процедуры и доказательство (здесь оно выходит за рамки изложения), которое описывает вероятность того, что у злоумышленника есть какая-либо информация о ключе!

3.3 Квантовое распределение ключей: BB84

Теперь мы увидели, как обмениваться ключами в двух разных базисах и что произойдет, если мы и Ева не будем использовать один и тот же базис. Опять же, вы, возможно, спросите, почему мы используем этот подход для обеспечения большей безопасности общего доступа к нашим секретным ключам. Существует большое *разнообразие* различных протоколов QKD, каждый из которых имеет конкретные преимущества и варианты использования (в отличие от классов персонажей в ролевой игре). Наиболее распространенный протокол QKD именуется BB84, названный по инициалам двух авторов надлежащего криптокодирования и года публикации протокола (Bennet and Brassard 1984).

Протокол BB84 очень похож на то, что мы разработали до этого для обмена ключами, но имеет критическую разницу в том, как мы и Ева выбираем наши базисы. В BB84 обе стороны выбирают свой базис случайно (и независимо), что означает, что в конечном итоге они будут использовать один и тот же базис в 50 % случаев. На рис. 3.7 показаны шаги протокола BB84.

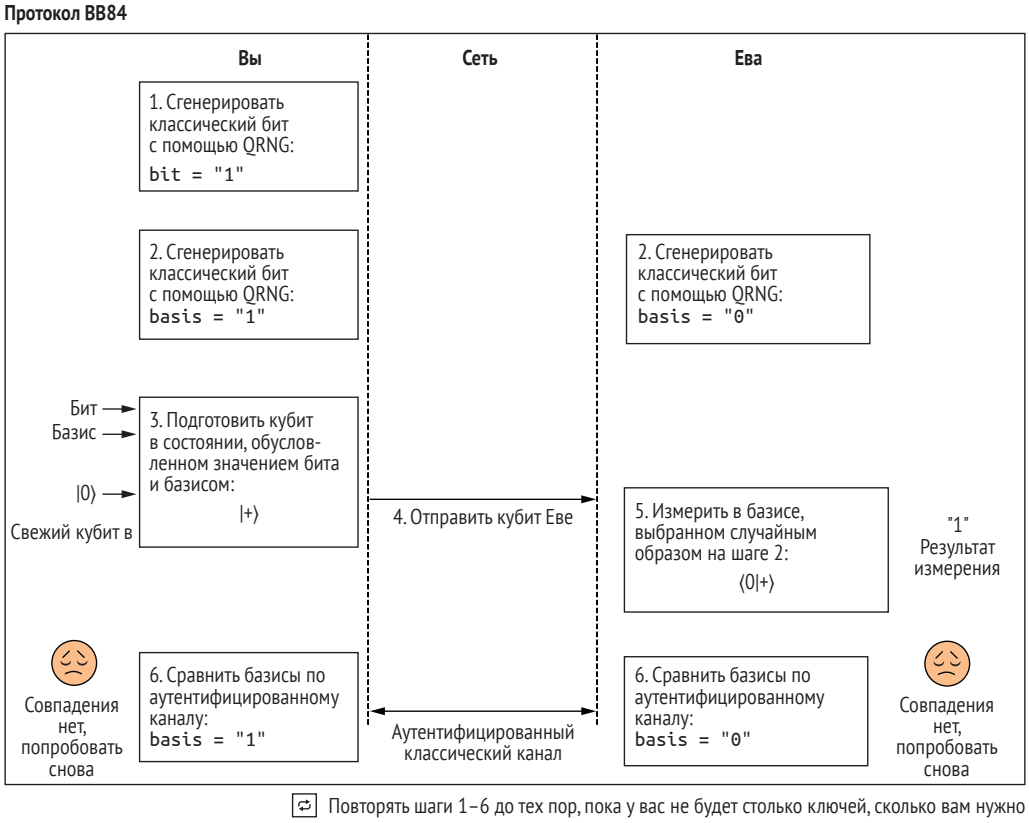


Рис. 3.7 Шаги в протоколе BB84, конкретной версии протокола QKD

Как следствие случайного выбора базисов мы и Ева также должны общаться по аутентифицированным классическим каналам (таким как интернет), чтобы каждый из нас брал наш ключ и преобразовывал его в ключ, который, по нашему мнению, идентичен ключу, имеющемуся у партнера. Это обусловлено тем, что мы имеем дело с реальной жизнью, и при обмене кубитов не исключена возможность, что окружающая среда и сторонние лица будут манипулировать состоянием кубита или его модифицировать.

Расширение ключа

В нашем описании классического канала связи, который мы с Евой используем, мы упустили одну деталь: он должен быть *аутентифицирован*. То есть при отправке классических сообщений Еве в рамках выполнения BB84 вполне нормально, если кто-то еще сможет их прочитать, но нам необходимо убедиться, что мы действительно говорим с Евой. В целях доказательства, что кто-то написал и отправил конкретное сообщение, нам на самом деле *уже* нужна какая-то форма общего секрета, которую мы можем использовать для проверки личности другого человека. Значит, у нас уже должен быть общий секрет с другим человеком в BB84. Этот секрет может быть меньше, чем сообщение, которое мы пытаемся отправить, поэтому BB84 технически является, скорее, протоколом *расширения ключа*.

Шаги протокола BB84 таковы:

1. Выбрать случайное однобитовое сообщение для отправки путем отбора из нашего QRNG-генератора.
2. Мы и Ева выбираем случайный базис с помощью нашего соответствующего QRNG-генератора (между ними нет связи).
3. Подготовить кубит на случайно отобранном базисе, представляющем наше случайно отобранное сообщение (см. табл. 3.2).
4. Отправить подготовленный кубит по квантовому каналу Еве.
5. Ева измеряет кубит при его поступлении, выполняя измерение в ее случайно отобранном базисе и регистрируя результат в виде классического бита.
6. Пообщаться с Евой по аутентифицированному классическому каналу и поделиться тем, какие базисы мы использовали для подготовки и измерения кубита. Если они совпадают, то сохранить бит и добавить его в ключ. Повторять шаги 1–6 до тех пор, пока у нас не будет столько ключей, сколько нам нужно.

Таблица 3.2 Какое состояние мы должны отправить для каждого случайного сообщения и варианта базиса

	Сообщение «0»	Сообщение «1»
«0»- (или *Z*-базис)	$\|0\rangle$	$\|1\rangle = X\|0\rangle$
«1»- (или *X*-базис)	$\|+\rangle = H\|0\rangle$	$\|-\rangle = H\|1\rangle = HX\|0\rangle$

Безошибочный мир

Поскольку мы симулируем протокол BB84, мы знаем, что кубит, который получает Ева, будет точно таким же, как и тот, который отправили мы. BB84 реалистичнее будет выполняться в пакетах, где сначала происходит обмен *n* кубитами, а затем раунд обмена базисными значениями (происходит исправление ошибок). В конце мы должны сжать ключ еще больше с помощью алгоритмов усиления приватности, чтобы учесть тот факт, что злоумышленник мог получить частичную информацию из обнаруженных нами ошибок. В нашей имплементации BB84 эти шаги мы опустили, чтобы все было просто, но в реальном мире они имеют решающее значение для обеспечения безопасности 😊.

Давайте перейдем к имплементации протокола BB84 QKD на Python! Мы начнем с написания функции, которая будет выполнять протокол BB84 (при условии передачи без потерь) для однобитовой передачи. Это не гарантирует, что мы получим один бит ключа из этого выполнения. Однако если мы с Евой выберем разные базисы, то этот обмен придется отбросить.

Прежде всего полезно настроить несколько функций, которые упростят написание полного протокола BB84. Нам с Евой нужно делать такие вещи, как отбор случайных битов, подготовка и измерение кубита сообщения, выделенные здесь для ясности.

Листинг 3.8 bb84.py: вспомогательные функции перед обменом ключами

```
def sample_random_bit(device: QuantumDevice) -> bool:
    with device.using_qubit() as q:
        q.h()
        result = q.measure()
        q.reset()                                    (1)
    return result

def prepare_message_qubit(message: bool, basis: bool, q: Qubit) -> None:    (2)
    if message:
        q.x()
    if basis:
        q.h()

def measure_message_qubit(basis: bool, q: Qubit) -> bool:
    if basis:
        q.h()
    result = q.measure()
    q.reset()                                        (3)
    return result

def convert_to_hex(bits: List[bool]) -> str:                                (4)
    return hex(int(
```

```
        "".join(["1" if bit else "0" for bit in bits]),
        2
    ))
```

❶ Функция sample_random_bit почти такая же, как и функция qrng, которую мы встречали ранее, за исключением того, что здесь мы будем сбрасывать кубит после измерения, поскольку мы знаем, что хотим иметь возможность использовать его более одного раза.
❷ Кубит кодируется значением бита ключа в случайно выбранном базисе.
❸ Аналогично функции sample_random_bit, после того как Ева измерит кубит сообщения, она должна его сбросить, потому что мы будем использовать его в симуляторе повторно в целях следующего обмена.
❹ В целях упрощения вывода на экран длинных двоичных ключей вспомогательная функция конвертирует их представление в более короткий шестнадцатеричный строковый литерал.

Листинг 3.9 bb84.py: протокол BB84 для отправки классического бита

```
def send_single_bit_with_bb84(
    your_device: QuantumDevice,
    eve_device: QuantumDevice
    ) -> tuple:

    [your_message, your_basis] = [
        sample_random_bit(your_device) for _ in range(2)          ❶
    ]

    eve_basis = sample_random_bit(eve_device)                     ❷

    with your_device.using_qubit() as q:
        prepare_message_qubit(your_message, your_basis, q)        ❸

        # ОТПРАВКА КУБИТА...                                      ❹

        eve_result = measure_message_qubit(eve_basis, q)          ❺

    return ((your_message, your_basis), (eve_result, eve_basis))  ❻
```

❶ Мы можем выбрать битовое значение и базис случайно, используя модифицированный ранее QRNG-генератор: здесь функция sample_random_bit.
❷ Еве нужно случайно выбрать базис со своим собственным кубитом, поэтому она использует отдельный экземпляр класса QuantumDevice.
❸ После всей подготовительной работы наш кубит готовится к отправке Еве.
❹ Поскольку все вычисления происходят внутри симулятора на нашем компьютере, ничего не нужно делать, чтобы «отправить» кубит от нас к Еве.
❺ Теперь у Евы есть наш кубит, и она измеряет его в случайно выбранном базисе, который она выбрала ранее.
❻ Возвращает значения битов ключа и базисы, которые мы и Ева имели бы в конце этого раунда

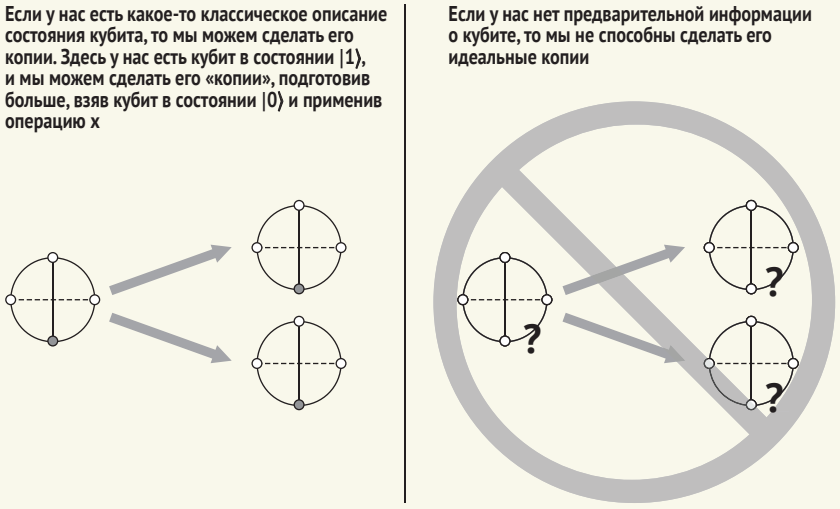
Кубиты и запрет клонирования

Из того, что мы видели до сих пор, похоже, что наш злоумышленник может сжульничать, подслушав кубиты в квантовом канале и сделав копии. Вот как это происходит: злоумышленник (здесь по имени Боб) сначала должен был бы (без обнаружения):

1. копировать кубиты по мере их отправки между нами и Евой и затем их сохранять;
2. пока мы с Евой заканчиваем классическую часть протокола, слушать базисы, которые мы оба объявляем, и следить за теми, которые мы оба выбрали одинаково;
3. для кубитов, соответствующих битам, в которых мы и Ева использовали один и тот же базис, также измерять копии кубитов в одном и том же базисе.

Та-да! У нас, и у Евы, *и* у Боба будет один и тот же ключ! Если вы думаете, что это кажется проблемой, то вы совершенно правы. Но не волнуйтесь, решение лежит в квантовой механике. Оказывается, проблема с планом Боба заключается в шаге 1, где ему нужно делать *идентичные* копии кубитов, которыми мы с Евой обмениваемся. Хорошая новость в том, что создание точной копии кубита, не зная заранее, что он из себя представляет, запрещено квантовой механикой. Правило, согласно которому кубиты не могут быть скопированы идентично без предварительного знания состояния, называется *теоремой о запрете клонирования* и формулируется следующим образом:

> *Никакая квантовая операция не может идеально скопировать состояние произвольного кубита на другой кубит.*



Визуализация теоремы о запрете клонирования

Мы сможем выполнить простое доказательство этой теоремы в следующей главе, как только научимся описывать состояние более чем одного кубита ☺.

В качестве альтернативного образа мыслей о теореме о запрете клонирования, если бы Боб мог измерять кубит, не нарушая его, он мог бы обходиться без копии кубитов, которые он перехватывает. Это невозможно, потому что, как только мы измеряем кубит, он «коллапсирует» или изменяется так, что Ева обнаружит дополнительный шум в результатах измерений, которые она собирает со своих кубитов. Таким образом, измерение при передаче – это не то, что Боб может делать, не будучи обнаруженным, и поэтому его подслушивание обречено на неудачу.

Обмена одним классическим битом ключа будет недостаточно для отправки целого ключа, поэтому теперь нам нужно применить более ранний технический прием для отправки нескольких битов.

Листинг 3.10 bb84.py: протокол BB84 для обмена ключом с Евой

```
def simulate_bb84(n_bits: int) -> tuple:
    your_device = SingleQubitSimulator()
    eve_device = SingleQubitSimulator()

    key = []
    n_rounds = 0

    while len(key) < n_bits:
        n_rounds += 1
        ((your_message, your_basis), (eve_result, eve_basis)) =
            ➥ send_single_bit_with_bb84(your_device, eve_device)

        if your_basis == eve_basis:                    ❶
            assert your_message == eve_result
            key.append(your_message)

    print(f"Потребовалось {n_rounds} раундов, чтобы сгенерировать
        ➥ {n_bits}-битовый ключ.")

    return key
```

❶ **В этой точке мы с Евой можем публично объявить базисы, которые каждый из нас использовал для измерения этого бита. Если все сработало правильно, то наши результаты должны совпадать всякий раз, когда наши базисы совпадают. Мы проверяем это здесь с помощью инструкции подтверждения истинности assert.**

Ключ теперь в сумке, так что мы можем перейти к использованию ключа и алгоритма шифрования на основе одноразового блокнота для отправки секретного сообщения!

3.4 Использование секретного ключа для отправки секретных сообщений

Мы с Евой выяснили, как использовать протокол BB84 для совместного использования случайного двоичного ключа, сгенерированного QRNG-генератором. Последний шаг состоит в использовании этого ключа, чтобы поделиться секретным сообщением с Евой. Мы с Евой ранее решили, что самый лучший протокол шифрования для использования – это одноразовый блокнот для отправки наших секретных сообщений. Оказывается, он является одним из самых безопасных протоколов шифрования, и, учитывая, что мы обмениваемся ключами одним из самых безопасных способов, имеет смысл поддержать этот стандарт!

Например, чтобы сказать Еве, что нам нравится Python, мы хотим отправить следующее сообщение: «💝🐍💻». Поскольку мы используем двоичный ключ, нам нужно конвертировать представление этого юникодного сообщения в двоичное, которое представляет собой следующий длинный список битов:

«1101100000111101 1101110010010110 1101100000111101
1101110000001101 1101100000111101 1101110010111011»

Это двоичное представление сообщения является *текстом сообщения*, и теперь мы хотим совместить его с ключом, чтобы получить шифротекст, который будет безопасно отправить по сети. Имея в распоряжении ключ из протокола BB84 (по меньшей мере, такой же длины, что и наше сообщение), нам нужно будет использовать схему шифрования на основе одноразового блокнота для кодирования сообщения. Мы встречали этот метод шифрования в главе 2; см. рис. 3.8 для быстрого освежения памяти.

В целях имплементации этой схемы нам нужно использовать классический побитовый XOR (оператор ^ в Python), чтобы совместить сообщение и ключ для создания шифротекста, который мы сможем безопасно отправить.

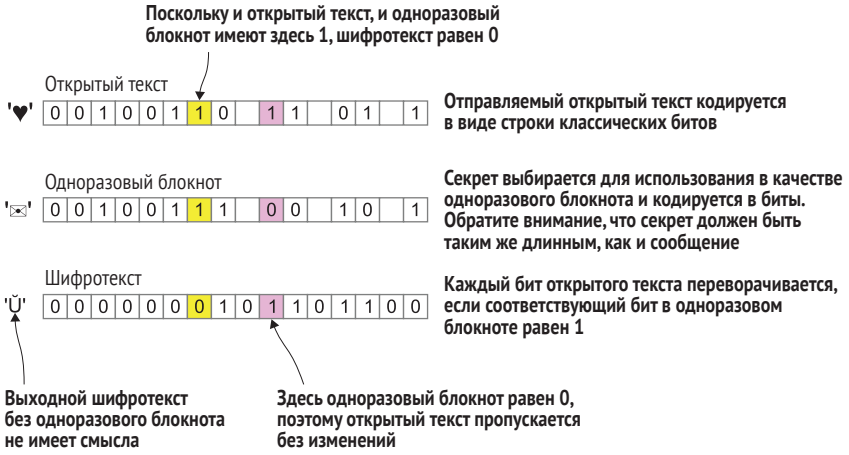


Рис. 3.8 Пример шифрования на основе одноразового блокнота, в котором для шифрования секретных сообщений используются случайные биты

Ева. В целях расшифровки нашего сообщения Ева выполнит ту же побитовую операцию XOR с шифротекстом и своим ключом (который должен быть таким же, как у вас). В результате она получит сообщение, потому что всякий раз, когда мы дважды соединяем битовую строку с другой, у нас остается исходная битовая строка. Вот как это будет выглядеть на Python.

Листинг 3.11 bb84.py: протокол BB84 для обмена ключом с Евой

```
def apply_one_time_pad(message: List[bool], key: List[bool]) -> List[bool]:
    return [
        message_bit ^ key_bit                                    ❶
        for (message_bit, key_bit) in zip(message, key)
    ]
```

❶ Оператор ^ является в Python побитовым XOR. Эта инструкция применяет один бит нашего ключа в качестве одноразового блокнота к тексту нашего сообщения.

Упражнение 3.4: шифрование на основе одноразового блокнота

Если бы у нас был шифротекст `10100101` и ключ `00100110`, то какое сообщение было бы изначально отправлено?

Давайте окончательно соберем все это вместе и поделимся сообщением («💌🐍💻») с Евой, выполнив файл bb84.py, который мы собрали.

Листинг 3.12 bb84.py: использование BB84 и шифрования на основе одноразового блокнота

```
if __name__ == "__main__":
    print("Генерирование 96-битового ключа путем симулирования BB84...")

    key = simulate_bb84(96)
    print(f"Получен ключ {convert_to_hex(key)}.")

    message = [
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        1, 0, 0, 1, 0, 1, 1, 0,
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        0, 0, 0, 0, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        1, 0, 1, 1, 1, 0, 1, 1
    ]
    print(f"Использование ключа для отправки секретного сообщения: {convert_to_
hex(message)}.")

    encrypted_message = apply_one_time_pad(message, key)
    print(f"Зашифрованное сообщение: {convert_to_hex(encrypted_message)}.")

    decrypted_message = apply_one_time_pad(encrypted_message, key)
    print(f"Ева расшифровала, получив: {convert_to_hex(decrypted_message)}.")
```

Листинг 3.13 Выполнение полного решения сценария этой главы

```
$ python bb84.py
Генерирование 96-битового ключа путем симулирования BB84...
Потребовалось 170 раундов, чтобы сгенерировать 96-битовый ключ.          (1)

Получен ключ:                                          0xb35e061b873f799c61ad8fad. (2)

Использование ключа для отправки секретного сообщения: 0xd83ddc96d83ddc0dd83ddcbb. (3)

Зашифрованное сообщение:                               0x6b63da8d5f02a591b9905316. (4)

Ева расшифровала, получив:                             0xd83ddc96d83ddc0dd83ddcbb. (5)
```

(1) Поскольку наш базис и базис Евы согласуются примерно в половине случаев, для каждого бита ключа, который мы хотим сгенерировать, потребуется около двух раундов BB84.
(2) Точный ключ, который мы генерируем, будет отличаться всякий раз, когда мы запускаем симуляцию BB84, – в конце концов, это колоссальная часть смысла протокола!
(3) Сообщение, которое мы получаем, записывая каждую кодовую точку Юникода для «💔🪢💻».
(4) Когда мы совмещаем наше секретное сообщение с ключом, который мы получили ранее, используя ключ в качестве одноразового блокнота, наше сообщение скремблируется.
(5) Когда Ева использует тот же ключ, она получает наше изначальное секретное сообщение.

Квантовое распределение ключей является одной из наиболее важных побочных технологий квантовых вычислений и потенциально способно оказать огромное влияние на нашу инфраструктуру обеспечения безопасности. Хотя в настоящее время довольно легко настроить квантовое распределение ключей (QKD) для сторон, которые находятся относительно близко друг к другу (около 200 км или менее), существуют значительные проблемы с развертыванием глобальной системы QKD. Обычно физическая система, используемая в QKD, представляет собой фотон, при этом имеются трудности в отправке отдельных частиц света на большие расстояния без их потери.

Теперь, когда мы создали однокубитовый симулятор и запрограммировали несколько однокубитовых приложений, мы готовы поиграть с несколькими кубитами. В следующей главе мы возьмем построенный нами симулятор, добавим функции для симулирования состояний нескольких кубитов и поиграем в нелокальные игры с Евой 💔.

Резюме

- Квантовое распределение ключей – это протокол, который позволяет нам случайным образом генерировать совместные ключи, которые мы можем использовать для безопасного и приватного общения.
- При измерении кубитов мы можем делать это в разных базисах; если мы измеряем в том же базисе, в котором готовим наши кубиты, результаты будут детерминированными, тогда как если мы измеряем в разных базисах, то результаты будут случайными.

- Теорема о запрете клонирования гарантирует, что злоумышленники не смогут угадывать правильный базис для измерения, не вызвав сбоя протокола распределения ключей.
- После того как мы использовали QKD для совместного использования ключа, мы можем использовать ключ с классическим алгоритмом, именуемым одноразовым шифровальным блокнотом, для безопасной отправки данных.

4 Нелокальные игры: работа с несколькими кубитами

Эта глава охватывает следующие ниже темы:

- использование нелокальных игр для проверки того, что квантовая механика согласуется с тем, как работает Вселенная;
- симулирование подготовки состояния, операций и результатов измерений для нескольких кубитов;
- выяснение характеристик запутанных состояний.

В предыдущей главе мы использовали кубиты для безопасной связи с Евой, разведав возможные способы использования квантовых устройств в криптографии. Забавно работать, используя по одному кубиту за раз, но все же работать с бóльшим их числом… забавнее! В этой главе мы узнаем, как симулировать состояния нескольких кубитов и что значит для них быть *запутанными*. Мы снова будем играть в игры с Евой, но на этот раз нам понадобится судья!

4.1 Нелокальные игры

К этому моменту мы узнали, как однокубитовые устройства могут программироваться для выполнения полезных задач, таких как генерирование случайных чисел и квантовое распределение ключей. Однако

самые захватывающие вычислительные задачи требуют совместного использования многочисленных кубитов. В этой главе мы узнаем о нелокальных играх: подходе к валидированию наших квантово-механических описаний Вселенной с друзьями, используя многокубитовые системы.

4.1.1 Что такое нелокальные игры?

Мы все играли в игры того или иного рода, будь то спортивные, настольные, видео- или ролевые игры. Игры – это один из лучших способов разведать новые миры и протестировать наши пределы силы, выносливости и понимания. Оказывается, Ева любит играть в игры, и последнее зашифрованное сообщение от нее было таким:

> «*Привет, игрок! Я очень хочу поиграть в игру под названием CHSH. Это нелокальная игра, в которой мы играем в присутствии судьи. Я отправлю инструкции в следующем сообщении. КОНЕЦ СВЯЗИ*».

Предложенную Евой игру делает *нелокальной* тот факт, что во время игры игроки (к сожалению) не находятся в одном и том же месте. Игроки участвуют в игре, отправляя и получая сообщения в присутствии центрального судьи, но во время игры у них нет возможности поговорить. Что действительно здорово, так это то, что, играя в эту игру, мы можем показать, что классической физики просто недостаточно для описания результатов, которые мы получаем в этих играх с теми или иными стратегиями. Конкретная выигрышная стратегия, которую мы рассмотрим здесь, предусматривает, что перед началом игры игроки делятся парой кубитов. В этой главе мы подробно рассмотрим смысл запутывания двух кубитов, но давайте начнем с описания полных правил нашей нелокальной игры.

> **ПРИМЕЧАНИЕ** Судья, обслуживающий нелокальную игру, обеспечивает, чтобы игроки не общались, разделяя их настолько большим расстоянием, что свет от одного игрока не способен достичь другого до окончания игры.

4.1.2 Тестирование квантовой физики: игра CHSH

Нелокальная игра, в которую предложила играть Ева, называется *игрой CHSH*, показанной на рис. 4.1[1].

В игре CHSH участвуют два игрока и судья. Мы можем играть столько раундов игры, сколько захотим, и каждый раунд состоит из трех эта-

---

[1] Название CHSH происходит от инициалов исследователей, которые изначально создали эту игру: Clauser (Клаузер), Horne (Хорн), Shimony (Шимони) и Holt (Холт). Если вам интересно, то вы можете найти оригинал по адресу: https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.23.880.

пов. Как упоминала Ева в своем первом сообщении, после начала раунда игроки не могут общаться и должны принимать свои собственные (возможно, заранее запланированные) решения.

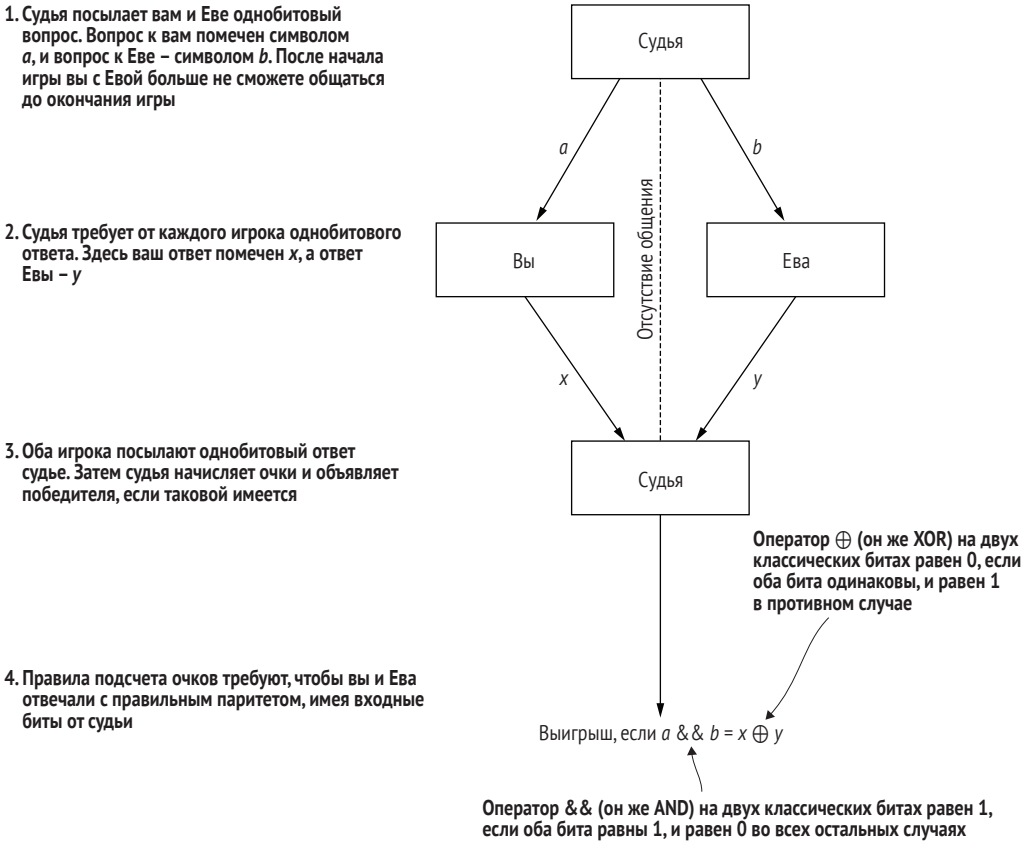


Рис. 4.1 Игра CHSH, нелокальная игра с двумя игроками и судьей. Судья задает каждому игроку вопрос в форме битового значения, а затем каждый игрок должен придумать, как ответить судье. Игроки выигрывают, если булево значение AND их ответов совпадает с классическим XOR вопросов судьи

Шаги для одного раунда игры CHSH таковы.

1. Судья начинает раунд, давая нам и Еве по одному классическому биту. Судья выбирает эти биты *независимо* и *равномерно случайным образом*, чтобы мы могли получать 0 либо 1, каждый с вероятностью 50 %, и то же самое для Евы. Это означает, что есть четыре возможных варианта, которыми судья может начать игру (наш бит, бит Евы): (0,0), (0,1), (1,0) или (1,1).
2. Мы и Ева должны каждый принять *самостоятельное* решение о том, какой классический бит отдать судье в качестве ответа.
3. Затем судья вычисляет паритет (XOR) наших и Евы ответов, данных в виде классических битов.

Как указано в табл. 4.1, в трех из четырех случаев мы и Ева должны ответить с четным паритетом (наши ответы должны быть одинаковыми), чтобы выиграть, в то время как в четвертом случае наши ответы должны быть разными. Эти правила, безусловно, *необычные*, но не так уж плохи по сравнению с некоторыми многодневными настольными играми.

Таблица 4.1 Условия выигрыша в игре CHSH

Наше входное значение	Входное значение Евы	Паритет ответа для выигрыша
0	0	Четный
0	1	Четный
1	0	Четный
1	1	Нечетный

Мы можем расширить табл. 4.1, чтобы получить все возможные исходы игры; см. табл. 4.2.

Таблица 4.2 Все возможные состояния игры CHSH с условиями выигрыша. Входные биты поступают от судьи, и оба игрока отвечают судье

Наше входное значение	Входное значение Евы	Наш ответ	Ответ Евы	Паритет	Выигрыш?
0	0	0	0	Четный	Да
0	0	0	1	Нечетный	Нет
0	0	1	0	Нечетный	Нет
0	0	1	1	Четный	Да
0	1	0	0	Четный	Да
0	1	0	1	Нечетный	Нет
0	1	1	0	Нечетный	Нет
0	1	1	1	Четный	Да
1	0	0	0	Четный	Да
1	0	0	1	Нечетный	Нет
1	0	1	0	Нечетный	Нет
1	0	1	1	Четный	Да
1	1	0	0	Четный	Нет
1	1	0	1	Нечетный	Да
1	1	1	0	Нечетный	Да
1	1	1	1	Четный	Нет

Давайте посмотрим на исходный код Python с симуляцией этой игры. Поскольку наши ответы и ответы Евы судье могут зависеть от сообщения, которое дает нам судья, мы можем представить действия каждого игрока как «функцию», которую судья вызывает.

Упражнение 4.1: судейский менталитет

Поскольку судья является чисто классическим, мы будем моделировать его как использование классических генераторов случайных чисел. Однако это оставляет открытой возможность того, что мы с Евой могли бы сжульничать,

угадывая вопросы судьи. Возможным улучшением может быть использование QRNGS из главы 2. Модифицируйте пример исходного кода в листинге 4.1 так, чтобы судья мог задавать вопросы нам и Еве, измеряя кубит, который начинается в состоянии |+⟩.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

Как видно из листинга 4.1, мы объявляем новый тип `Strategy` для определения кортежа функций, представляющих наши и Евы однобитовые функции, которые представляют наши индивидуальные стратегии. На эти функции можно смотреть как на такие, которые являют то, что мы и Ева делаем с битами, поступающими к нам от судьи.

Листинг 4.1 chsh.py: имплементация игры CHSH на Python

```
import random
from functools import partial
from typing import Tuple, Callable
import numpy as np

from interface import QuantumDevice, Qubit
from simulator import Simulator

Strategy = Tuple[Callable[[int], int], Callable[[int], int]]          (1)

def random_bit() -> int:
    return random.randint(0, 1)                                       (2)

def referee(strategy: Callable[[], Strategy]) -> bool:
    you, eve = strategy()                                             (3)
    your_input, eve_input = random_bit(), random_bit()                (4)
    parity = 0 if you(your_input) == eve(eve_input) else 1            (5)
    return parity == (your_input and eve_input)                       (6)

def est_win_probability(strategy: Callable[[], Strategy],             (7)
                        n_games: int = 1000) -> float:
    return sum(
        referee(strategy)
        for idx_game in range(n_games)
    ) / n_games                                                       (8)
```

(1) **Использование Python'овского модуля typing позволяет документировать, что значение типа Strategy представляет собой кортеж из двух функций, каждая из которых принимает значение int и возвращает значение int.**

❷ Классический генератор случайных чисел, который будет использоваться судьей.
❸ Функция strategy: Callable[[] будет назначать однобитовые функции, представляющие то, что «вы» и «Ева» будете делать, основываясь на нашем входном значении.
❹ Судья выбирает два случайных бита, по одному для каждого игрока.
❺ Дает каждому игроку его случайный бит и затем вычисляет паритет их ответов.
❻ Свериться с табл. 4.1, чтобы узнать о выигрыше игроков.
❼ Делит на число сыгранных нами игр, а затем оценивает вероятность того, что наша стратегия и стратегия Евы выиграют игру CHSH.
❽ Мы можем использовать встроенную в Python функцию sum для подсчета числа раз, когда судья возвращает True для конкретной стратегии – другими словами, число раз, когда мы выиграли игру.

Обратите внимание, что в листинге 4.1 у нас еще нет определения входной стратегии `strategy` для судьи. Теперь, когда мы имплементировали правила игры на языке Python, давайте обсудим *стратегию* и перейдем к имплементированию классической стратегии игры CHSH.

ДЛЯ СПРАВКИ Полезно иметь правила именования переменных, раскрывающие роль, которую играет каждая переменная в нашем исходном коде. Мы решили использовать приставку `n_` в переменной `n_games`, чтобы указывать на то, что переменная ссылается на число или размер, и мы используем приставку `idx_` для ссылки на индекс каждой отдельной игры. Как и вождение автомобиля, неплохо, когда наш исходный код поддается предсказанию.

4.1.3 Классическая стратегия

Самая простая стратегия для нас и Евы состоит в том, чтобы полностью игнорировать наши входные значения. Глядя на табл. 4.3, если перед игрой мы оба согласимся, что никогда не будем менять свои выходные значения (т. е. всегда будем возвращать 0), то мы будем выигрывать в 75 % случаев (эта стратегия основывается на допущении, что судья равномерно выбирает случайные биты для каждого игрока).

Таблица 4.3 Наилучшая классическая стратегия для игры CHSH, где мы оба всегда отвечаем 0 с условиями выигрыша

Наше входное значение	Входное значение Евы	Наш ответ	Ответ Евы	Паритет	Выигрыш?
0	0	0	0	Четный	Да
0	1	0	0	Четный	Да
1	0	0	0	Четный	Да
1	1	0	0	Четный	Нет

Если бы мы написали эту стратегию как функцию Python, то у нас был бы следующий ниже исходный код.

Листинг 4.2 chsh.py: простая, постоянная стратегия для игры CHSH

```
def constant_strategy() -> Strategy:
    return (
        lambda your_input: 0,
        lambda eve_input: 0
    )
```

Теперь мы можем протестировать ожидаемую частоту выигрыша в раунде игры CHSH, используя `constant_strategy`:

```
>>> est_win_probability(constant_strategy)          ❶
0.771
```

❶ **Обратите внимание, что вы можете получить немного больше или меньше 75 % при попытке сделать это самим. Это связано с тем, что вероятность выигрыша оценивается с использованием конечного числа раундов (в статистике это называется биномиальным распределением). В данном примере мы ожидаем, что полосы ошибок[1] составят около 1.5 %.**

Хорошо. У нас есть простая стратегия, но есть ли что-нибудь умнее, что можно было бы сделать? Если учесть, что у нас с Евой есть только классические ресурсы, то, к сожалению, это лучшее, что можно сделать. За исключением жульничества 😈 (например, общения с Евой или угадывания входных значений судьи), выигрыш в 75 % случаев – это предел, и выиграть игру за пределами этого процента мы в среднем не сможем.

Все это приводит к очевидному вопросу: что, если бы мы и Ева могли использовать кубиты? Тогда какой будет наша наилучшая стратегия и как часто мы будем выигрывать? Что это говорит о нашем понимании Вселенной, если у нас есть доказательства того, что мы не можем выиграть CHSH более чем в 75 % случаев, а затем находим способ превзойти этот показатель выигрыша? Как вы можете догадаться, мы *можем* добиться большего, чем 75%-го выигрыша, играя в CHSH, если игроки делятся квантовыми ресурсами, т. е. имеют кубиты. Позже в этой главе мы перейдем к квантовым стратегиям для игры CHSH, но вот вам спойлер: нам нужно будет просимулировать более одного кубита.

4.2 Работа с многокубитовыми состояниями

До сих пор в этой книге мы работали, используя только по одному кубиту за раз. Однако, чтобы играть в нелокальную игру, каждому игроку понадобится свой собственный кубит. В связи с этим возникает вопрос: как

[1] Полосы, или столбики, ошибок (error bars) представляют собой графические представления изменчивости данных и используются в результирующих ячейках среды блокнотов Jupyter Notebook с ядром IQ# для языка Q# и на графиках для указания ошибки или неопределенности в измерении. См. https://en.wikipedia.org/wiki/Error_bar. – *Прим. перев.*

все изменится, когда рассматриваемая нами система будет иметь более одного кубита? Главное отличие будет состоять в том, что мы не сможем описывать каждый кубит отдельно и должны размышлять в терминах состояния, которое описывает всю систему.

> **ПРИМЕЧАНИЕ** При описании группы или реестра кубитов мы, как правило, *не можем* описывать каждый кубит по отдельности. Наиболее полезное квантовое поведение можно увидеть только тогда, когда мы описываем состояние группы или реестра кубитов.

Следующий далее раздел поможет связать это системно-уровневое представление с аналогичной классической программной концепцией реестра.

4.2.1 Реестры

Предположим, что у нас есть *реестр* классических битов, т. е. коллекция из нескольких классических битов. Мы можем проиндексировать каждый бит в этом реестре и просматривать его значение независимо, даже если он по-прежнему является частью этого реестра. Содержимое реестра может представлять более сложное значение, как в случае битов, которые вместе представляют символ Юникода (как мы увидели в главе 3), но эта высокоуровневая интерпретация не является необходимой.

Когда мы храним информацию в классическом реестре, число разных состояний этого реестра очень быстро растет по мере добавления новых битов. Например, если у нас есть три бита, то наш реестр может находиться в восьми разных состояниях; пример см. в листинге 4.3. В случае состояния классического реестра 101 мы говорим, что нулевой бит равен 1, первый – 0 и второй – 1. При конкатенировании этих значений они дают битовую строку 101.

Листинг 4.3 Перечисление всех состояний классического трехбитового реестра

```
>>> from itertools import product
>>> ", ".join(map("".join, product("01", repeat=3)))
'000, 001, 010, 011, 100, 101, 110, 111'
```

Если у нас есть четыре бита, то мы можем хранить одно из 16 разных состояний; если у нас есть n бит, то мы можем хранить одно из 2^n. Мы говорим, что число разных возможных состояний классического реестра *растет экспоненциально* вместе с числом битов. Битовые строки, выводимые в листинге 4.3, показывают фактические данные в реестре для каждого состояния реестра. Они также служат удобными метками для одного из восьми возможных сообщений, которые мы можем закодировать тремя классическими битами.

Какое отношение все это имеет к кубитам? В главах 2 и 3 мы узнали, что любое состояние классического бита также описывает состояние кубита. То же самое относится и к реестрам кубитов. Например, трехбитовое состояние «010» описывает трехкубитовое состояние $|010\rangle$.

ДЛЯ СПРАВКИ В главе 2 мы увидели, что состояния кубитов, описываемые классическими битами в таком ключе, называются *вычислительно-базисными состояниями*; здесь мы используем тот же термин для состояний нескольких кубитов, которые описываются строками классических битов.

Однако, подобно одиночным кубитам, состояние реестра из нескольких кубитов также может составляться путем сложения разных кубитовых состояний. Ровно в таком же ключе, в котором мы пишем $|+\rangle$ как $(|0\rangle + |1\rangle)/\sqrt{2}$, чтобы получить еще одно валидное кубитовое состояние, наш трехкубитовый реестр может находиться в самых разных состояниях:

- $(|010\rangle + |111\rangle)/\sqrt{2}$;
- $(|001\rangle + |010\rangle + |100\rangle)/\sqrt{3}$.

ДЛЯ СПРАВКИ По мере изложения мы увидим еще больше состояний, но подобно тому, как нам был нужен квадратный корень из 2, чтобы вероятности измерений работали как надо для $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$, нам нужно делить на $\sqrt{2}$ и $\sqrt{3}$ в примерах, чтобы обеспечивать реалистичность всех вероятностей каждого измерения, т. е. получать в сумме единицу.

Этот пример линейности квантовых реестров называется *принципом суперпозиции*.

ОПРЕДЕЛЕНИЕ *Принцип суперпозиции* говорит о том, что мы можем сложить два разных состояния квантового реестра и получить еще одно валидное состояние. Состояние $|+\rangle$, которое мы видели ранее, является хорошим тому примером, за исключением того, что каждый реестр имеет только один кубит.

В целях записи состояния квантового реестра в компьютере мы снова будем использовать векторы, подобно тому, как делали в главе 2. Главное различие заключается в том, сколько чисел мы перечисляем в каждом векторе. Давайте посмотрим, как выглядит запись состояния двухкубитового реестра на компьютере. Например, вектор для двухкубитового состояния $(|00\rangle + |11\rangle)/\sqrt{2}$ также может быть записан как состояние $(1 \times |00\rangle + 0 \times |01\rangle + 0 \times |10\rangle + 1 \times |11\rangle)/\sqrt{2}$. Если мы составим список того, на что нам нужно умножить каждое вычислительно-базисное состояние, чтобы получить желаемое состояние, то у нас будет именно та информация, которую нам нужно записать в наш вектор. В листинге 4.4 мы записали $(|00\rangle + |11\rangle)/\sqrt{2}$ как вектор.

Листинг 4.4 Использование Python для написания примера двухкубитового состояния

```
>>> import numpy as np
>>> two_qubit_state = np.array([[          ❶
...
...     1,                                 ❷
...
...     0,                                 ❸
...
...     0,
...     1
... ]]) / np.sqrt(2)                       ❹
```

❶ Мы начинаем таким же образом, используя функцию np.array для создания нового вектора.
❷ Каждый элемент в этом векторе описывает разное вычислительно-базисное состояние. Этот элемент говорит о том, что мы должны умножить |00⟩ на 1.
❸ Аналогичным образом этот элемент говорит о том, сколько состояния |01⟩ нам нужно добавить, чтобы получить желаемое состояние.
❹ Наконец, мы делим на $\sqrt{2}$, дабы обеспечить, чтобы все вероятности измерений работали как надо, подобно тому, как мы делали с состоянием |+⟩ в главах 2 и 3.

Числа в векторе из листинга 4.4 являются коэффициентами, которые мы умножаем на каждое из вычислительно-базисных состояний, которые затем складываем, чтобы создать новое состояние. Эти коэффициенты также называются *амплитудами* для каждого из базисных состояний в сумме.

Привычка мыслить направлениями

Об этом примере можно рассуждать и по-другому, с точки зрения направлений на карте местности, аналогично тому, как мы обсуждаем векторы в приложении С книги. Каждое отличающееся вычислительно-базисное состояние говорит нам о *направлении*, в котором состояние кубита может быть указано. Двухкубитовое состояние можно рассматривать как направление в четырех измерениях вместо двумерных карт, которые мы видим в приложении С книги. Поскольку эта книга является двумерной, а не четырехмерной, мы, к сожалению, не способны нарисовать здесь рисунок, но иногда думать о векторах как о направлениях бывает полезнее, чем думать о векторах как о списках чисел.

4.2.2 Почему трудно симулировать квантовые компьютеры?

Мы видели, что по мере роста числа битов число разных состояний, в которых может находиться реестр, растет экспоненциально. Хотя для нас это не будет проблемой, когда мы будем играть в нелокальную игру толь-

ко с двумя кубитами, мы захотим использовать больше, чем два кубита, когда продолжим эту книгу.

В этом случае у нас также будет экспоненциально много разных вычислительно-базисных состояний для квантового реестра, а это означает, что по мере увеличения размера квантового реестра нам понадобится экспоненциально много разных амплитуд в векторах. Для того чтобы записать состояние 10-кубитового реестра, нам нужно будет использовать вектор, представляющий собой список из 2^{10} = 1024 разных амплитуд. Для 20-кубитового реестра нам понадобится около 1 миллиона амплитуд в векторах. К тому времени, когда мы доберемся до 60 кубитов, нам понадобится около 1.15×10^{18} чисел в векторах. Это примерно одна амплитуда на каждую песчинку на планете.

В табл. 4.4 мы подытоживаем смысл этого экспоненциального роста, когда мы пытаемся симулировать квантовые компьютеры, используя классические компьютеры, такие как телефоны, ноутбуки, кластеры и облачные среды. Данная таблица показывает, что хотя очень сложно рассуждать о квантовых компьютерах с использованием классических компьютеров, мы можем довольно легко рассуждать о малых примерах. С помощью ноутбука или настольного компьютера мы без особых хлопот можем просимулировать до 30 кубитов. Как мы увидим в остальной части книги, этого более чем достаточно, чтобы понять принцип работы квантовых программ и путей, которыми квантовые вычисления могут использоваться для решения интересных задач.

Таблица 4.4 Сколько памяти требуется для хранения квантового состояния?

К-во кубитов	К-во амблитуд	Память	Сравнения размеров
1	2	128 бит	
2	4	256 бит	
3	8	512 бит	
4	16	1 килобит	
8	256	4 Кб	Бесконтактная кредитная карточка
10	1024	16 Кб	
20	1,048,576	16 Мб	
26	67,108,864	1 Гб	RAM для Raspberry Pi
28	268,435,456	4 Гб	RAM для iPhone XS Max
30	1,073,741,824	16 Гб	RAM для ноутбука или настольного компьютера
40	1,099,511,627,776	16 Тб	
50	1,125,899,906,842,624	16 петабайт	
60	1,152,921,504,606,846,976	16 экзабайт	
80	1,208,925,819,614,629,174,706,176	16 йотабайт	Приблизительный размер интернета
410	2.6×10^{123}	4.2×10^{124} байт	Компьютер размером со Вселенную

Глубокое погружение: разве мощность квантовых компьютеров не экспоненциально больше?

Возможно, вы слышали, что число разных чисел, которые мы должны отслеживать, чтобы симулировать квантовый компьютер при участии классического компьютера, является причиной, почему квантовые компьютеры мощнее, или что квантовый компьютер может хранить столько много информации. Это не совсем так. Математическая теорема под названием *теорема Холево*, говорит о том, что квантовый компьютер, состоящий из 410 кубитов, может хранить не более 410 классических бит информации, даже если для записи состояния этого квантового компьютера потребуется классический компьютер размером со всю Вселенную.

Говоря по-другому, то, что *квантовый компьютер трудно симулировать, вовсе не означает, что он делает что-то полезное*. В остальной части книги мы увидим, что в целях понимания того, как использовать квантовый компьютер для решения полезных задач, требуется немного искусства.

4.2.3 Тензорные произведения для подготовки состояний

Уметь описывать квантовые реестры как векторы, описывающие вычислительно-базисные состояния, – все это хорошо и здорово, но даже если мы знаем состояние, в которое хотим попасть, нам нужно знать, как его подготовить. Например, если один игрок в нелокальной игре имеет кубит в состоянии $|0\rangle$, а другой игрок – кубит в состоянии $|1\rangle$, то мы можем скомбинировать эти два однокубитовых состояния простым способом и описать состояние игры как $|01\rangle$. Что значит «комбинировать» состояния двух (или более) кубитов? Мы можем сделать это, добавив в наш математический инструментарий еще одно понятие, именуемое *тензорным произведением*.

Точно так же, как мы использовали функцию `product` в листинге 4.3 для комбинирования меток реестра из трех классических битов, мы можем использовать концепцию тензорного произведения, записываемого как $\otimes$, для комбинирования квантовых состояний каждого кубита и создавать состояние, описывающее несколько кубитов. Результатом работы функции `product` был список всех возможных состояний этих трех классических битов. Результатом тензорного произведения по аналогии является состояние, в котором перечислены все вычислительно-базисные состояния квантового реестра. Для вычисления тензорных произведений мы можем воспользоваться пакетом NumPy; он предлагает имплементацию тензорного произведения в виде функции `np.kron`, как показано в листинге 4.5.

Листинг 4.5 Двухкубитовое состояние с использованием тензорного произведения на основе NumPy

```
>>> import numpy as np
>>> ket0 = np.array([[1], [0]])      ❶
```

```
>>> ket1 = np.array([[0], [1]])
>>> np.kron(ket0, ket1)                    ❷
array([[0],
       [1],                                ❸
       [0],
       [0]])
```

❶ Определяет векторы для однокубитовых состояний |0⟩ и |1⟩, как и в главах 2 и 3.
❷ Мы можем построить вектор для |0⟩ ⊗ |1⟩ путем вызова имплементации тензороного произведения в пакете Numpy. По историческим причинам тензорное произведение в Numpy представлено функцией kron.
❸ Вектор, возвращаемый функцией np.kron, имеет 1 для элемента, соответствующего вычислительно-базисному состоянию |01⟩, и нули в остальных случаях, и поэтому мы понимаем, что этот вектор находится в состоянии |01⟩.

Почему kron?

Возможно, название `np.kron` выглядит странным для функции, в которой имплементировано тензорное произведение, но это название является сокращением для родственной математической концепции под названием *произведение Кронекера*. Использование функции `kron` пакета NumPy в качестве сокращения от *Kronecker* следует общепринятому соглашению, используемому в MATLAB, R, Julia и других научно-вычислительных платформах.

Приведенный выше пример показывает, что |0⟩ ⊗ |1⟩ = |01⟩. То есть если у нас есть состояние каждого кубита в отдельности, то мы можем их скомбинировать, используя тензорное произведение для описания состояния всего реестра.

Используя такой подход, мы можем комбинировать столько кубитов, сколько захотим. Допустим, у нас было четыре кубита, все в состоянии |0⟩. Реестр всех четырех кубитов можно описать следующим образом: |0⟩ ⊗ |0⟩ ⊗ |0⟩ ⊗ |0⟩.

Листинг 4.6 Создание |0000⟩ в качестве тензорного произведения 4 копий |0⟩

```
>>> import numpy as np
>>> ket0 = np.array([[1], [0]])
>>> from functools import reduce
>>> reduce(np.kron, [ket0] * 4)            ❶
array([[1],                                ❷
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
```

```
       [0],
       [0],
       [0],
       [0],
       [0],
       [0]])
```

❶ Функция reduce, предлагаемая стандартной библиотекой языка Python, позволяет нам применять двухаргументную функцию, подобную kron, между каждым элементом в списке. Здесь мы используем reduce вместо np.kron(ket0, [0], np.kron(ket0, np.kron(ket0, ket0))).

❷ Мы получаем обратно четырехкубитовый вектор состояний, представляющий $|0\rangle \otimes |0\rangle \otimes |0\rangle \otimes |0\rangle = |0000\rangle$.

ПРИМЕЧАНИЕ Двухкубитовое состояние $(|00\rangle + |11\rangle)/\sqrt{2}$, которое мы встречали ранее, *невозможно* записать как тензорное произведение двух однокубитовых состояний. Многокубитовые состояния, которые нельзя записать в виде тензорных произведений, называются *запутанными*. Мы узнаем гораздо больше о запутанности в остальной части этой главы и в остальной части книги.

4.2.4 Тензорные произведения для кубитовых операций на реестрах

Теперь, когда мы знаем, как использовать тензорное произведение для комбинирования квантовых состояний, следует выяснить, что на самом деле делает функция `np.kron`. По сути, тензорное произведение представляет собой таблицу всех разных способов комбинирования его двух аргументов, как показано на рис. 4.2.

То же самое тензорное произведение двух матриц на Python, показанное на рис. 4.2, проиллюстрировано в следующем ниже листинге.

Листинг 4.7 Отыскание тензорного произведения A и B с помощью NumPy

```
>>> import numpy as np
>>> A = np.array([[1, 3], [5, 7]])          ❶
>>> B = np.array([[2, 4], [6, 8]])
>>> np.kron(A, B)                           ❷
array([[ 2,  4,  6, 12],
       [ 6,  8, 18, 24],
       [10, 20, 14, 28],
       [30, 40, 42, 56]])
>>> np.kron(B, A)                           ❸
array([[ 2,  6,  4, 12],
       [10, 14, 20, 28],
       [ 6, 18,  8, 24],
       [30, 42, 40, 56]])
```

❶ Матрицы A и B являются произвольными матрицами 2×2, используемыми здесь в качестве примеров.

❷ Как мы узнали ранее, функция np.kron является имплементацией тензорного произведения в NumPy.

❸ Обратите внимание, что порядок следования аргументов тензорного произведения имеет значение. Хотя np.kron(A, B) и np.kron(B, A) содержат одну и ту же информацию, элементы в каждом из них упорядочены совершенно по-разному!

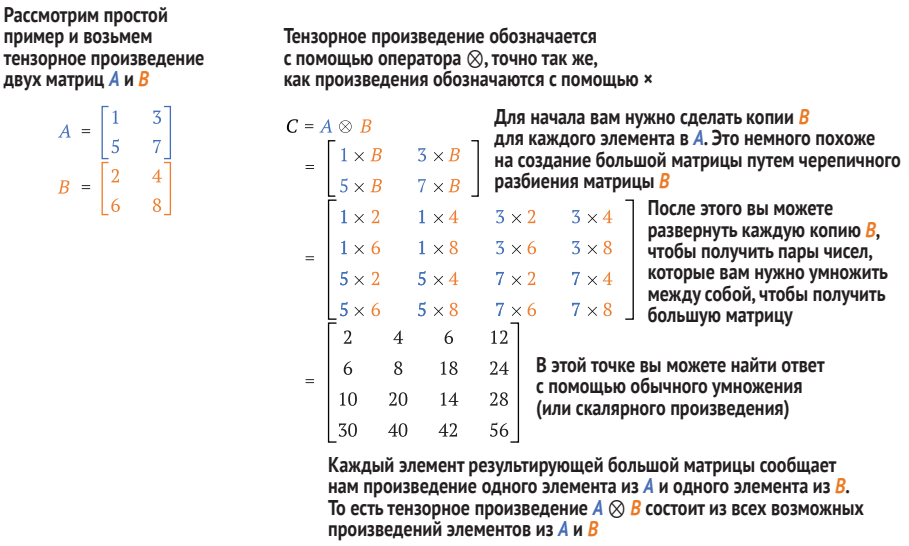


Рис. 4.2 Тензорное произведение двух матриц, показанное пошагово. Мы начинаем с умножения каждого члена в *A* на полную копию *B*. Результирующие размеры матрицы тензорного произведения являются произведением размеров входных матриц

Используя тензорное произведение между двумя матрицами, мы можем найти то, как разные квантовые операции преобразовывают состояние квантового реестра. Мы также можем понять и то, как квантовая операция преобразовывает состояние нескольких кубитов, взяв вместо этого тензорное произведение двух матриц, позволяя нам понять, как наши с Евой ходы в нелокальной игре влияют на наше общее состояние

Например, мы знаем, что можем записать $|1\rangle$ как $X|0\rangle$ (т. е. результат применения инструкции x к инициализированному кубиту). Это также дает нам еще один способ записи многокубитовых состояний, таких как состояние $|01\rangle$, которое мы встречали ранее. В этом случае мы можем получить $|01\rangle$, применив инструкцию x *только* ко второму кубиту двухкубитового реестра. Используя тензорное произведение, мы можем найти унитарную матрицу, которая будет его представлять.

Листинг 4.8 Вычисление тензорного произведения двух матриц

```
>>> import numpy as np
>>> I = np.array([[1, 0], [0, 1]])          ❶
>>> X = np.array([[0, 1], [1, 0]])          ❷
>>> IX = np.kron(I, X)                      ❸
```

```
>>> IX
array([[0, 1, 0, 0],                    ❹
       [1, 0, 0, 0],
       [0, 0, 0, 1],
       [0, 0, 1, 0]])
>>> ket0 = np.array([[1], [0]])         ❺
>>> ket00 = np.kron(ket0, ket0)
>>> ket00
array([[1],
       [0],
       [0],
       [0]])
>>> IX @ ket00
array([[0],
       [1],                              ❻
       [0],
       [0]])
```

❶ **Определяет матрицу, которая представляет ничегонеделание с первым кубитом, именуемую матрицей тождественности $\mathbb{1}$. Поскольку символ $\mathbb{1}$ трудно написать на Python, вместо этого мы используем `I`.**

❷ **Определяет унитарную матрицу *X*, которая позволяет нам просимулировать инструкцию `x`.**

❸ **Комбинирует эти два параметра с помощью тензорного произведения $I \otimes X$.**

❹ **Матрица $\mathbb{1} \otimes X$ состоит из двух копий *X*, представляющих то, что происходит со вторым кубитом для каждого возможного состояния первого кубита.**

❺ **Давайте посмотрим, что происходит, когда мы используем $\mathbb{1} \otimes X$ для симулирования того, как инструкция x преобразовывает второй кубит в двухкубитовом реестре. Мы начнем с этого реестра в состоянии $|00\rangle = |0\rangle \otimes |0\rangle$.**

❻ **Мы понимаем, что состояние, которое мы получаем назад, является состоянием, описанным ранее в этом разделе. Как и ожидается, это состояние мы получаем, переворачивая второй кубит из $|0\rangle$ в $|1\rangle$.**

Упражнение 4.2: операция Адамара с двухкубитовым реестром

Как бы вы подготовили состояние $|+0\rangle$? Прежде всего какой вектор вы бы использовали для представления двухкубитового состояния $|+0\rangle = |+\rangle \otimes |0\rangle$? У вас есть начальный двухкубитовый реестр в состоянии $|00\rangle$. Какую операцию вы должны применить, чтобы получить желаемое состояние?

Подсказка: попробуйте $(H \otimes \mathbb{1})$, если вы застряли!

Глубокое погружение: окончательное доказательство теоремы о запрете клонирования

Усвоив, что операции на нескольких кубитах также представляются унитарными матрицами, мы наконец получаем возможность доказать теорему о запрете клонирования, с которой мы сталкивались несколько раз до этого. Ключевая идея заключается в том, что клонирование состояния не является линейным и, следовательно, не может быть записано в виде матрицы.

Как и во многих доказательствах в математике, доказательство теоремы о запрете клонирования выполняется *от противного*. То есть мы допускаем существование противоположного теореме, а затем показываем, что мы получаем что-то ложное вследствие этого допущения.

Без дальнейших церемоний мы начнем с допущения о том, что у нас есть какая-то удивительная инструкция клонирования `clone`, которая может идеально копировать состояние своего кубита. Например, если у нас есть кубит `q1`, состояние которого начинается в $|1\rangle$, и кубит `q2`, состояние которого начинается в $|0\rangle$, то после вызова `q1.clone(q2)` у нас будет реестр $|11\rangle$.

Аналогичным образом, если `q1` начинается в $|+\rangle$, то `q1.clone(q2)` должно дать нам реестр в состоянии $|++\rangle = |+\rangle \otimes |+\rangle$. Проблема заключается в согласовании того, что конкретно `q1.clone(q2)` должна делать в этих двух случаях. Мы знаем, что любая квантовая операция, отличная от операции измерения, должна быть линейной, поэтому давайте дадим матрице, которая позволяет нам симулировать клонирование, имя: C выглядит довольно разумным.

Используя C, мы можем подразделить случай, в котором мы хотим клонировать $|+\rangle$, на случай, в котором мы хотим клонировать $|0\rangle$, плюс случай, в котором хотим клонировать $|1\rangle$. Мы знаем, что $C|+0\rangle = |++\rangle$, но мы также знаем, что $C|+0\rangle = C(|00\rangle + |10\rangle)/\sqrt{2} = (C|00\rangle + C|10\rangle)/\sqrt{2}$. Поскольку функция clone должна клонировать $|0\rangle$ и $|1\rangle$, а также $|+\rangle$, мы знаем, что $C|00\rangle = |00\rangle$ и $C|10\rangle = |11\rangle$. Это дает нам, что $(C|00\rangle + C|10\rangle)/\sqrt{2} = (|00\rangle + |11\rangle)/\sqrt{2}$, но ранее мы пришли к выводу, что $C|+0\rangle = |++\rangle = (|00\rangle + |01\rangle + |10\rangle + |11\rangle)/\sqrt{2}$.

Следовательно, мы имеем противоречие и можем сделать вывод о том, что мы ошиблись на самом первом шаге, где мы допустили, что функция clone может существовать! Таким образом, мы продемонстрировали доказательство теоремы о запрете клонирования.

Из этой аргументации следует отметить одну важную вещь: мы всегда можем скопировать информацию из одного кубита в другой, *если знаем правильный базис*. Проблема возникла, когда мы не знали, что нам следует копировать: информацию о $|0\rangle$ или $|1\rangle$ либо информацию о $|+\rangle$ или $|-\rangle$, так как мы могли копировать либо $|0\rangle$, либо $|+\rangle$, но не оба. Это не проблема в классическом смысле, поскольку у нас всегда для работы есть только вычислительный базис.

До этого мы использовали пакет NumPy для написания симулятора кубита, `SingleQubitSimulator()`. Это очень помогло, так как без NumPy нам пришлось бы писать собственные функции и методы матричного анализа. Однако часто бывает удобно опираться на пакеты Python со специальной поддержкой квантовых концепций, беря за основу превосходную числовую поддержку, предоставляемую пакетами NumPy и SciPy (расширением числовых возможностей NumPy). В следующей главе мы рассмотрим пакет Python, специально предназначенный для квантовых вычислений, под названием QuTiP (Quantum Toolbox in Python, т. е. квантовый инструментарий на Python), который поможет нам завершить модернизацию нашего симулятора для симулирования игры CHSH.

Резюме

- Нелокальные игры, такие как игра CHSH, представляют собой эксперименты, которые можно использовать для проверки того, что квантовая механика согласуется с тем, как на самом деле работает наша Вселенная.
- Для записи состояния квантового реестра требуется экспоненциально много классических битов, что очень затрудняет симулирование более 30 кубитов с использованием традиционных компьютеров.
- Мы можем комбинировать однокубитовые состояния, используя тензорное произведение, позволяющее нам описывать многокубитовые состояния.
- Многокубитовые состояния, которые невозможно записать путем комбинирования однокубитовых состояний, представляют кубиты, которые запутаны друг с другом.

5 Нелокальные игры: имплементирование многокубитового симулятора

Эта глава охватывает следующие ниже темы:

- программирование симулятора для нескольких кубитов с использованием пакета QuTiP языка Python и тензорных произведений;
- знакомство с доказательством того, что квантовая механика согласуется с нашими наблюдениями Вселенной, путем симулирования экспериментальных результатов.

В предыдущей главе мы узнали о нелокальных играх и о том, как их можно использовать для подтверждения нашего понимания квантовой механики. Мы также узнали о том, как представлять состояния нескольких кубитов и что такое *запутанность* (*entanglement*, переплетенность).

В этой главе мы подробно остановимся на новом пакете Python под названием QuTiP, который позволит нам программировать квантовые системы быстрее и имеет несколько интересных встроенных функций для симулирования квантовой механики. Затем мы узнаем, как использовать QuTiP для программирования симулятора для нескольких кубитов, и посмотрим, как это изменит (или не изменит!) три главные задачи для наших кубитов: подготовку состояния, операции и измерение. Это позволит нам завершить имплементацию игры CHSH из главы 4!

5.1 Квантовые объекты в QuTiP

Пакет QuTiP (Квантовый инструментарий на Python, www.qutip.org) в особенности полезен тем, что он обеспечивает встроенную поддержку представления состояний и измерений соответственно в виде бра и кетов, а также для построения матриц для представления квантовых операций. Подобно тому, как `np.array` лежит в самой сердцевине пакета NumPy, все наше использование пакета QuTiP будет сосредоточено вокруг класса `Qobj` (сокращение от *quantum object*). Этот класс инкапсулирует векторы и матрицы, предоставляя дополнительные метаданные и полезные методы, которые облегчат нам усовершенствование нашего симулятора. На рис. 5.1 показан пример создания квантового объекта `Qobj` из вектора, где он отслеживает некоторые метаданные:

- `data` содержит массив, представляющий квантовый объект `Qobj`;
- `dims` – это размер нашего квантового реестра. Его можно рассматривать как способ отслеживать то, как мы регистрируем кубиты, с которыми работаем;
- `shape` содержит размер изначального объекта, который мы использовали для создания квантового объекта `Qobj`. Он похож на атрибут `np.shape`;
- `type` – это то, что квантовый объект `Qobj` представляет (состояние = `ket`, измерение = `bra` или оператор = `oper`).

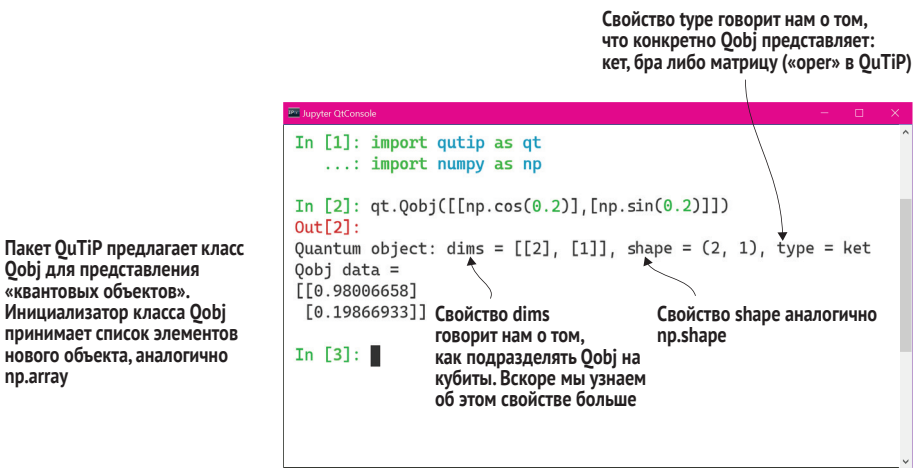


Рис. 5.1 Свойства класса `Qobj` в пакете QuTiP. Здесь мы видим такие вещи, как свойства `type` и `dims`, которые помогают нам и пакету отслеживать метаданные о наших квантовых объектах

Давайте попробуем импортировать QuTiP и запросить операцию Адамара; см. листинг 5.1.

ПРИМЕЧАНИЕ При выполнении убедитесь, что вы находитесь в правильной среде `conda env`; дополнительные сведения см. в приложении А к книге.

Листинг 5.1 Представление операции Адамара в QuTiP

```
>>> from qutip.qip.operations import hadamard_transform
>>> H = hadamard_transform()
>>> H
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.70710678  0.70710678]
 [ 0.70710678 -0.70710678]]
```

Обратите внимание, что пакет QuTiP распечатывает диагностическую информацию о каждом экземпляре класса `Qobj` вместе с самими данными. Здесь, например, `type = oper` говорит нам о том, что `H` представляет оператор (более формальный термин для матриц, которые мы встречали до сих пор) вместе с некоторой информацией о размерности оператора, представленного `H`. Наконец, `isherm = True` говорит нам о том, что `H` является примером особого вида матрицы, именуемой *эрмитовым оператором*.

Мы можем создавать новые экземпляры класса `Qobj` почти так же, как создавали массивы NumPy, передавая списки Python инициализатору класса `Qobj`.

Листинг 5.2 Создание `Qobj` из вектора, представляющего состояние кубита

```
>>> import qutip as qt
>>> ket0 = qt.Qobj([[1], [0]])                                    (1)
>>> ket0
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket     (2)
Qobj data =
[[1.]
 [0.]]
```

(1) Одно из ключевых различий между созданием экземпляров класса `Qobj` и массивов заключается в том, что при создании экземпляров класса `Qobj` нам всегда нужны списки двух уровней. Внешний список – это список строк в новом экземпляре класса `Qobj`.

(2) QuTiP распечатывает метаданные о размере и форме нового квантового объекта вместе с данными, содержащимися в новом объекте. Здесь данные для нового квантового объекта `Qobj` имеют две строки, каждая с одним столбцом. Мы идентифицируем их как вектор или кет, который используем для записи состояния $|0\rangle$.

Упражнение 5.1: создание квантового объекта `Qobj` для других состояний

Как бы вы создали квантовый объект `Qobj` для представления состояния $|1\rangle$? А как насчет состояния $|+\rangle$ или $|-\rangle$? Если вам нужно, вернитесь к разделу 2.3.5, чтобы узнать, какие векторы представляют эти состояния.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

Пакет QuTiP действительно помогает, предлагая много хороших сокращений для типов объектов, с которыми нам нужно работать в квантовых вычислениях. Например, в предыдущем примере мы могли бы также изготовить `ket0`, используя функцию `basis` пакета QuTiP; см. листинг 5.3. Функция `basis` принимает два аргумента. Первый говорит QuTiP о том, что нам требуется состояние кубита: `2` для одного кубита из-за длины вектора, необходимого для его представления. Второй аргумент говорит QuTiP о том, какое базисное состояние нам нужно.

Листинг 5.3 Использование QuTiP для легкого создания |0⟩ и |1⟩

```
>>> import qutip as qt
>>> ket0 = qt.basis(2, 0)                                          ❶
>>> ket0
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket      ❷
Qobj data =
[[1.]
 [0.]]
>>> ket1 = qt.basis(2, 1)                                          ❸
>>> ket1
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[0.]
 [1.]]
```

❶ Передает 2 в качестве первого аргумента, говоря о том, что нам нужен один кубит, и передает 0 для второго аргумента, потому что мы хотим |0⟩.

❷ Обратите внимание, что здесь мы получаем точно такой же результат, как и в предыдущем примере.

❸ Мы также можем построить квантовый объект для |1⟩, передав 1 вместо 0.

...базис?

Как мы видели ранее, состояния |0⟩ и |1⟩ составляют *вычислительный базис* для одного кубита. Функция `basis` пакета QuTiP получает свое название из этого определения, поскольку она создает квантовые объекты для представления вычислительно-базисных состояний.

Много в мире есть того, что вашим кубитам и не снилось

Может показаться немного странным, что нам пришлось сообщать пакету QuTiP о том, что нам нужен кубит. В конце концов, чего еще мы *могли* хотеть? Как оказалось, довольно много!

Кроме битов, существует еще много других способов представления классической информации, таких как *триты*, которые имеют три возможных значения. Однако при написании программ мы склонны не видеть классическую информацию, представляемую с использованием чего-либо еще, кроме битов, так как очень полезно выбрать общепринятое правило и его придерживаться. Однако вещи, отличные от битов, все-таки используются, в частности в специализированных областях, таких как телекоммуникационные системы.

Ровно в таком же ключе квантовые системы могут иметь любое число разных состояний: у нас могут быть кутриты (qutrits), ку4иты (qu4its), ку5иты (qu5its), ку17иты (qu15its) и т. д., в совокупности именуемые кудитами (qudits). В то время как представление квантовой информации с помощью кудитов, отличных от кубитов, бывает полезным в некоторых случаях и обладает некоторыми очень интересными математическими свойствами, кубиты дают нам все, что нам нужно, чтобы погрузиться в квантовое программирование.

Упражнение 5.2: использование функции `qt.basis` для нескольких кубитов

Как применить функцию `qt.basis` для создания двухкубитового реестра в состоянии $|10\rangle$? Как создать состояние $|001\rangle$? Помните, что второй аргумент функции `qt.basis` является индексом вычислительно-базисных состояний, которые мы встречали ранее.

Пакет QuTiP также предоставляет разнообразные функции по созданию квантовых объектов для представления унитарных матриц. Например, мы можем создать квантовый объект для матрицы X с помощью функции `sigmax`.

Листинг 5.4 Использование пакета QuTiP при создании объекта для матрицы X

```
>>> import qutip as qt
>>> qt.sigmax()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[0. 1.]
 [1. 0.]]
```

Как мы видели в главе 2, матрица для `sigmax` представляет собой поворот на 180° (рис. 5.2).

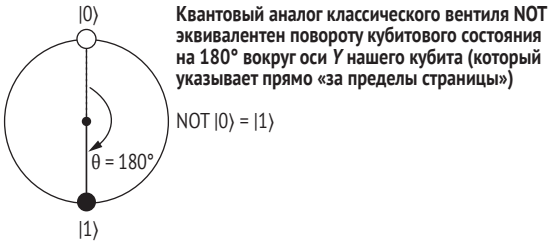


Рис. 5.2 Визуализация квантового эквивалента операции NOT, работающей с кубитом в состоянии |0⟩, оставляя кубит в состоянии |1⟩

Пакет QuTiP также предлагает функцию `ry` для представления поворачивания на любой желаемый угол, вместо 180°, как в операции *x*. В главе 2 мы уже видели операцию, представляемую функцией `ry`, когда рассматривали поворачивание |0⟩ на произвольный угол θ. Вы можете освежить свои знания по операции, которую мы теперь знаем как `ry`, обратившись к рис. 5.3.

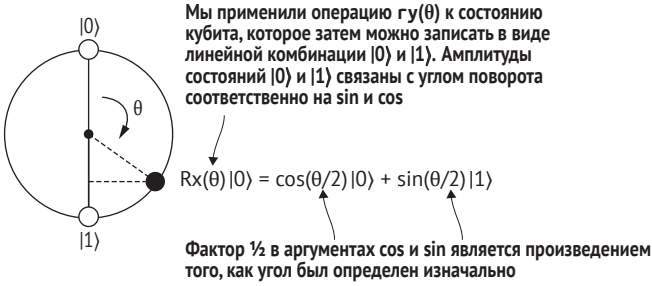


Рис. 5.3 Визуализация функции `ry`, которая соответствует переменному повороту θ вокруг оси *Y* нашего кубита (который указывает непосредственно за пределы страницы)

Теперь, когда у нас есть еще несколько однокубитовых операций, каким образом мы могли бы легко просимулировать многокубитовые операции в QuTiP? Для быстрой организации работы с тензорными произведениями с целью создания наших многокубитовых реестров и операций мы можем задействовать функцию `tensor` пакета QuTiP, как показано в листинге 5.5.

ПРИМЕЧАНИЕ Поскольку матрицы тожественности часто пишутся с использованием буквы *I*, во многих научно-вычислительных пакетах используется имя `eye` как своего рода каламбур для обозначения матрицы тождественности.

Листинг 5.5 Тензорные произведения в пакете QuTiP

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
```

```
>>> psi = qt.basis(2, 0)                    ❶

>>> phi = qt.basis(2, 1)                    ❷
>>> qt.tensor(psi, phi)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.]                                      ❸
 [ 1.]
 [ 0.]
 [ 0.]]
>>> H = hadamard_transform()                ❹

>>> I = qt.qeye(2)                          ❺

>>> qt.tensor(H, I)                         ❻
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4),
➥ type = oper, isherm = True
Qobj data =
[[ 0.70710678  0.          0.70710678  0.        ]
 [ 0.          0.70710678  0.          0.70710678]
 [ 0.70710678  0.         -0.70710678  0.        ]
 [ 0.          0.70710678  0.         -0.70710678]]
```

❶ Задает переменную psi как представляющую $|\psi\rangle = |0\rangle$.
❷ Задает переменную phi как представляющую $|\varphi\rangle = |1\rangle$.
❸ После вызова функции tensor пакет QuTiP сообщает нам амплитуды для каждой классической метки в $|\psi\rangle \otimes |\varphi\rangle = |0\rangle \otimes |1\rangle = |01\rangle$, используя тот же порядок, что и в листинге 4.3.
❹ Задает переменную H как представляющую рассмотренную ранее операцию Адамара.
❺ Мы можем применить функцию qeye пакета QuTiP, чтобы получить копию экземпляра Qobj, представляющего матрицу тождественности, которую мы впервые увидели в листинге 4.8.
❻ Унитарные матрицы, представляющие квантовые операции, комбинируются, используя тензорные произведения таким же образом, как состояния и измерения.

Мы можем использовать распространенный математический прием, чтобы доказать применение тензорных произведений состояний и операций. Скажем, мы хотим доказать вот это утверждение:

> *Если мы применим унитарный оператор к состоянию, а затем возьмем тензорное произведение, то получим тот же ответ, как если бы мы применили тензорное произведение, а потом унитарный оператор.*

В математике мы сказали бы, что для любого унитарного оператора U и V и для любого состояния $|\psi\rangle$ и $|\varphi\rangle$: $(U|\psi\rangle) \otimes (V|\varphi\rangle) = (U \otimes V)(|\psi\rangle \otimes |\varphi\rangle)$. Математический прием, который можно тут применить, состоит в том, чтобы взять левую сторону и вычесть из нее правую. В итоге мы должны получить 0. Мы попробуем это сделать в следующем ниже листинге.

Листинг 5.6 Проверка тензорного произведения в QuTiP

```
>>> (
...     qt.tensor(H, I) * qt.tensor(psi, phi) -     ❶
...     qt.tensor(H * psi, I * phi)                 ❷
```

```
... )
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.]                                                    ❸
 [ 0.]
 [ 0.]
 [ 0.]]
```

❶ Правая сторона утверждения, которое мы пытаемся доказать, где мы используем *H* и *I* как *U* и *V*: $(U \otimes V)\ (|\psi\rangle \otimes |\varphi\rangle)$.
❷ Левая сторона утверждения, которое мы пытаемся доказать, где мы используем *H* и *I* как *U* и *V*: $(U|\psi\rangle) \otimes (V|\varphi\rangle)$.
❸ Ура! Две стороны уравнения равны, если их разница равна 0.

ПРИМЕЧАНИЕ Список всех встроенных в пакет QuTiP состояний и операций см. по адресу http://qutip.org/docs/latest/guide/guide-basics.html#states-and-operators.

5.1.1 Модернизация симулятора

Теперь наша цель состоит в том, чтобы применить пакет QuTiP для модернизации нашего однокубитового симулятора, сделав его многокубитовым с несколькими функциональностями из пакета QuTiP. Мы сделаем это, добавив несколько функциональных элементов в указанный однокубитовый симулятор из глав 2 и 3.

Самое существенное изменение, которое нам нужно будет внести в наш симулятор из предыдущих глав, состоит в том, что мы больше не можем назначать состояние каждому кубиту. Вернее, мы должны назначать состояние всему *реестру* кубитов в нашем устройстве, поскольку некоторые кубиты могут быть *запутаны* между собой. Давайте перейдем к внесению модификаций, необходимых для разделения концепции состояния на уровне устройства.

ПРИМЕЧАНИЕ Для того чтобы взглянуть на исходный код, который мы написали ранее, а также на примеры этой главы, обратитесь к репозиторию книги на GitHub: https://github.com/crazy-4pi314/learn-qc-with-python-and-qsharp.

В целях ревизии напомним, что наш симулятор имеет два файла: интерфейс (interface.py) и сам симулятор (simulator.py). Интерфейс устройства (`QuantumDevice`) определяет способ взаимодействия с реальным или симулированным квантовым устройством, которое на Python представлено как объект, позволяющий нам выделять и высвобождать кубиты.

В интерфейсе для моделирования игры CHSH нам не понадобится ничего нового для класса `QuantumDevice`, так как нам все равно нужно будет выделять и высвобождать кубиты. Место, куда мы можем добавить функциональность, находится в классе `Qubit`, размещенном вместе с классом `SingleQubitSimulator` в файле simulator.py.

Теперь нам нужно подумать о том, что конкретно необходимо изменить в интерфейсе для класса `Qubit`, который мы выделяем из `QuantumDevice`. В главе 2 мы увидели, что операция Адамара была полезна для поворачивания кубитов между разными базисами для создания QRNG-генератора. Давайте будем опираться на это, добавив в класс `Qubit` новый метод, чтобы предоставлять квантовым программам возможность отправлять новый вид поворотных инструкций, которые нам понадобятся для использования квантовой стратегии в игре CHSH.

Листинг 5.7 interface.py: добавление новой операции `ry`

```
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass

    @abstractmethod
    def x(self): pass

    @abstractmethod
    def ry(self, angle: float): pass      (1)

    @abstractmethod
    def measure(self) -> bool: pass

    @abstractmethod
    def reset(self): pass
```

(1) **Абстрактный метод `ry`, принимающий аргумент `angle`, который указывает угол, на который следует повернуть кубит вокруг оси *Y***

Это должно охватить все необходимые изменения, которые мы внесем в интерфейс классов `Qubit` и `QuantumDevice` для игры в CHSH с Евой. Осталось только решить, какие изменения нам необходимо внести в simulator.py, чтобы обеспечить выделение, управление и измерение многокубитовых состояний.

Главные изменения в классе `Simulator`, имплементирующем класс `QuantumDevice`, заключаются в том, что нам нужны атрибуты для отслеживания числа кубитов и совокупного состояния реестра. В следующем ниже листинге показаны все эти изменения, а также обновление методов выделения и высвобождения.

Листинг 5.8 simulator.py: многокубитовый класс `Simulator`

```
class Simulator(QuantumDevice):                    (1)
    capacity: int                                  (2)
    available_qubits: List[SimulatedQubit]         (3)
    register_state: qt.Qobj                        (4)
    def __init__(self, capacity=3):
        self.capacity = capacity
        self.available_qubits = [                  (5)
            SimulatedQubit(self, idx)
```

```
            for idx in range(capacity)
        ]
        self.register_state = qt.tensor(                    ❻
            *[
                qt.basis(2, 0)
                for _ in range(capacity)
            ]
        )
    def allocate_qubit(self) -> SimulatedQubit:             ❼
        if self.available_qubits:
            return self.available_qubits.pop()
    def deallocate_qubit(self, qubit: SimulatedQubit):
        self.available_qubits.append(qubit)
```

❶ Мы поменяли имя класса SingleQubitSimulator на Simulator, чтобы обозначить, что этот класс является более обобщенным. Это означает, что с его помощью мы можем симулировать несколько кубитов.
❷ Более общий класс Simulator нуждается в нескольких атрибутах, первым из которых является capacity, представляющий число кубитов, которые он может симулировать.
❸ available_qubits – это список, содержащий кубиты, используемые симулятором.
❹ register_state использует новый Qobj пакета QuTiP, который представляет состояние всего симулятора.
❺ Операция включения в список позволяет нам составлять список доступных кубитов путем вызова SimulatedQubit с индексами из диапазона capacity.
❻ register_state инициализируется, принимая тензорное произведение числа копий состояния $|0\rangle$, равное емкости (capacity) симулятора. Нотация *[...] превращает сгенерированный список в последовательность аргументов для функции qt.tensor.
❼ Методы allocate_qubit и deallocate_qubit такие же, как и в главе 3.

Не вглядывайся в ящик, смертная душа!

Подобно тому, как мы использовали пакет NumPy для представления состояния симулятора, свойство `register_state` нашего только что модернизированного симулятора использует пакет QuTiP для предсказания того, как каждая инструкция преобразовала состояние нашего реестра. Однако когда мы пишем квантовые программы, мы делаем это согласно интерфейсу из листинга 5.7, который никак не позволяет нам получать доступ к `register_state`.

Симулятор можно рассматривать как своего рода черный ящик, который *инкапсулирует* понятие состояния. Если бы наши квантовые программы могли заглядывать в этот ящик, то они могли бы жульничать, копируя информацию путями, запрещенными теоремой о запрете клонирования. Это означает, что для того, чтобы квантовая программа была правильной, мы не можем заглядывать внутрь симулятора, чтобы увидеть его состояние.

В этой главе мы немного сжульничаем, но в следующей главе мы это исправим с целью обеспечения того, чтобы наши программы могли выполняться на реальном квантовом оборудовании.

Мы также добавим в класс `Simulator` новый *приватный* метод, который позволит нам применять операции к конкретным кубитам в нашем

устройстве. Это позволит нам написать методы на кубитах, которые отправляют операции обратно в симулятор для их применения к состоянию всего реестра кубитов.

ДЛЯ СПРАВКИ Python не строг в отношении поддержки методов или атрибутов приватными, но мы добавим к имени этого метода символ подчеркивания, чтобы указать, что он предназначен только для использования в классе.

Листинг 5.9 simulator.py: один дополнительный метод для класса `Simulator`

```
def _apply(self, unitary: qt.Qobj, ids: List[int]):          ❶
    if len(ids) == 1:                                        ❷
        matrix = qt.circuit.gate_expand_1toN(
            unitary, self.capacity, ids[0]
        )
    else:
        raise ValueError("Поддерживаются только однокубитовые унитарные матрицы.")
    self.register_state = matrix * self.register_state       ❸
```

❶ Приватный метод _apply на входе принимает аргумент unitary типа Qobj, представляющий применяемую унитарную операцию, и список int для указания индексов списка available_qubits, в котором мы хотим применить операцию. Пока что этот список будет содержать только один элемент, поскольку в нашем симуляторе мы имплементируем только однокубитовые операции. Мы ослабим это ограничение в следующей главе.

❷ Если мы хотим применить к кубиту в реестре однокубитовую операцию, то мы можем применить QuTiP для генерирования нужной нам матрицы. QuTiP делает это, применяя матрицу для нашей однокубитовой операции к правильному кубиту и применяя $\mathbb{1}$ в остальных случаях. Это делается для нас автоматически с помощью функции gate_expand_1toN.

❸ Теперь, когда у нас есть правильная матрица, на которую можно умножить весь реестр register_state, мы можем обновить значение этого реестра соответствующим образом.

Давайте перейдем к имплементации класса `SimulatedQubit`, представляющей то, как мы симулируем один кубит, учитывая наше знание о том, что он является частью устройства, имеющего несколько кубитов. Главное различие между одно- и многокубитовыми версиями класса `SimulatedQubit` заключается в том, что нам нужно, чтобы каждый кубит помнил свое «родительское» устройство и местоположение или идентификатор в этом устройстве, чтобы у нас была возможность ассоциировать состояние с реестром, а не с каждым кубитом. Это важно, как мы увидим в следующем разделе, когда возникнет потребность измерять кубиты в многокубитовом устройстве.

Листинг 5.10 simulator.py: однокубитовая операция на многокубитовом устройстве

```
class SimulatedQubit(Qubit):
    qubit_id: int
    parent: "Simulator"
```

```
def __init__(self, parent_simulator: "Simulator", id: int):      ❶
    self.qubit_id = id
    self.parent = parent_simulator

def h(self) -> None:
    self.parent._apply(H, [self.qubit_id])                       ❷

def ry(self, angle: float) -> None:
    self.parent._apply(qt.ry(angle), [self.qubit_id])            ❸

def x(self) -> None:
    self.parent._apply(qt.sigmax(), [self.qubit_id])
```

❶ **Для инициализации кубита нам требуется имя родительского симулятора (чтобы иметь возможность легко его ассоциировать) и индекс кубита в реестре симулятора. Затем __init__ задает эти атрибуты и сбрасывает кубит в состояние |0⟩.**

❷ **В целях имплеменирования операции h мы просим свойство parent класса SimulatedQubit (которое содержит экземпляр симулятора) использовать метод _apply для создания правильной матрицы, которая будет представлять операцию в полном реестре, а затем обновить register_state.**

❸ **Мы также можем передать параметризованную операцию qt.ry из QuTiP в _apply, чтобы повернуть наш кубит вокруг оси *Y* на угол angle.**

Отлично! Мы почти закончили модернизацию нашего симулятора, чтобы использовать пакет QuTiP и поддерживать несколько кубитов. В следующем далее разделе мы решим задачу симулирования операции измерения на многокубитовых состояниях.

5.1.2 Измерение: как измерить несколько кубитов?

> **ДЛЯ СПРАВКИ** Этот раздел является одним из самых сложных в книге. Пожалуйста, не волнуйтесь, если в первый раз он не будет для вас вполне понятным.

В некотором смысле измерение нескольких кубитов работает так же, как мы привыкли измерять однокубитовые системы. Мы по-прежнему можем использовать правило Борна для предсказания вероятности любого конкретного результата измерения. Например, давайте вернемся к состоянию $(|00\rangle + |11\rangle)/\sqrt{2}$, которое мы уже встречали несколько раз. Если бы мы измерили пару кубитов в этом состоянии, то получили бы либо «00», либо «11» в качестве наших классических результатов с равной вероятностью, поскольку оба имеют одинаковую амплитуду: $1/\sqrt{2}$.

Схожим образом мы по-прежнему будем требовать, чтобы при измерении одного и того же реестра дважды подряд мы получали один и тот же ответ. Например, если мы получим результат «00», то мы знаем, что кубиты остаются в состоянии $|00\rangle = |0\rangle \otimes |0\rangle$.

Однако все становится немного сложнее, если мы измеряем часть квантового реестра, не измеряя весь целиком. Давайте рассмотрим несколько примеров, чтобы понять, как это может работать. Снова беря $(|00\rangle + |11\rangle)/\sqrt{2}$ в качестве примера, если мы измеряем *только* первый

кубит и получаем «0», то мы знаем, что нам нужно снова получить тот же ответ при следующем измерении. Это может произойти только в том случае, если состояние преобразовывается в |00⟩ в результате наблюдения «0» на первом кубите.

С другой стороны, что произойдет, если мы измерим первый кубит из пары кубитов в состоянии |+⟩? Прежде всего полезно освежить свои знания о том, как выглядит |+⟩ при его записи в виде вектора.

Листинг 5.11 Представление состояния |++⟩ с помощью пакета QuTiP

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> ket_plus = hadamard_transform() * qt.basis(2, 0)                    ❶
>>> ket_plus
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket          ❷
Qobj data =
[[0.70710678]
 [0.70710678]]
>>> register_state = qt.tensor(ket_plus, ket_plus)                      ❸
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =                                                            ❹
[[0.5]
 [0.5]
 [0.5]
 [0.5]]
```

❶ Начнем с написания |+⟩ как $H|0\rangle$. В QuTiP мы используем функцию hadamard _transform, чтобы получить экземпляр Qobj для представления H, и используем basis(2, 0), чтобы получить Qobj, представляющий |0⟩.

❷ Мы можем распечатать ket_plus, чтобы получить список элементов в этом векторе; как и раньше, мы называем каждый из этих элементов амплитудой.

❸ Для представления состояния |+⟩ мы используем это |+⟩ = |+⟩ ⊗ |+⟩.

❹ Этот результат говорит нам о том, что |++⟩ имеет одинаковую амплитуду для каждого из четырех вычислительно-базисных состояний |00⟩, |01⟩, |10⟩ и |11⟩, подобно тому, как ket_plus имеет одинаковую амплитуду для каждого из вычислительно-базисных состояний |0⟩ и |1⟩.

Предположим, что мы измеряем первый кубит и получаем результат «1». В целях получения того же результата при следующем измерении состояние после измерения не может иметь никакой амплитуды на |00⟩ или |01⟩. Если мы сохраняем амплитуды только на |10⟩ и |11 ⟩ (третья и четвертая строки вектора, который мы рассчитали ранее), то получим, что состояние наших двух кубитов станет $(|10\rangle + |11\rangle)/\sqrt{2}$.

Откуда взялся $\sqrt{2}$?

Мы включили $\sqrt{2}$ в целях обеспечения того, чтобы все наши вероятности измерения по-прежнему в сумме составляли 1 при измерении второго кубита. Для того чтобы правило Борна имело хоть какой-то смысл, нам всегда нужно, чтобы сумма квадратов каждой амплитуды составляла 1.

Однако есть еще один способ записи этого состояния, которое мы можем проверить с помощью QuTiP.

Листинг 5.12 Представление состояния |1+⟩ с помощью пакета QuTiP

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> ket_0 = qt.basis(2, 0)
>>> ket_1 = qt.basis(2, 1)
>>> ket_plus = hadamard_transform() * ket_0                    (1)
>>> qt.tensor(ket_1, ket_plus)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.        ]
 [0.        ]
 [0.70710678]
 [0.70710678]]
```

(1) Напомним, что мы можем записать |+⟩ как $H|0\rangle$.

Это говорит нам о том, что если мы сохраним только те части состояния |+⟩, которые согласуются с получением результата «1» измерения первого кубита, то мы получим $|1\rangle = |1\rangle \otimes |+\rangle$. То есть со вторым кубитом в этом случае вообще ничего не происходит!

Упражнение 5.3: измерение другого кубита

Предположим, что в примере, в котором наши два кубита начинаются в состоянии |++⟩, вместо этого мы измерили второй кубит. Убедитесь, что независимо от того, какой результат мы получим, с состоянием первого кубита ничего не произойдет.

В целях выработки понимания того, что значит измерять часть реестра в более общем плане, мы можем применить еще одно понятие из линейной алгебры, именуемое *проекторами*.

ОПРЕДЕЛЕНИЕ *Проектор* – это произведение вектора состояния (кета или части |+⟩ бракета) и измерения (бра или части ⟨+| бракета), которое представляет нашу потребность в том, что *если* происходит определенный результат измерения, *то* мы должны преобразовать его в состояние, соответствующее этому измерению.

На рис. 5.4 приведен краткий пример однокубитового проектора. Определение проекторов на нескольких кубитах работает точно так же.

В пакете QuTiP мы записываем бра, соответствующий кету, с помощью метода `.dag()` (сокращение от *dagger* – кинжал, отсылка к математической нотации, которую мы увидим на рис. 5.4, где эта операция обозна-

чается символом кинжала). К счастью, даже если математика не проста, писать на Python не так уж и плохо, как мы увидим далее.

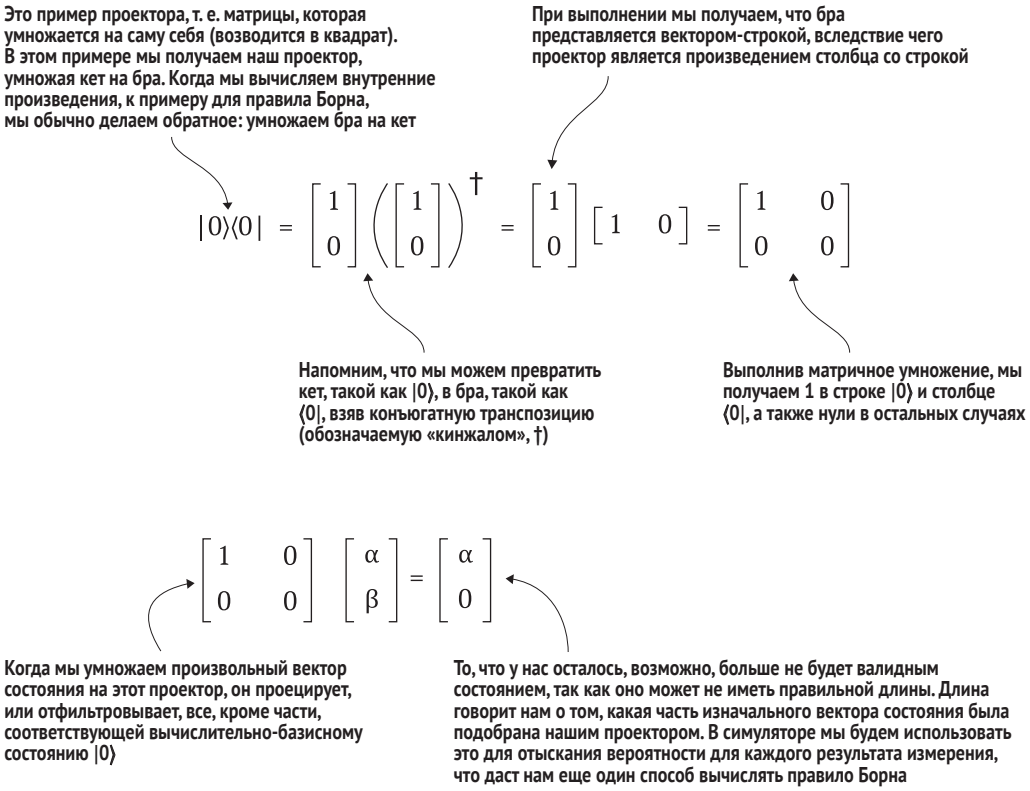


Рис. 5.4 Пример проектора, действующего на однокубитовое состояние

Листинг 5.13 simulator.py: измерение одиночных кубитов в реестре

```
def measure(self) -> bool:
    projectors = [                                                    (1)
        qt.circuit.gate_expand_1toN(                                  (2)
            qt.basis(2, outcome) * qt.basis(2, outcome).dag(),
            self.parent.capacity,
            self.qubit_id
        )
        for outcome in (0, 1)
    ]
    post_measurement_states = [
        projector * self.parent.register_state                        (3)
        for projector in projectors
    ]
    probabilities = [                                                 (4)
        post_measurement_state.norm() ** 2
        for post_measurement_state in post_measurement_states
```

```
        ]
        sample = np.random.choice([0, 1], p=probabilities)                    ❺
        self.parent.register_state = post_measurement_states[sample].unit()   ❻
        return bool(sample)

    def reset(self) -> None:                                                  ❼
        if self.measure(): self.x()
```

❶ Использует QuTiP для составления списка проекторов, по одному для каждого возможного результата измерения.
❷ Как и в листинге 5.9, использует функцию gate_expand_1toN для расширения каждого однобитового проектора до проектора, который действует на всем реестре.
❸ Использует каждый проектор для подбора частей состояния, которые согласуются с каждым результатом измерения.
❹ Длина того, что каждый проектор подбирает (написанная как метод .norm() в QuTiP), говорит нам о вероятности результата каждого измерения.
❺ Имея вероятности для каждого результата, мы можем выбрать результат, используя NumPy.
❻ Использует встроенный в QuTiP метод .unit() для обеспечения того, чтобы сумма вероятностей измерений по-прежнему равнялась 1.
❼ Если результат измерения равен |1⟩, то переворачивание с помощью инструкции x сбрасывает назад в |0⟩.

5.2 Игра CHSH: квантовая стратегия

Теперь, когда мы расширили симулятор для манипулирования несколькими кубитами, давайте обратимся к симулированию *квантовой стратегии* для наших игроков, которая приведет к вероятности выигрыша выше, чем это возможно с любой классической стратегией! См. рис. 5.5 для напоминания о том, как играть в игру CHSH.

У вас с Евой теперь есть квантовые ресурсы, поэтому давайте начнем с самого простого варианта: у каждого из вас есть по одному кубиту, выделенному из одного и того же устройства. Мы будем использовать симулятор для имплементирования этой стратегии, так что на самом деле это не проверка квантовой механики, а проверка того, что наш симулятор согласуется с квантовой механикой.

> **ПРИМЕЧАНИЕ** Мы не можем симулировать игроков, которые действительно находятся нелокально, так как части симулятора должны взаимодействовать так, чтобы эмулировать квантовую механику. Добросовестное симулирование квантовых игр и квантовых сетевых протоколов в таком ключе раскрывает множество интересных вопросов классической сетевой топологии, которые выходят далеко за рамки этой книги. Если вас интересуют симуляторы, больше предназначенные для использования в квантовых сетях, чем для квантовых вычислений, то мы рекомендуем ознакомиться с проектом SimulaQron (www.simulaqron.org), где можно получить дополнительную информацию.

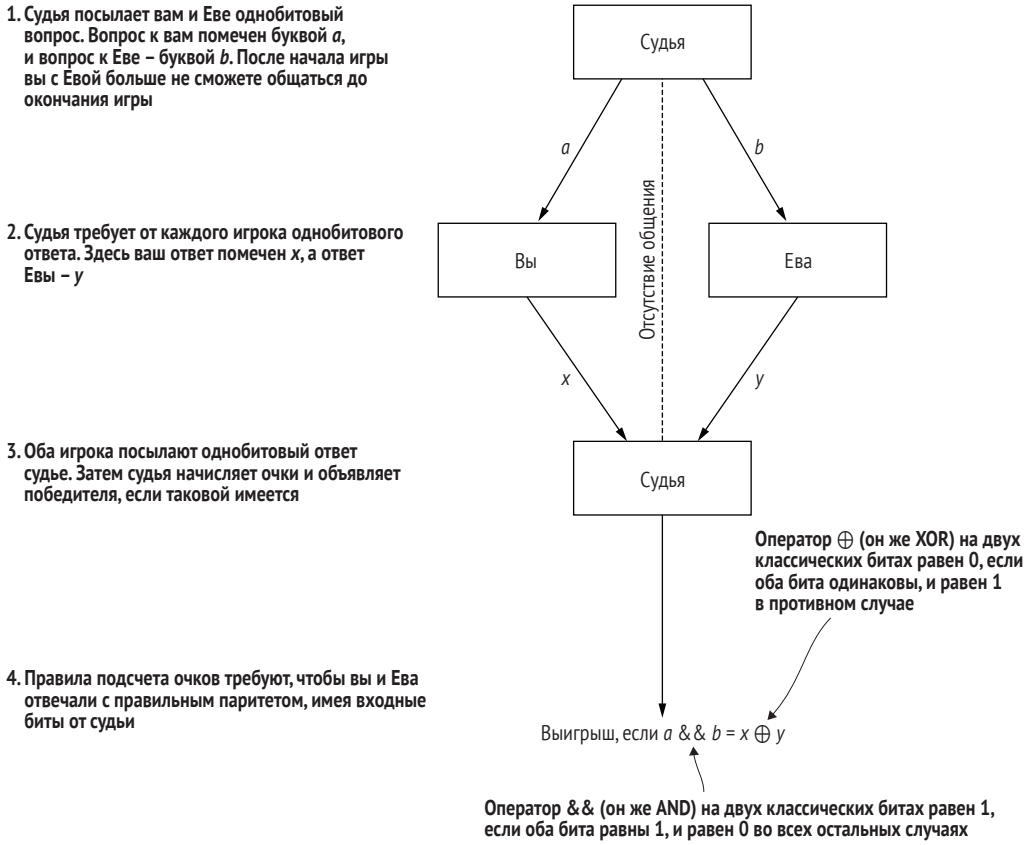


Рис. 5.5 Игра CHSH, нелокальная игра с двумя игроками и судьей. Судья задает каждому игроку вопрос в форме битового значения, а затем каждый игрок должен придумать, как ответить судье. Игроки выигрывают, если булево значение AND их ответов совпадает с классическим XOR вопросов судьи

Давайте посмотрим на частоту наших с Евой возможных выигрышей, если каждый из нас начнет с одного кубита и если эти кубиты начнутся в состоянии $(|00\rangle + |11\rangle)/\sqrt{2}$, которое мы встречали в этой книге уже несколько раз. Не беспокойтесь о том, как подготовить это состояние; мы узнаем, как это делается, в главе 6. А пока давайте просто посмотрим, что можно делать с кубитами в этом состоянии после того, как они у нас появятся.

Используя эти кубиты, мы можем сформировать новую квантовую стратегию для игры CHSH, которую мы видели в начале главы. Хитрость состоит в том, что мы с Евой можем применять операции к каждому нашему кубиту, как только получим соответствующие сообщения от судьи.

Как оказалось, операция `ry` для этой стратегии весьма полезна. Она позволяет нам и Еве задействовать небольшой компромисс между частотой наших выигрышей в случае, когда судья просит нас давать одина-

ковые ответы (случаи 00, 01 и 10), с целью добиться чуть лучшего в случае, когда нам нужно выдавать разные ответы (случай 11), как показано на рис. 5.6.

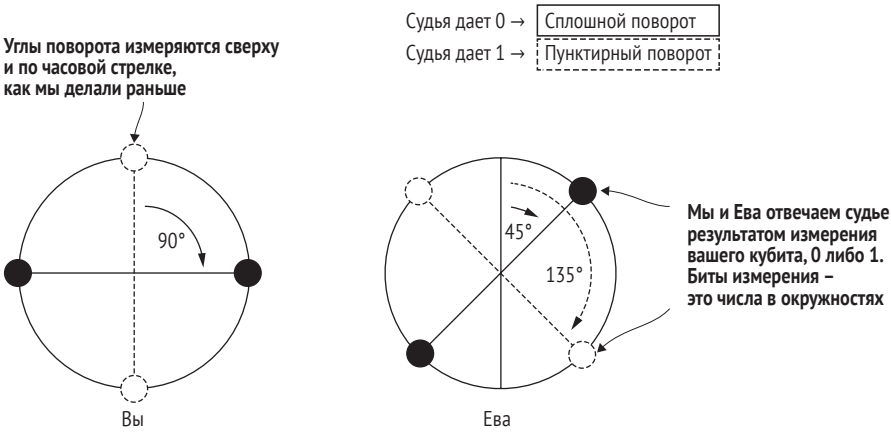


Рис. 5.6 Поворачивание кубитов для выигрыша в игре CHSH. Если мы получим от судьи 0, то должны повернуть кубит на 45°; а если мы получим 1, то должны повернуть кубит на 135°

Из этой стратегии мы видим, что и у нас, и у Евы есть довольно простое, прямолинейное правило того, что делать с нашими соответствующими кубитами, прежде чем мы их измерим. Если мы получим от судьи 0, то должны повернуть кубит на 45°, а если мы получим 1, то должны повернуть кубит на 135°. Если вам нравится табличный подход к данной стратегии, то в табл. 5.1 показано краткое изложение.

Таблица 5.1 Повороты, которые мы и Ева делаем с нашими кубитами в зависимости от входного бита, получаемого нами от судьи. Обратите внимание, что все они являются ry-поворотами, только на разные углы (преобразованные для ry в радианы)

Входное значение от судьи	Наш поворот	Поворот Евы
0	`ry(90 * np.pi / 180)`	`ry(45 * np.pi / 180)`
1	`ry(0)`	`ry(135 * np.pi / 180)`

Не волнуйтесь, если эти углы выглядят случайными. Мы можем убедиться, что они работают, с помощью нашего нового симулятора! В следующем ниже листинге используются новые функции, которые мы добавили в симулятор, чтобы написать квантовую стратегию.

Листинг 5.14 chsh.py: квантовая стратегия для CHSH, использующая два кубита

```
import qutip as qt
def quantum_strategy(initial_state: qt.Qobj) -> Strategy:
    shared_system = Simulator(capacity=2)                    ❶
```

```
    shared_system.register_state = initial_state
    your_qubit = shared_system.allocate_qubit()                    ❷
    eve_qubit = shared_system.allocate_qubit()

    shared_system.register_state = qt.bell_state()                 ❸
    your_angles = [90 * np.pi / 180, 0]                            ❹
    eve_angles = [45 * np.pi / 180, 135 * np.pi / 180]

    def you(your_input: int) -> int:
        your_qubit.ry(your_angles[your_input])                     ❺
        return your_qubit.measure()                                ❻

    def eve(eve_input: int) -> int:                                ❼
        eve_qubit.ry(eve_angles[eve_input])
        return eve_qubit.measure()

    return you, eve                                                ❽
```

❶ Для запуска квантовой стратегии нам нужно создать экземпляр класса QuantumDevice, в котором мы будем симулировать наши кубиты.
❷ Метки могут быть назначены каждому кубиту, когда мы выделяем их в совместную систему shared_system.
❸ Мы немного сжульничаем, чтобы установить состояние наших кубитов в запутанное состояние $(|00\rangle + |11\rangle)/\sqrt{2}$. В главе 6 мы увидим, как готовить это состояние с нуля и почему функция для подготовки этого состояния называется bell_state.
❹ Углы для поворотов, которые мы и Ева должны делать, основываясь на наших входных значениях от судьи.
❺ Стратегия игры в CHSH начинается с того, что мы поворачиваем наш кубит, основываясь на входном классическом бите от судьи.
❻ Значение классического бита, которое наша стратегия возвращает, является значением бита, которое мы получаем при измерении нашего кубита.
❼ Стратегия Евы похожа на нашу; просто для своего первоначального поворота она использует другие углы.
❽ Подобно нашей классической стратегии, квантовая стартегия quantum_strategy возвращает кортеж функций, представляющих наши с Евой индивидуальные действия.

Теперь, когда мы имплементировали версию `quantum_strategy` на Python, давайте посмотрим на частоту наших возможных выигрышей, используя функцию `est_win_probability` игры CHSH.

Листинг 5.15 Выполнение CHSH с нашей новой стратегией quantum_strategy

```
>>> est_win_probability(quantum_strategy)                          ❶
0.832
```

❶ При попытке это сделать вы, возможно, получите немного больше или меньше 85 %, потому что под капотом вероятность выигрыша оценивается с помощью биномиального распределения. В данном примере мы ожидаем, что полосы ошибок составят около 1.5 %.

Оценочная вероятность выигрыша в 83.2 % в листинге 5.15 выше, чем при использовании любой классической стратегии. Это означает, что мы

с Евой можем начать выигрывать в игре CHSH чаще, чем любые другие классические игроки, – потрясающе! Вместе с тем эта стратегия показывает пример того, насколько состояния наподобие $(|00\rangle + |11\rangle)/\sqrt{2}$ являются важными ресурсами, предоставляемыми квантовой механикой.

> **ПРИМЕЧАНИЕ** Состояния наподобие $(|00\rangle + |11\rangle)/\sqrt{2}$ называются *запутанными*, потому что их нельзя записать как тензорное произведение однокубитовых состояний. По ходу мы увидим еще много примеров запутанности, но запутанность – это одна из самых удивительных и забавных вещей, которые мы можем использовать при написании квантовых программ.

Как мы увидели в этом примере, запутанность позволяет нам создавать в наших данных корреляции, которые могут использоваться в наших интересах, когда мы хотим получать из квантовых систем полезную информацию.

Скорость света все еще актуальна

Если вы читали о теории относительности (если нет, не беспокойтесь), то, возможно, слышали, что невозможно передавать информацию быстрее скорости света. Из того, что мы узнали о запутанности до сих пор, может показаться, что квантовая механика это нарушает, но, как оказалось, запутанность *никогда* не может использоваться для передачи сообщения по нашему выбору сама по себе. Вместе с использованием запутывания нам всегда нужно посылать что-то еще. Это означает, что скорость света все еще ограничивает скорость распространения информации во Вселенной – фу!

Запутанность вовсе не является странной или непонятной, она является прямым результатом того, что мы уже узнали о квантовых вычислениях: это прямое следствие *линейности* квантовой механики. Если мы можем подготовить двухкубитовый реестр в состоянии $|00\rangle$ и состоянии $|11\rangle$, то мы также можем подготовить состояние в линейной комбинации этих двух, например в $(|00\rangle + |11\rangle)/\sqrt{2}$.

Поскольку запутанность является прямым результатом линейности квантовой механики, игра CHSH также дает нам отличный способ проверить правильность квантовой механики (или в лучшем случае наши данные могут это показать). Давайте вернемся к вероятности выигрыша в листинге 5.15. Если мы проведем эксперимент и увидим вероятность выигрыша 83.2 %, то это говорит нам о том, что наш эксперимент не мог быть чисто классическим, поскольку мы знаем, что классическая стратегия может выиграть не более, чем в 75 % случаях. Этот эксперимент проводился много раз на протяжении всей истории и является частью демонстрации того, откуда мы знаем, что наша Вселенная не просто классическая – нам нужна квантовая механика, чтобы ее описывать.

ПРИМЕЧАНИЕ В 2015 году в одном эксперименте два игрока в игре CHSH были разделены более чем на километр!

Симулятор, о котором мы писали в этой главе, дает нам все необходимое, чтобы понять принцип работы подобных экспериментов. Теперь мы можем двинуться вперед и использовать квантовую механику и кубиты для выполнения потрясающих вещей, вооруженные знанием того, что квантовая механика действительно показывает нам принцип работы нашей Вселенной.

Самотестирование: приложение для нелокальных игр

Этот пример дает намек на еще одно приложение для нелокальных игр: если мы можем играть *и выигрывать* нелокальную игру с Евой, то по пути мы должны создать что-то, что можем использовать для отправки квантовых данных. Такого рода понимание приводит к идеям, именуемым *квантовым самотестированием*, когда мы заставляем части устройства играть в нелокальные игры с другими частями устройства, чтобы убедиться, что устройство работает правильно.

Резюме

- Мы можем использовать пакет QuTiP в помощь для работы с тензорными произведениями и другими вычислениями, необходимыми для написания многокубитового симулятора на Python.
- Класс Qobj в QuTiP отслеживает многие полезные свойства состояний и операций, которые мы хотим симулировать.
- Мы с Евой можем использовать квантовую стратегию игры в CHSH, где мы делимся между собой парой запутанных кубитов, перед тем как начать игру.
- Написав игру CHSH как квантовую программу, мы можем доказать, что игроки, использующие запутанные пары кубитов, могут выигрывать чаще, чем игроки, использующие только классические компьютеры, что согласуется с нашим пониманием квантовой механики.

6 Телепортация и запутанность: перемещение квантовых данных с места на место

Эта глава охватывает следующие ниже темы:

- перемещение данных по квантовому компьютеру с использованием классического и квантового контроля;
- визуализирование однокубитовых операций сферой Блоха;
- предсказывание результата двухкубитовых операций и операций Паули.

В предыдущей главе мы добавили поддержку нескольких кубитов в наш симулятор квантовых устройств с помощью пакета QuTiP. Это позволило нам сыграть в игру CHSH и показать, что наше понимание квантовой механики согласуется с тем, что мы наблюдаем в реальном мире.

В этой главе мы узнаем, как перемещать данные между разными людьми или реестрами в квантовом устройстве. Мы рассмотрим влияние таких вещей, как теорема о запрете клонирования, на то, как мы управляем нашими данными в квантовом устройстве. Мы также сможем проверить уникальный квантовый протокол, который наше квантовое устройство может выполнять. Этот протокол называется *телепортацией* и служит для перемещения данных (в отличие от копий).

6.1 Перемещение квантовых данных

Подобно классическим вычислениям, иногда в квантовом компьютере есть какие-то данные *здесь*, которые мы очень хотели бы иметь где-то *там*. Классически эту задачу легко решить путем копирования данных; но, как мы видели в главах 3 и 4, теорема о запрете клонирования в общем случае означает, что мы *не можем* копировать данные, хранящиеся в кубитах.

> **Перемещение данных классически**
>
> В некоторых частях классических вычислений мы сталкиваемся с той же самой проблемой невозможности копирования информации, но по другим причинам. Копирование данных в многопоточном (многонитевом) приложении может приводить к возникновению едва уловимых гоночных ситуаций, тогда как соображения производительности могут побуждать нас уменьшать объем копируемых данных.
>
> Решение, используемое во многих классических языках (например, в C++11 и Rust), сосредоточивается на *перемещении* данных. Мышление в терминах перемещения данных полезно и в квантовых вычислениях, хотя мы будем имплементировать перемещения совсем по-другому.

Итак, что можно сделать, если мы хотим переместить данные в квантовом устройстве? К счастью, существует масса разных способов перемещения квантовых данных вместо их копирования. В этой главе мы рассмотрим некоторые из этих подходов и добавим последние несколько функций в наш симулятор, чтобы их имплементировать. Давайте перейдем к обмену квантовой информацией!

Предположим, у Евы есть несколько кубитов, кодирующих данные, которыми она хотела бы поделиться с нами:

> *Эй, игрок! У меня есть кое-какая квантовая информация, которой я хочу с тобой поделиться. Могу я переслать ее тебе?*

Здесь Ева имеет в виду инструкцию `swap` – эта инструкция немного отличается от инструкций, которые мы видели до сих пор, в том, что она оперирует на *двух* кубитах сразу. В отличие от нее, каждая операция, которую мы видели до сих пор, оперирует только на одном кубите за раз.

Глядя на то, что делает инструкция `swap`, понимаешь всю наглядность ее названия: она буквально меняет местами состояние двух кубитов в одном реестре. Например, предположим, что у нас есть два кубита в состоянии $|01\rangle$. Если мы применим инструкцию `swap` для обоих кубитов, то наш реестр теперь будет находиться в состоянии $|10\rangle$. Давайте рассмотрим пример использования встроенной в пакет QuTiP матрицы `swap`.

Листинг 6.1 Использование swap-пакета QuTiP на состоянии |+0⟩ для получения состояния |0+⟩

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> ket_0 = qt.basis(2, 0)
>>> ket_plus = hadamard_transform() * ket_0
>>> initial_state = qt.tensor(ket_plus, ket_0)
>>> initial_state                                                    (1)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678]                                                       (2)
 [0.        ]
 [0.70710678]
 [0.        ]]
>>> swap_matrix = qt.swap()                                         (3)
>>> swap_matrix * initial_state                                     (4)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678]                                                       (5)
 [0.70710678]
 [0.        ]
 [0.        ]]
>>> qt.tensor(ket_0, ket_plus)                                      (6)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678]
 [0.70710678]
 [0.        ]
 [0.        ]]
```

(1) Использует qt.basis, hadamard_transform и qt.tensor с целью определения переменной для нашего старого друга из главы 5: вектора состояния |+0⟩.

(2) Как мы видели в главе 4, это состояние имеет равные амплитуды на вычислительно-базисных состояниях |00⟩ и |10⟩.

(3) Получает копию унитарной матрицы для инструкции swap путем вызова qt.swap.

(4) Точно так же, как мы симулировали однокубитовые операции, мы можем умножить наше состояние на унитарную матрицу, чтобы найти состояние нашего двухкубитового реестра после применения инструкции swap.

(5) При таком выполнении мы в конечном итоге будем в суперпозиции между |00⟩ и |01⟩, а не между |00⟩ и |10⟩.

(6) Оперативно проверяет, что результат применения инструкции swap на реестре из двух кубитов, которые начинаются в состоянии |+0⟩, равен |0+⟩.

Глядя на листинг 6.1, мы видим, что инструкция `swap` сделала в значительной степени то, что предполагает ее название. В частности, `swap` перевела два кубита, которые начали в состоянии |+0⟩, в состояние |0+⟩. В более общем случае мы можем разобраться в том, что делает инструкция `swap`, посмотрев на унитарную матрицу, которую мы использовали для ее симулирования.

Листинг 6.2 Унитарная матрица для инструкции swap

```
>>> import qutip as qt
>>> qt.swap()
Quantum object: dims = [[2, 2], [2, 2]],
➥ shape = (4, 4), type = oper, isherm = True          ❶
Qobj data =
[[1. 0. 0. 0.]          ❷
 [0. 0. 1. 0.]          ❸
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]]         ❹
```

❶ **Унитарная матрица, которую мы используем для симулирования инструкции swap, представлена матрицей 4×4, потому что она действует на двухкубитовые состояния.**

❷ **Каждый столбец этой унитарной матрицы говорит нам о том, что происходит с одним из вычислительно-базисных состояний; здесь инструкция swap ничего не делает с кубитами, которые начинаются в состоянии $|00\rangle$.**

❸ **С другой стороны, состояния $|01\rangle$ и $|10\rangle$ меняются местами инструкцией swap.**

❹ **Инструкция swap также оставляет нетронутым $|11\rangle$.**

Каждый столбец этой унитарной матрицы сообщает нам о том, что происходит с одним из вычислительно-базисных состояний. Например, первый столбец говорит о том, что состояние $|00\rangle$ соотнесено с вектором [[1], [0], [0], [0]], который мы распознаем как $|00\rangle$.

> **ПРИМЕЧАНИЕ** Унитарная матрица для инструкции `swap` *не может* быть записана как тензорное произведение любых двух однокубитовых унитарных матриц. То есть мы не можем понять, что конкретно `swap` делает, беря по одному кубиту за раз, – нам необходимо выяснить, что она делает с состоянием пары кубитов, на которые инструкция `swap` воздействует.

На рис. 6.1 показано, что мы можем увидеть работу `swap` в целом, независимо от того, в каком состоянии начинаются наши два кубита.

Напомним, что в главе 2 мы узнали, что унитарная матрица во многом похожа на таблицу истинности. То есть унитарные матрицы, подобные тем, которые мы получаем из `qt.swap`, полезны тем, что они помогают нам симулировать то, что делает инструкция `swap`. Однако, подобно тому как классический сумматор не является его таблицей истинности, полезно помнить, что эти унитарные матрицы не являются квантовыми программами, но являются инструментами, которые мы используем для симулирования работы квантовых программ.

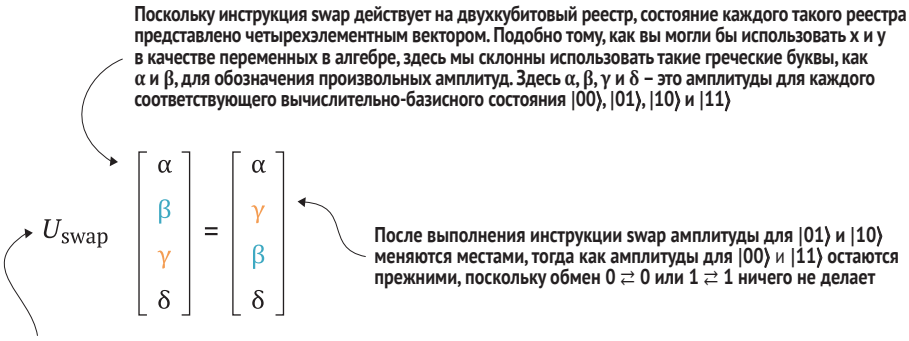


Рис. 6.1 Двухкубитовая операция swap меняет местами состояния двух кубитов в реестре. Мы видим это в показанном здесь общем примере, потому что члены, описывающие состояния |01⟩ и |10⟩, меняются местами. Два других – нет, поскольку мы не можем понять, когда два кубита находятся в одном и том же состоянии

Упражнение 6.1: обмен между вторым и третьим кубитами в реестре

Допустим, у вас есть реестр с тремя кубитами в состоянии |01+⟩. Используя пакет QuTiP, запишите это состояние, а затем выполните обмен между вторым и третьим кубитами, чтобы ваш реестр находился в состоянии |0+1⟩.

Подсказка: поскольку с первым кубитом ничего не произойдет, обязательно возьмите тензорное произведение матрицы тождественности и `qt.swap`, чтобы построить правильную операцию для вашего реестра.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

А в это время Ева уже едва не умирает от ожидания отправки своих кубитов. Давайте добавим в наш симулятор то, что нам нужно, чтобы больше не заставлять ее ждать!

6.1.1 Обменные операции в симуляторе

Симулятор, над которым мы работали в главе 4, нуждается всего в паре настроек, чтобы иметь возможность использовать двухкубитовые операции, такие как `swap`. Изменения, которые нам необходимо внести, таковы:

- модифицировать `_apply` для работы с двухкубитовыми операциями;
- добавить `swap` и другие двухкубитовые инструкции;
- добавить остальные однокубитовые поворотные инструкции.

Как мы увидели в главе 4, если матрица воздействует на однокубитовый реестр, то мы можем использовать пакет QuTiP, чтобы применить эту матрицу к реестру с произвольным числом кубитов с помощью функции `gate_expand _1toN` указанного пакета. Это предусматривает взятие тензорного произведения операторов тождественности на каждом кубите, кроме кубитов, с которыми мы работаем.

Точно таким же образом мы можем вызвать функцию `gate_expand_2toN` пакета QuTiP и превратить двухкубитовые унитарные матрицы в матрицы, которые мы можем использовать для симулирования того, как двухкубитовые операции, такие как `swap`, преобразовывают состояние всего реестра. Давайте теперь добавим это в наш симулятор (листинг 6.3).

> **ДЛЯ СПРАВКИ** В этой главе мы внесли несколько небольших изменений в исходный код, чтобы сделать распечатываемый результат немного приятнее на вид. Эти изменения, а также все примеры из этой и других глав, находятся в репозитории данной книги на GitHub: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

Листинг 6.3 simulator.py: применение двухкубитовых унитарных матриц

```
def _apply(self, unitary: qt.Qobj, ids: List[int]):
    if len(ids) == 1:
        matrix = qt.circuit.gate_expand_1toN(unitary,
                                             self.capacity, ids[0])
    elif len(ids) == 2:                                              (1)
        matrix = qt.circuit.gate_expand_2toN(unitary,                (2)
                                             self.capacity, *ids)
    else:
        raise ValueError("Поддерживаются только одно- и двухкубитовые унитарные
                         ➥ матрицы.")
    self.register_state = matrix * self.register_state
```

(1) В целях симулирования двухкубитовых операций нам нужны два индекса для кубитов в реестре: по одному для каждого кубита, на которые наша инструкция воздействует.

(2) Вызов gate_expand_2toN очень похож на вызов gate_expand_1toN, за исключением того, что вместо матрицы 2×2 мы передаем матрицу 4×4.

Мы видели, что пакет QuTiP предлагает функцию `swap`, которая дает нам копию унитарной матрицы, симулирующей инструкцию `swap`. Указанная функция может использоваться для довольно быстрого добавления инструкции `swap` в наш симулятор, задействуя изменения, которые мы внесли в `Simulator._apply`.

Листинг 6.4 simulator.py: добавление инструкции swap

```
def swap(self, target: Qubit) -> None:
    self.parent._apply(
        qt.swap(),                                  (1)
        [self.qubit_id, target.qubit_id]            (2)
    )
```

(1) Для получения унитарной матрицы 4×4 нам нужно перейти в «apply»; мы просто используем функцию qt.swap, которую уже встречали в этой главе несколько раз.

(2) Нам нужно обеспечить передачу индексов для обоих кубитов, между которыми мы хотим выполнить обмен, чтобы функция gate_expand_2toN правильно применила унитарную матрицу для нашей новой инструкции swap к состоянию всего реестра.

Раз уж мы работаем с симулятором, давайте добавим еще одну инструкцию, чтобы легче распечатывать его состояние без необходимости доступа к его внутренностям.

Листинг 6.5 simulator.py: добавление инструкции dump

```
def dump(self) -> None:
    print(self.register_state)
```

Благодаря ей мы сможем просить симулятор помогать нам в отладке квантовых программ, но так, чтобы иметь возможность безопасно ее изымать для устройств, которые его не поддерживают (например, фактического квантового оборудования).

ДЛЯ СПРАВКИ Напомним, что кубит – это не состояние. Состояние – это просто удобный способ представления того, как будет вести себя квантовая система.

Имея все эти изменения на своих местах, мы готовы задействовать инструкцию swap. Давайте повторим эксперимент, в котором мы выполнили обмен между двумя кубитами, начинающимися в состоянии |0+⟩, чтобы преобразовать их в состояние |+0⟩.

ДЛЯ СПРАВКИ Как всегда, полные примеры файлов см. в репозитории этой книги на GitHub: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

Листинг 6.6 Тестирование инструкции swap на состоянии |0+⟩

```
>>> from simulator import Simulator
>>> sim = Simulator(capacity=2)
>>> with sim.using_register(n_qubits=2) as (you, eve):                    (1)
...     eve.h()
...     sim.dump()
```

```
...     you.swap(eve)
...     sim.dump()
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket        ❷
Qobj data =
[[0.70710678]
 [0.70710678]
 [0.        ]
 [0.        ]]
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket        ❸
Qobj data =
[[0.70710678]
 [0.        ]
 [0.70710678]
 [0.        ]]
```

❶ **Поскольку в этой главе мы будем много работать с многокубитовыми реестрами, мы добавили новый удобный метод, который позволяет нам выделять несколько кубитов сразу.**

❷ **Первый дамп поступает от нашего первого вызова и подтверждает, что eve.h() подготовила кубиты в состоянии $|0+\rangle$.**

❸ **После вызова you.swap(eve) наш кубит заканчивается в состоянии $|+\rangle$, а кубит Евы – в том, в каком начали мы: в состоянии $|0\rangle$.**

Отлично! Теперь у нас есть способ делиться квантовыми данными с Евой! По крайней мере, до тех пор, пока мы используем *одно квантовое устройство*, благодаря чему мы имеем возможность применять инструкцию swap к обоим нашим кубитам одновременно.

Что произойдет, если мы захотим обмениваться квантовой информацией *между* устройствами? К счастью, квантовые вычисления дают нам возможность отправлять кубиты, обмениваясь классическими данными, только (?) если мы оба начинаем с некоторой запутанности между нашими кубитами. Как и многое в квантовых вычислениях, этот технический прием получил причудливое название: *квантовая телепортация*. Однако не позволяйте названию этого приема вас обманывать. Если добраться до самой сути, то в телепортации используется то, что мы узнали в главе 4 о том, что позволяет нам делиться квантовыми данными полезным образом. На рис. 6.2 показан список шагов программы телепортации.

Самое замечательное в телепортации то, что хотя нам с Евой все еще нужно выполнять несколько двухкубитовых операций между нашими соответствующими кубитами, Ева может выбирать данные, которые она хочет нам отправить, *после* того как мы выполним эти операции. Это означает, что мы могли бы подготавливать запутанные кубиты до того, как нам понадобится обмен квантовыми данными, и просто использовать их по мере необходимости.

Используя симулятор, который мы разрабатывали на протяжении последних нескольких глав, мы могли бы написать телепортацию с помощью квантовой программы, подобной той, что показана в следующем ниже листинге.

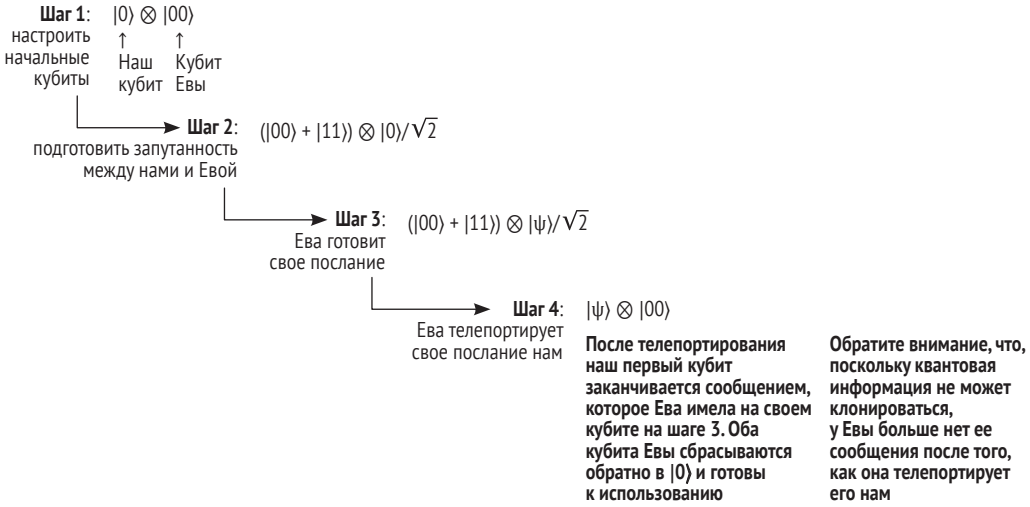


Рис. 6.2 Шаги программы телепортации. Мы готовим и запутываем реестр кубитов, а затем Ева может подготовить и телепортировать свое состояние нам. Обратите внимание, что у нее не будет этого состояния после телепортации

Листинг 6.7 Программа телепортации на Python

```
def teleport(msg : Qubit, here : Qubit, there : Qubit) -> None:
    here.h()
    here.cnot(there)

    msg.cnot(here)
    msg.h()

    if msg.measure(): there.z()
    if here.measure(): there.x()

    msg.reset()
    here.reset()
```

Однако в этой программе есть несколько новых инструкций. В остальной части данной главы мы увидим другие порции кода, необходимые для выполнения квантовой телепортации с помощью нашего симулятора.

6.1.2 Какие еще существуют двухкубитовые вентили?

Как вы можете догадаться, `swap` не единственная двухкубитовая операция. И действительно, как показано в листинге 6.7, чтобы привести телепортацию в рабочее состояние, нам нужно добавить в симулятор еще одну двухкубитовую инструкцию, именуемую `cnot`. Инструкция `cnot` делает нечто похожее на `swap`, за исключением того, что она переключает вычислительно-базисные состояния |10⟩ и |11⟩ вместо состояний |01⟩

и |10⟩. На это можно взглянуть и по-другому – как на то, что `cnot` переворачивает второй кубит, *контролируемый* на состоянии первого кубита, равном |1⟩. Отсюда и происходит название `cnot`: оно сокращенно означает «контролируемый» NOT[1].

ДЛЯ СПРАВКИ Мы часто называем первый переданный инструкции `cnot` кубит *контрольным кубитом*, а второй кубит – *целевым кубитом*. Однако, как мы увидим в главе 7, в этих названиях есть некоторая тонкость.

Давайте сразу перейдем к делу и посмотрим, как работает инструкция `cnot`, применив ее к прекрасному примеру, состоянию |+0⟩:

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> ket_0 = qt.basis(2, 0)
>>> ket_plus = hadamard_transform() * ket_0
>>> initial_state = qt.tensor(ket_plus, ket_0)                          (1)
>>> qt.cnot() * initial_state                                           (2)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket   (3)
Qobj data =
[[0.70710678]
 [0.        ]
 [0.        ]
 [0.70710678]]
>>> qt.cnot()
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
     isherm = True                                                    (4)
Qobj data =
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]]
```

(1) Инициализирует два кубита так, чтобы они начинали в состоянии $|+0\rangle = (|00\rangle + |10\rangle)/\sqrt{2}$.
(2) Пакет QuTiP обеспечивает унитарную матрицу для инструкции cnot в виде функции qt.cnot.
(3) Инструкция cnot оставляет наши кубиты в состоянии $(|00\rangle + |11\rangle)/\sqrt{2}$.
(4) Матрица для cnot соотносит вычислительно-базисное состояние |10⟩ с |11⟩ и наоборот, как мы и ожидали из описания.

ПРИМЕЧАНИЕ Инструкция `cnot` отличается от инструкций `if` классических языков программирования тем, что инструкция `cnot` сохраняет суперпозицию. Если бы мы хотели использовать инструкцию `if`, то нам пришлось бы измерить контрольный кубит, что привело бы к коллапсу любой суперпозиции на контрольном кубите. На самом деле мы будем использовать как инструкции `cnot`, так и инструкции `if` в зависимости от результатов измере-

[1] Вариант перевода: контролируемый NOT-*вентиль* (Controlled NOT gate). См. https://en.wikipedia.org/wiki/Controlled_NOT_gate. – *Прим. перев.*

ний, когда мы будем писать нашу программу телепортации в конце этой главы, – и то, и другое может быть полезно! В главах 8 и 9 мы рассмотрим отличие контролируемых операций от инструкций `if` подробнее.

На рис. 6.3 показано, как инструкция `cnot` оперирует на двухкубитовых состояниях в общем случае. Пока что, однако, мы распознаем выходное состояние, которое мы получили в приведенном выше фрагменте кода, применив `cnot` на двух кубитах в состоянии $|0\rangle$, как запутанное[1], которое нам было нужно для игры CHSH из главы 4, $(|00\rangle + |11\rangle)/\sqrt{2}$.

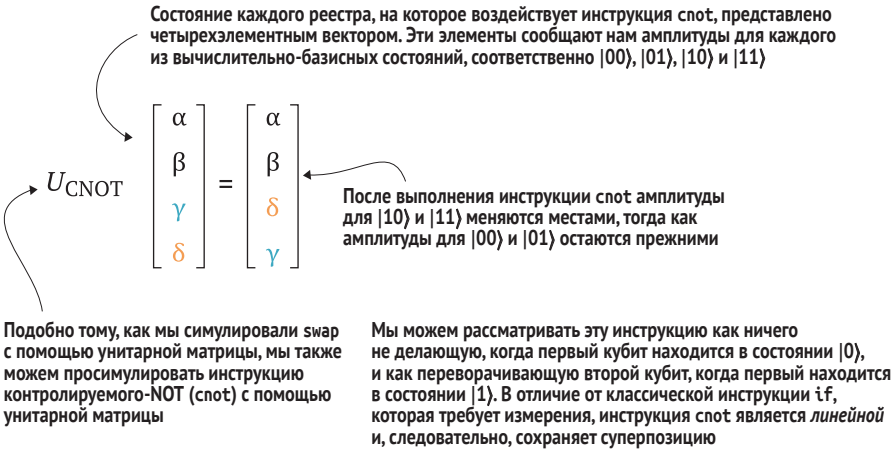


Рис. 6.3 Двухкубитовая операция `cnot` применяет операцию NOT, обусловленную состоянием контрольного кубита

Это означает, что у нас есть все необходимое для написания квантовой программы, которая запутывает два кубита, начинающихся в состоянии $|00\rangle$. Нам осталось лишь добавить инструкцию `cnot` в симулятор, точно так же, как мы добавили ранее `swap`.

Листинг 6.8 simulator.py: добавление инструкции `cnot`

```
def cnot(self, target: Qubit) -> None:
    self.parent._apply(
        qt.cnot(),
        [self.qubit_id, target.qubit_id]
    )
```

Теперь мы можем написать программу для подготовки двух кубитов в запутанной паре:

[1] В качестве напоминания: состояния, подобные $(|00\rangle + |11\rangle)/\sqrt{2}$, называются запутанными, т. к. их невозможно записать в виде тензорного произведения однокубитовых состояний. – *Прим. перев.*

```
>>> from simulator import Simulator
>>> sim = Simulator(capacity=2)
>>> with sim.using_register(2) as (you, eve):
...     eve.h()
...     eve.cnot(you)
...     sim.dump()
...
Quantum object: dims = [[2, 2], [1, 1]],
➥ shape = (4, 1), type = ket
Qobj data =
[[0.70710678]
 [0.        ]
 [0.        ]
 [0.70710678]]
```

В этой точке полезно сделать паузу (извини, Ева!) и поразмыслить о том, что мы только что сделали. В главе 4, когда мы симулировали игру CHSH с Евой, нам пришлось «сжульничать», допустив, что мы и Ева можем получать доступ к двум кубитам, которые волшебным образом начинают в запутанном состоянии $(|00\rangle + |11\rangle)/\sqrt{2}$. Теперь, однако, мы точно понимаем, как мы с Евой можем подготовить эту запутанность, выполнив еще одну квантовую программу, перед тем как играть в CHSH. Инструкция `h` готовит нужную нам суперпозицию, тогда как новая инструкция `cnot` позволяет нам подготовить запутывание с Евой. Эта запутанность «делится» данной суперпозицией между нашими двумя кубитами. (В конце концов, кушаешь сам – поделись с товарищем.)

Точно так же, как подготовка запутывания между нами и Евой выражала нашу подготовку к игре CHSH в главе 4, этот первый шаг нужен нам для того, чтобы Ева могла телепортировать нам свои квантовые данные. Этот факт делает `cnot` очень важной инструкцией в будущем.

Однако, возвращаясь к Еве, следующий ее шаг по телепортированию нам своих данных состоит в использовании одной из четырех однокубитовых операций для *декодирования* квантовых данных, которые она нам посылает (вспомните рис. 6.2). Давайте обратимся к ним далее.

6.2 Все одиночные (однокубитовые) повороты

Последнее, что нам потребуется для программирования квантовой телепортации, – это применить коррекцию, основываясь на классических данных, которые нам посылает Ева. Для этого нам нужна пара новых однокубитовых инструкций. Поэтому полезно вернуться к рисункам, которые мы использовали, чтобы изобразить квантовые инструкции в виде поворотов, потому что мы, возможно, немного сжульничали. Мы пока что изобразили наши кубиты в любом положении на окружности, но на самом деле нам для нашей модели кубита не хватает размерности. Состояние *одиночного* кубита представляется любой точкой на поверхности сферы, обычно именуемой *сферой Блоха*.

Только для одиночных кубитов!

Этот (и предыдущий) подход к визуализации состояния кубита работает *только* в том случае, если данный кубит не запутан с другими кубитами. Это то же самое, как если бы сказать, что мы не можем легко визуализировать многокубитовое состояние. Даже попытка визуализировать состояние двухкубитового реестра с запутанностью предусматривала бы рисование изображений в семи размерностях. В то время как 7D, возможно, отлично подошло бы для рекламы поездки на Ниагарский водопад, рисовать полезные рисунки в таком виде гораздо сложнее.

Окружность, с которой мы знакомы, на самом деле была просто срезом сферы, и все повороты, которые мы делали, приводили к состояниям, по-прежнему находящимся на этой окружности. На рис. 6.4 показано, как предыдущая модель кубита соотносится со сферой Блоха.

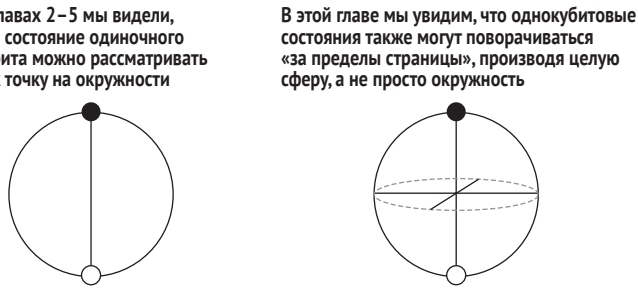


Рис. 6.4 Сравнение нашей предыдущей модели кубита (точки на окружности) и сферы Блоха. Сфера Блоха – это более общая модель состояния одиночного кубита. Нам требуется еще одна размерность, чтобы уловить это состояние кубита, представленного вектором комплексных чисел, но он работает только для одиночных кубитов

ДЛЯ СПРАВКИ Из того факта, что мы показали ось Z и ось X, вы могли бы заключить, что ось Y где-то прячется! Как оказалось, при переходе от окружности к сфере ось, которая выходит за пределы страницы, часто именуется «осью Y».

Когда мы впервые вводили векторное представление кубитового состояния в главе 2, вы, возможно, помните, что амплитуды в каждом векторе были *комплексными числами*. В остальной части этой главы мы увидим, что в общем случае при использовании поворотных инструкций для преобразования состояния одиночного кубита мы получаем комплексные числа. Комплексные числа являются чрезвычайно полезным инструментом для отслеживания поворотов и, следовательно, играют большую роль в квантовых вычислениях. В первую очередь они помогают нам понимать углы и фазы между разными квантовыми состояниями. Не волнуйтесь, если вы немного подзабыли комлексные числа, так

как у вас будет много возможностей с ними попрактиковаться на протяжении всей остальной части книги.

6.2.1 Привязка поворотов к координатам: операции Паули

Давайте на рис. 6.5 воспользуемся моментом, чтобы провести быструю ревизию пары однокубитовых операций, которые мы встречали до сих пор: `x` и `ry`.

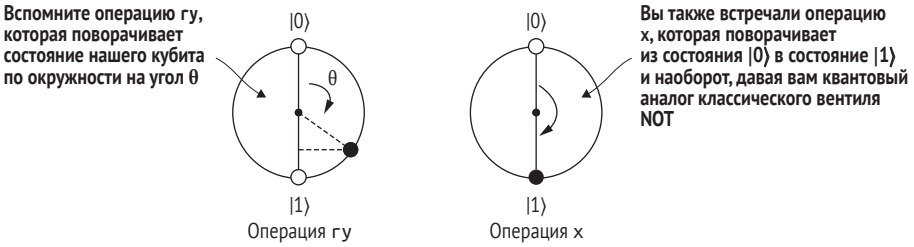


Рис. 6.5 Иллюстрации того, что `x` и `ry` делают с кубитом. Мы встречали обе эти операции в предыдущих главах. Инструкция `ry` поворачивает состояние нашего кубита на угол тета, а `x` переводит кубиты из состояния |0⟩ в состояние |1⟩ и наоборот

Теперь, когда мы знаем, что состояние нашего кубита может поворачиваться на поверхности сферы, какие другие повороты помогают нам поворачивать состояние из плоскости? Мы можем добавить поворот вокруг линии между состояниями |+⟩ и |−⟩. Эта линия условно называется осью *X*, чтобы отличить ее от оси *Z*, соединяющей состояния |0⟩ и |1⟩. Функция `rx` в пакете QuTiP дает нам объект класса `Qobj`, инкапсулирующий матрицу поворота для поворота по оси *X*.

Листинг 6.9 Использование встроенной в QuTiP функции `qt.sigmaz`

```
>>> import qutip as qt
>>> import numpy as np
>>> qt.rx(np.pi).tidyup()                                    ❶
Quantum object: dims = [[2], [2]], shape = (2, 2),
➥ type = oper, isherm = False
Qobj data =
[[ 0.+0.j 0.-1.j]                                            ❷
 [ 0.-1.j 0.+0.j]]
```

❶ **Экземпляры класса `Qobj` имеют метод очистки `tidyup`, который помогает делать матрицы более читаемыми, так как арифметика с плавающей точкой на классических компьютерах может приводить к небольшим ошибкам.**

❷ **Вплоть до коэффициента $-i$ (записываемого на Python как `-1j`) поворачивание на 180° вокруг оси *X* приводит к инструкции `x` (NOT), которую мы впервые увидели в главе 2.**

ДЛЯ СПРАВКИ В Python комплексное число *i* представляется как `1.0j`: 1 умножить на *j*, т. е. тем, что в других областях иногда называется мнимым числом *i*.

Этот фрагмент кода иллюстрирует нечто очень важное: операция *x* – это именно то, что мы получаем, поворачивая вокруг оси *X* на угол 180° (π).

ОПРЕДЕЛЕНИЕ Как отмечено в выносках для листинга 6.1, мы можем проверить, что `qt.rx(np.pi)` фактически смещается на коэффициент $-i$ от `qt.sigmax()`. Этот фактор является примером *глобальной фазы*. Как мы вскоре увидим, глобальные фазы *не способны* влиять на результаты измерений. Отсюда `qt.rx(np.pi)` и `qt.sigmax()` являются разными унитарными матрицами, представляющими одну и ту же операцию. В главах 7 и 8 мы рассмотрим глобальную и локальную фазы подробнее.

По аналогии, мы называем поворачивание на 180° вокруг оси *Z* операцией *z*. В главе 3 пакет QuTiP предоставил функцию `qt.sigmax`, которая симулирует инструкцию `x`. Аналогичным образом `qt.sigmaz` предоставляет унитарные матрицы, необходимые для симулирования инструкций `z`. В следующем ниже листинге показан пример использования функции `qt.sigmaz`. Обратите внимание, что мы включили коэффициент (т. е. глобальную фазу) $-i$, сразу же умножив на i; это работает, так как $-i \times i = -(-1) = 1$.

Листинг 6.10 Использование встроенных в QuTiP функций `qt.rz` и `qt.sigmaz`

```
>>> import qutip as qt
>>> import numpy as np
>>> 1j * qt.rz(np.pi).tidyup()        ❶
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]
>>> qt.sigmaz()                       ❷
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]
```

❶ Взаимопогашение глобальной фазы в таком ключе облегчает чтение выходных данных.
❶ Как и было обещано, вплоть до коэффициента $-i$ инструкция `z` применяет поворот на 180° вокруг оси *Z*.

Подобно тому, как операция `X` переворачивает между $|0\rangle$ и $|1\rangle$, оставляя только $|+\rangle$ и $|-\rangle$, операция `z` переворачивает между $|+\rangle$ и $|-\rangle$, оставляя только $|0\rangle$ и $|1\rangle$.

Таблица 6.1 показывает таблицу истинности, подобную тем, которые мы составили для операции Адамара в главе 2. Взглянув на таблицу ис-

тинности, мы можем подтвердить, что если мы дважды выполним операцию z на любом входном состоянии, то мы вернемся к тому, с чего начали. Другими словами, z соответствует операции тождественности $\mathbb{1}$ таким же образом, как $X^2 = \mathbb{1}$.

Таблица 6.1 Представление инструкции z в виде таблицы

Входное состояние	Выходное состояние
$\|0\rangle$	$\|0\rangle$
$\|1\rangle$	$-\|1\rangle$
$\|+\rangle$	$\|-\rangle$
$\|-\rangle$	$\|+\rangle$

ПРИМЕЧАНИЕ В табл. 6.1 перечислено четыре строки, но нам нужны только две из них, чтобы полностью описать действие Z для любого входного значения. Две другие строки служат для того, чтобы подчеркнуть, что мы можем выбирать между определением Z по ее действию на $|0\rangle$ и $|1\rangle$ либо по ее действию на $|+\rangle$ и $|-\rangle$.

Упражнение 6.2: попрактикуйтесь в использовании rz и z

Предположим, что вы готовите кубит в состоянии $|-\rangle$ и применяете поворот z. Если вы измерите вдоль оси X, то что вы получите? Что вы измерите, если примените два поворота z? Если вы имплементируете те же два поворота с помощью rz, то какие углы вы должны использовать?

Мы можем определить еще один поворот таким же образом: поворот вокруг оси, выходящей «за пределы страницы». Эта ось соединяет состояния $(|0\rangle + i|1\rangle)/\sqrt{2} = R_x(\pi/2)|0\rangle$ и $(|0\rangle - i|1\rangle)/\sqrt{2} = R_x(\pi/2)|1\rangle$ и условно называется осью Y. Поворот на 180° вокруг оси Y одновременно переворачивает битовые метки ($|0\rangle \leftrightarrow |1\rangle$) и фазы ($|+\rangle \leftrightarrow |-\rangle$), но оставляет нетронутыми два состояния вдоль оси Y.

Упражнение 6.3: таблица истинности для sigmay

Примените функцию `qt.sigmay()`, чтобы создать таблицу, аналогичную табл. 6.1, но для инструкции y.

ОПРЕДЕЛЕНИЕ Три матрицы X, Y и Z, представляющие операции x, y и z, вместе называются *матрицами Паули* в честь физика Вольфганга Паули. Иногда также включается матрица тожественности $\mathbb{1}$, представляющая операцию «ничего не делать», или операцию тождественности.

Игра в «камень-ножницы-бумага» с матрицами Паули

Матрицы Паули обладают рядом полезных свойств, которые мы будем использовать в остальной части книги. Многие из этих свойств позволяют легко разрабатывать различные уравнения, включающие операторы Паули. Например, если умножить X на Y, то мы получим iZ, но если умножить YX, то получим $-iZ$.

Пакет QuTiP помогает разведать то, что происходит, когда мы умножаем матрицы Паули между собой: например, умножение X и Y между собой в обоих возможных порядках:

```
>>> import qutip as qt
>>> qt.sigmax() * qt.sigmay()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
      False
Qobj data =
[[0.+1.j 0.+0.j]
 [0.+0.j 0.-1.j]]
>>> qt.sigmay() * qt.sigmax()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
      False
Qobj data =
[[0.-1.j 0.+0.j]
 [0.+0.j 0.+1.j]]
```

Аналогичным образом $YZ = iX$ и $ZX = iY$, но $ZY = -iX$ и $XZ = -iY$. Это можно запомнить, если рассматривать X, Y и Z как небольшую игру в «камень-ножницы-бумага»: в порядке следования X «бьет» Y, Y «бьет» Z, и Z «бьет» X.

Мы можем рассуждать об этих матрицах как об установлении своего рода системы координат для кубитовых состояний, именуемой *сферой Блоха*. Как показано на рис. 6.6, оси X и Z образуют окружность, которую мы встречали в книге до этого, тогда как ось Y выходит за пределы страницы.

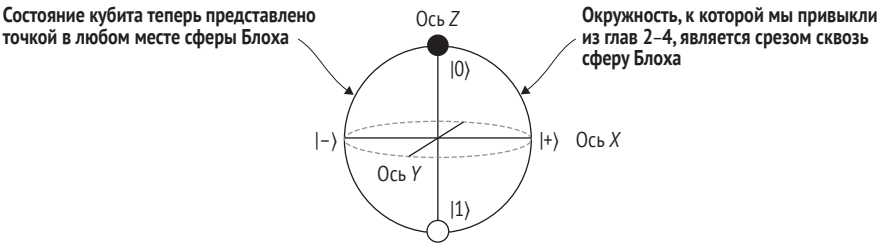


Рис. 6.6 Сфера Блоха во всей ее сферической красе. Здесь каждая из осей помечена соответствующим оператором Паули, представляющим повороты вокруг этой оси

Описание состояний измерениями Паули

Любое однокубитовое состояние может полностью определяться вплоть до глобальной фазы вероятностями измерения для измерений по *X*, *Y* и *Z*. То есть если мы сообщим вам вероятность получить результат «1» для каждого из трех измерений Паули, которые мы могли бы выполнить, то вы можете использовать эту информацию для написания вектора состояния, идентичного нашему, вплоть до глобальной фазы. Это делает аналогию с точками в трех размерностях полезной для размышлений об однокубитовых состояниях.

У *i* это есть

Состояния на концах оси *Y* обычно обозначаются как $|i\rangle$ и $|-i\rangle$, но не часто используются сами по себе. Мы просто будем придерживаться помеченных состояний, которые использовали раньше: $|0\rangle$, $|1\rangle$, $|+\rangle$ и $|-\rangle$.

Запомнив это изображение, легче понять, почему некоторые повороты не влияют на результаты измерений. Например, как показано на рис. 6.7, изображение сферы Блоха помогает нам понять, что произойдет, если мы повернем $|0\rangle$ вокруг оси *Z*.

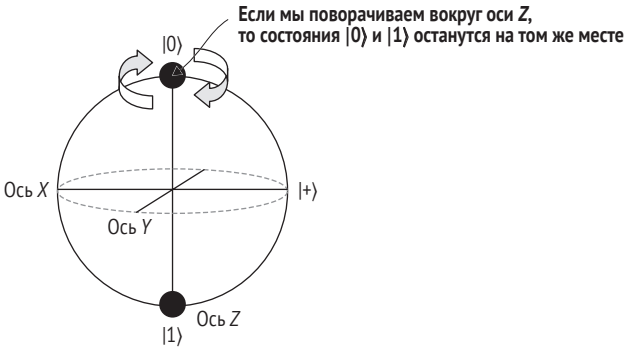


Рис. 6.7 Сфера Блоха, иллюстрирующая, как поворот `rz` оставляет состояние $|0\rangle$ неизменным

Точно так же, как Северный полюс остается в одном и том же месте, независимо от того, насколько сильно мы вращаем Земной шар, если мы вращаем состояние вокруг оси, параллельной этому состоянию, то наблюдаемый эффект на нашем кубите будет отсутствовать. Мы также понимаем, что этот эффект вытекает из математики.

Листинг 6.11 Как повороты `rz` не влияют на состояние $|0\rangle$

```
>>> ket0 = qt.basis(2, 0)                    (1)
>>> ket_psi = qt.rz(np.pi / 3) * ket0        (2)
```

```
>>> ket_psi                                                    ❸
Quantum object: dims = [[2], [1]], shape = (2, 1),
➥ type = ket
Qobj data =
[[ 0.8660254-0.5j]
 [ 0.0000000+0.j ]]
>>> bra0 = ket0.dag()                                          ❹
>>> bra0
Quantum object: dims = [[1], [2]], shape = (1, 2),
➥ type = bra
Qobj data =                                                    ❺
[[ 1. 0.]]
>>> np.abs((bra0 * ket_psi)[0, 0]) ** 2                        ❻
1.0
```

❶ Определяет переменную, которая представляет состояние $|0\rangle$.
❷ Вводит новое состояние $|\psi\rangle$, т. е. поворот состояния $|0\rangle$ на 60° ($\pi/3$ в радианах) относительно оси Z.
❸ Результирующее состояние $|\psi\rangle = [\cos(60°/2) - i\sin(60°/2)]\,|0\rangle = [\sqrt{3}/2 - i/2]\,|0\rangle$.
❹ Вспомните, что в QuTiP мы записываем оператор «кинжал» (†) путем вызова метода .dag экземпляров класса Qobj, в данном случае получая вектор-строку для $\langle 0|$.
❺ Взяв внутреннее произведение $\langle 0|\psi\rangle$, мы можем вычислить правило Борна $\Pr(0|\psi) = |\langle 0|\psi\rangle|^2$. Обратите внимание, что нам нужно индексировать по [0, 0], так как пакет QuTiP представляет внутреннее произведение $|0\rangle$ с $|\psi\rangle$ в виде матрицы 1×1.
❻ Как и прежде, вероятность наблюдать «0» при измерении вдоль оси *Z* не изменилась.

ДЛЯ СПРАВКИ При записи состояний $|\psi\rangle$ часто используется в качестве произвольного имени, аналогично тому, как *x* в алгебре часто используется для представления произвольной переменной.

В общем случае мы всегда можем умножить состояние на комплексное число, абсолютное значение которого равно 1, не изменяя вероятности любого измерения. Любое комплексное число $z = a + bi$ может быть записано как $z = re^{i\theta}$ для действительных чисел r и θ, где r – это абсолютное значение z и где θ – это угол. При $r = 1$ мы имеем число вида $e^{i\theta}$, которое называем *фазой*. Тогда мы говорим, что умножение состояния на фазу применяет *глобальную фазу* к этому состоянию.

Упражнения 6.4: проверка неизменности $|0\rangle$ при применении `rz`

Мы проверили только то, что вероятность одного измерения остается прежней, но, возможно, вероятности изменились для измерений по *X* или *Y*. В целях полной проверки того, что глобальная фаза ничего не изменяет, подготовьте то же состояние и поворот, что и в листинге 6.11, и проверьте, что при применении инструкции `rz` вероятности измерить состояние вдоль оси *X* либо *Y* не изменяются.

ПРИМЕЧАНИЕ Никакая глобальная фаза никогда не может быть обнаружена каким-либо измерением.

Состояния $|\psi\rangle$ и $e^{i\theta}|\psi\rangle$ во всех мыслимых отношениях являются двумя разными способами описания одного и того же состояния. Не существует никакого квантового измерения, которое мы могли бы сделать *даже в принципе*, чтобы узнать о глобальных фазах. С другой стороны, мы видели, что можем различать такие состояния, как $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ и $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$, которые отличаются только *локальной фазой* вычислительно-базисного состояния $|1\rangle$.

Сделав шаг назад, давайте обобщим то, что мы узнали об инструкциях x, y и z до этого, а также о матрицах Паули, которые мы используем для симулирования указанных инструкций. Мы узнали, что инструкция x переключает между $|0\rangle$ и $|1\rangle$, тогда как инструкция z переключает между состояниями $|+\rangle$ и $|-\rangle$. Говоря по-другому, инструкция x переворачивает биты, тогда как инструкция z переворачивает фазы.

Глядя на сферу Блоха, мы видим, что поворачивание вокруг оси Y должно выполнять и то, и другое. Мы также можем увидеть это посредством $Y = -iXZ$, так как это легко проверить с помощью пакета QuTiP. Мы подытоживаем все, что делает каждая инструкция Паули, в табл. 6.2.

Таблица 6.2 Матрицы Паули в виде битовых и фазовых переворотов

Инструкция	Матрица Паули	Переворачивает биты ($\|0?\rangle \leftrightarrow \|1?\rangle$)	Переворачивает фазы ($\|+?\rangle \leftrightarrow \|-?\rangle$)?
(инструкция отсутствует)	$\mathbb{1}$	Нет	Нет
x	X	Да	Нет
y	Y	Да	Да
z	Z	Нет	Да

Упражнение 6.5: хм, это что-то напоминает

Состояние $(|00\rangle + |11\rangle)/\sqrt{2}$, которое мы встречали несколько раз, не единственный пример запутанного состояния. На самом деле если мы выберем двухкубитовое состояние случайным образом, то оно почти наверняка будет запутано. Подобно тому, как вычислительный базис $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ является особенно полезным множеством незапутанных состояний, существует множество из четырех конкретных запутанных состояний под названием базис Белла, названное в честь физика Джона Стюарта Белла:

Имя	Расширение в вычислительном базисе
$\|\beta_{00}\rangle$	$(\|00\rangle + \|11\rangle)/\sqrt{2}$
$\|\beta_{01}\rangle$	$(\|00\rangle - \|11\rangle)/\sqrt{2}$
$\|\beta_{10}\rangle$	$(\|01\rangle + \|10\rangle)/\sqrt{2}$
$\|\beta_{11}\rangle$	$(\|01\rangle - \|10\rangle)/\sqrt{2}$

Используя то, что вы узнали об инструкции `cnot` и инструкциях Паули (x, y и z), напишите программы для подготовки каждого из четырех *состояний Белла* из таблицы.

Подсказка: в этом упражнении табл. 6.2 окажется весьма кстати.

Мы закончим наше обсуждение однокубитовых операций, добавив в наш интерфейс `Qubit` и симулятор инструкции для операций `x`, `y` и `z`. Для этого мы имплементируем поворотные инструкции `rx`, `ry` и `rz`, используя соответствующие функции пакета QTIP `qt.rx`, `qt.ry` и `qt.rz`, чтобы получить копии унитарных матриц, которые нам потребуются для симулирования каждой инструкции. Вот как это можно сделать в нашем симуляторе.

Листинг 6.12 simulator.py: добавление всех поворотов Паули

```
def rx(self, theta: float) -> None:
    self.parent._apply(qt.rx(theta), [self.qubit_id])

def ry(self, theta: float) -> None:
    self.parent._apply(qt.ry(theta), [self.qubit_id])

def rz(self, theta: float) -> None:
    self.parent._apply(qt.rz(theta), [self.qubit_id])

def x(self) -> None:
    self.parent._apply(qt.sigmax(), [self.qubit_id])

def y(self) -> None:
    self.parent._apply(qt.sigmay(), [self.qubit_id])

def z(self) -> None:
    self.parent._apply(qt.sigmaz(), [self.qubit_id])
```

В пакете QuTiP для матриц Паули используется обозначение σ_x вместо X. Используя эту нотацию, функция `sigmax()` возвращает новый `Qobj`, представляющий матрицу Паули X. Благодаря этому мы можем имплементировать инструкции `x`, `y` и `z`, соответствующие каждой матрице Паули.

Никто не может сказать, что такое матрица; ты должен увидеть ее сам

В части I этой книги мы много говорили о матрицах. Очень много. Заманчиво сказать, что квантовое программирование всецело касается матриц и что кубиты на самом деле – это просто векторы. Однако в действительности матрицы – это способ *симулирования* того, что делает квантовое устройство. Мы узнаем об этом больше в части II, но квантовые программы вообще не манипулируют матрицами и векторами – они манипулируют классическими данными, такими как выбор инструкций для отправки на квантовое устройство и что делать с данными, которые мы получаем из устройств. Например, если мы выполняем инструкцию на устройстве, то не существует простого способа увидеть, какую матрицу мы должны использовать для симулирования этой инструкции, – правильнее сказать, мы должны реконструировать эту матрицу из многих повторных измерений, используя метод, именуемый процессной томографией.

Когда мы пишем матрицу, будь то в исходном коде или на листе бумаги, мы неявно симулируем квантовую систему. Если это действительно испортит вашу лапшу, то не волнуйтесь; это будет иметь для вас гораздо больше смысла, когда вы прочтете остальную часть книги.

6.3 Телепортация

Итак, теперь у нас есть все необходимое, для того чтобы описать то, как выглядит телепортация как квантовая программа. В качестве краткой ревизии на рис. 6.8 показано то, что мы хотим сделать с этой программой.

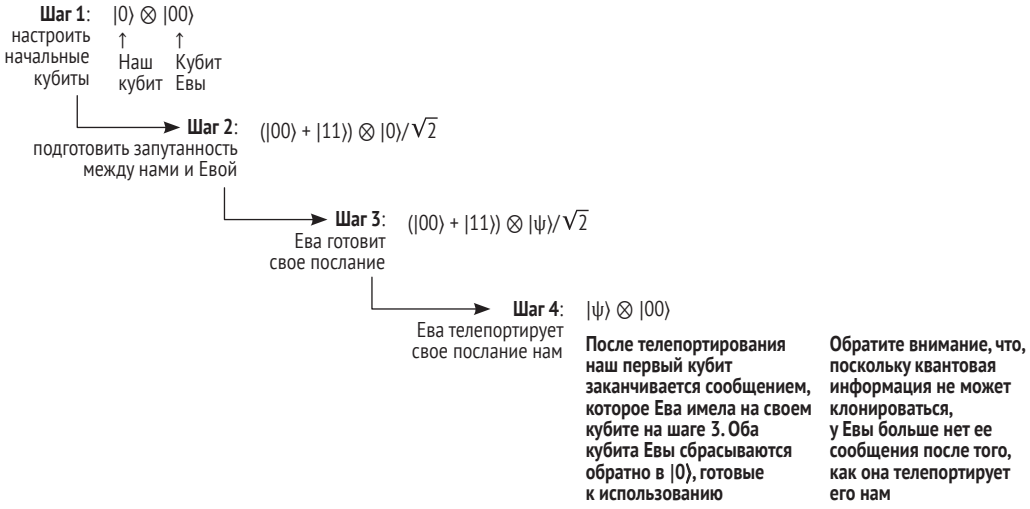


Рис. 6.8 Вспомните шаги программы телепортации

Будем считать, что вы можете подготовить несколько запутанных кубитов, пока они находятся на одном устройстве, и что у нас с Евой есть средство классического обмена информацией, которое мы можем использовать, чтобы сигнализировать о правильной коррекции для использования. Теперь мы можем воспользоваться функциональностями, которые добавили в наш симулятор в этой главе, с целью имплементирования программы телепортации.

Листинг 6.13 teleport.py: квантовая телепортация всего в нескольких строках кода

```
from interface import QuantumDevice, Qubit
from simulator import Simulator

def teleport(msg: Qubit, here: Qubit, there: Qubit) -> None:          (1)
```

```
here.h()                                                    ❷
here.cnot(there)                                            ❸

# ...                                                       ❹
msg.cnot(here)                                              ❺
msg.h()

if msg.measure(): there.z()                                 ❻
if here.measure(): there.x()

msg.reset()                                                 ❼
here.reset()
```

❶ Функция teleport принимает на входе два кубита: кубит, который мы хотим переместить (msg), и место, куда мы хотим его переместить (there). Нам также нужен один временный кубит, который мы называем here. Мы предполагаем по соглашению, что и here, и there начинаются в состоянии $|0\rangle$.

❷ Нам нужно начать с некоторой запутанности между here (здесь) и there (там). Мы можем применить нашего старого друга, инструкцию h, вместе с нашим новым другом, инструкцией `cnot`.

❸ Единственная инструкция в этой программе, которая должна действовать как здесь, так и там. После ее выполнения мы можем отправить Еве наш кубит, и мы оба сможем выполнить остальную часть программы только с классическим обменом информацией.

❹ В этой точке программы here и there находятся в состоянии $(|00\rangle + |11\rangle)/\sqrt{2}$, которое мы впервые увидели в главе 4.

❺ Выполняет программу, которую мы использовали для обратной подготовки состояния $(|00\rangle + |11\rangle)/\sqrt{2}$, но на кубитах msg и here, которые обитают полностью на нашем устройстве. Мы можем рассуждать о выполнении подготовки в обратную сторону как о своего рода измерении, вследствие чего эти шаги позволяют нам измерить квантовое сообщение, которое мы пытаемся отправить Еве в запутанном состоянии.

❻ Когда мы выполняем это измерение фактически, то получаем классические данные для отправки Еве. Получив эти данные, она сможет применить инструкции x и z для декодирования квантового сообщения.

❼ Теперь, когда мы закончили с нашими кубитами, неплохо поставить их в состояние $|0\rangle$, чтобы они были готовы к использованию снова. Однако это не влияет на состояние there, так как мы сбросили только наши кубиты, а не тот, который мы дали Еве!

Видишь *с*? Вот ей мы там и занимались

Предположим, нам не нужно было отправлять Еве классический результат измерения в рамках телепортации. В этом случае мы могли бы использовать телепортацию для передачи как классических, так и квантовых данных быстрее скорости света. Подобно тому, как мы не могли общаться с Евой, когда играли в игру CHSH в главе 5, скорость света означает, что нам нужно общаться с Евой классически, чтобы использовать запутанность для отправки квантовых данных. В обоих случаях запутанность помогает нам общаться, но она не позволяет нам общаться самостоятельно: нам всегда также нужен какой-то другой вид общения.

Для того чтобы убедиться, что это действительно работает, мы можем подготовить что-то на нашем кубите и отправить это Еве, а затем она может откатить нашу подготовку на своем кубите. До сих пор сообщения,

которые мы и Ева посылали, были классическими, но здесь сообщение является *квантовым*. Мы можем и будем измерять квантовое сообщение, чтобы получать классический бит, но мы также можем использовать квантовое сообщение, которое мы получаем от Евы, как и любые другие квантовые данные. Например, мы можем применять любые повороты и другие инструкции, которые нам нравятся.

Для чего это нужно?

Отправка квантовых данных может показаться не намного более полезной, чем отправка классических данных; в конце концов, отправка классических данных до сих пор приносила нам много хороших вещей. В отличие от нее, приложения для отправки квантовых данных в настоящее время, как правило, являются чуть более нишевыми.

Тем не менее перемещение квантовых данных представляет собой действительно полезный пример, помогающий нам разобраться в том, как работают квантовые компьютеры. Идеи, разработанные в этой главе, не часто полезны непосредственно, но помогут нам строить отличные вещи в будущем.

Допустим, мы готовим *квантовое* сообщение с помощью операции `msg.ry(0.123)`. В следующем ниже листинге показано то, каким образом можно телепортировать это сообщение Еве.

Листинг 6.14 teleport.py: использование телепортации для перемещения квантовых данных

```
if __name__ == "__main__":
    sim = Simulator(capacity=3)
    with sim.using_register(3) as (msg, here, there):        (1)
        msg.ry(0.123)                                        (2)
        teleport(msg, here, there)                           (3)

        there.ry(-0.123)                                     (4)
        sim.dump()
```

(1) Как и прежде, выделяет реестр кубитов и дает каждому кубиту имя.
(2) Готовит сообщение для отправки Еве. Здесь мы показали использование конкретного угла в качестве сообщения, но это может быть что угодно.
(3) Вызывает телепортационную программу, которую мы написали ранее, чтобы переместить подготовленное нами сообщение на кубит Евы.
(4) Проверяет результат инструкции dump, чтобы убедиться, что выделенный нами реестр вернулся в состояние |000⟩.

Если Ева затем откатит наш поворот путем поворачивания на противоположный угол, мы сможем проверить, что выделенный нами реестр вернулся в состояние |000⟩. Это показывает, что наша телепортация сработала! При выполнении данной программы вы получите результат, аналогичный следующему ниже:

```
Quantum object: dims = [[2, 2, 2], [1, 1, 1]],
➥ shape = (8, 1), type = ket
Qobj data =
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

ПРИМЕЧАНИЕ Ваши данные на выходе могут отличаться на глобальную фазу, в зависимости от того, какие результаты измерения вы получили.

В целях верификации того, что телепортация сработала, если Ева откатит инструкцию, которую мы выполнили на ее кубите (`there.ry(0.123)`), то она должна вернуться в состояние $|0\rangle$, с которого мы начали. С помощью телепортации мы и Ева можем передавать квантовую информацию, используя запутанность и классический обмен информацией, как вы можете убедиться в упражнении 6.6.

Упражнение 6.6: что делать, если она ничего не сделала?

Попробуйте изменить свою операцию или операцию Евы, чтобы убедиться, что в конце вы получите состояние $|000\rangle$, только если откатите ту же операцию, которую Ева применила к своему кубиту.

Теперь мы можем похвастаться перед всеми нашими друзьями (ссылаясь на любые научно-фантастические штучки, которые мы захотим), что мы умеем телепортировать. Надеюсь, вы понимаете, почему это *не то же самое*, что телепортировать на планету с орбиты, и что при телепортировании вашего сообщения оно не передается быстрее скорости света.

Резюме

- Теорема о запрете клонирования не позволяет нам копировать произвольные данные, хранящиеся в квантовых реестрах, но мы все же можем перемещать данные с помощью операции `swap` и таких алгоритмов, как телепортация.
- Когда кубит не запутан с другими кубитами, мы можем визуализировать его состояние как точку на сфере, именуемой *сферой Блоха*.
- В общем случае мы можем поворачивать состояние одного кубита вокруг оси *X*, *Y* либо *Z*. Операции, которые применяют повороты на 180°

вокруг каждой из этих осей, называются *операциями Паули* и описывают переворачивание либо битов, либо фаз, либо обоих.
- В квантовой телепортации запутывание используется вместе с поворотами Паули для передачи квантовых данных из одного кубита в другой кубит без его копирования.

Часть I: заключение

Мы дошли до конца первой части книги, но, к сожалению, наши кубиты находятся в другом за́мке 😆. Пройти через эту часть было непросто, так как мы строили симулятор для квантового устройства, усваивая целый ряд новых квантовых концепций. Вы, вероятно, немного не уверены в некоторых вопросах или темах, и это нормально.

> **ДЛЯ СПРАВКИ** Приложение B к книге содержит небольшой справочный материал (глоссарий и определения), который будет полезен по мере продвижения вперед по этой книге и в ваших последующих усилиях по квантовой разработке.

Мы будем использовать и практиковать эти навыки для разработки более сложных квантовых программ для таких интересных приложений, как химия и криптография. Однако, прежде чем перейти к этим темам, похлопайте себя по спине: на данный момент вы многого добились! Давайте же подытожим некоторые ваши достижения:

- освежили свои навыки в линейной алгебре и комплексных числах;
- узнали, что такое кубит и что можно с ним делать;
- построили многокубитовый симулятор на языке Python;
- написали квантовые программы для таких задач, как квантовое распределение ключей (QKD), игры в нелокальные игры и даже квантовая телепортация;
- усвоили бракетную нотацию для состояний квантовых систем.

Ваш Python'овский симулятор по-прежнему будет полезным инструментом для понимания того, что происходит, когда мы перейдем к более крупным приложениям. В части II мы перейдем к использованию языка Q# в качестве инструмента для написания квантовых программ. В следующей далее главе мы поговорим о том, почему мы будем писать более продвинутые квантовые программы на Q#, а не на Python, но главными причинами являются скорость и расширяемость. Кроме того, вы можете использовать Q# из Python либо с ядром Q# для Jupyter, так что вы можете работать с любой средой разработки, которая вам нравится больше.

Увидимся в части II!

Часть II

Программирование квантовых алгоритмов на Q#

При выполнении своих задач в качестве разработчика наличие правильного инструмента помогает выполнять работу правильно – и квантовые вычисления не являются исключением. Квантово-вычислительные программные стеки, как правило, довольно разнообразны. В части II мы расширим наш стек, чтобы включить Q#, *предметно-специфичный* язык программирования для работы с квантовыми устройствами. Подобно тому, как другие предметно-специфичные языки помогают в работе со специализированным классическим оборудованием, таким как графические процессоры и FPGA, Q# помогает нам получать максимальную отдачу от квантовых вычислений, предоставляя правильные инструменты для имплементирования и применения квантовых алгоритмов. Как и в случае с другими специализированными языками, Q# отлично работает при взаимодействии с более универсальными языками и платформами, такими как Python и .NET.

Имея все это в виду, мы начинаем оттачивать свои навыки и изучать основы написания квантовых алгоритмов. В главе 7 мы вернемся к квантовым генераторам случайных чисел, чтобы узнать основы языка Q#, опираясь на навыки, которые мы развили в главе 2. Далее, в главе 8, мы расширим наш инструментарий методов квантового программирования, узнав о *фазовой отдаче* (кикбэке), *оракулах* и других новых хитростях, и воспользуемся ими, чтобы поиграть в несколько забавных новых игр. Наконец, в главе 9 мы узнаем о методе фазового оценивания, развив навыки, которые нам понадобятся, когда мы начнем работать над более прикладными задачами в части III.

7 Перевес в другую пользу: введение в язык программирования Q#

Эта глава охватывает следующие ниже темы:

- использование Комплекта инструментов для квантовой разработки с целью написания квантовых программ на Q#;
- применение среды блокнотов Jupyter Notebook для работы с Q#;
- выполнение программ на Q# с использованием классического симулятора.

До этого момента для имплементирования нашего собственного программного стека с целью симулирования квантовых программ мы использовали Python. Если вы помните, рис. 2.1 (снова приводимый как рис. 7.1) был хорошей моделью того, как программы, которые мы пишем, взаимодействуют с квантовым симулятором и устройствами, которые мы используем и строим в качестве квантовых разработчиков.

Продвигаясь вперед, мы будем писать более сложные квантовые программы, которые будут выигрывать от специализированных языковых функциональностей, которые трудно имплементировать, встраивая наш программный стек в Python. В особенности когда мы занимаемся разведкой квантовых алгоритмов, полезно иметь в своем распоряжении язык, специально разработанный для квантового программирования. В этой главе мы начнем с Q#, предметно-ориентированного языка квантового

программирования от компании Microsoft, входящего в Комплект инструментов для квантовой разработки.

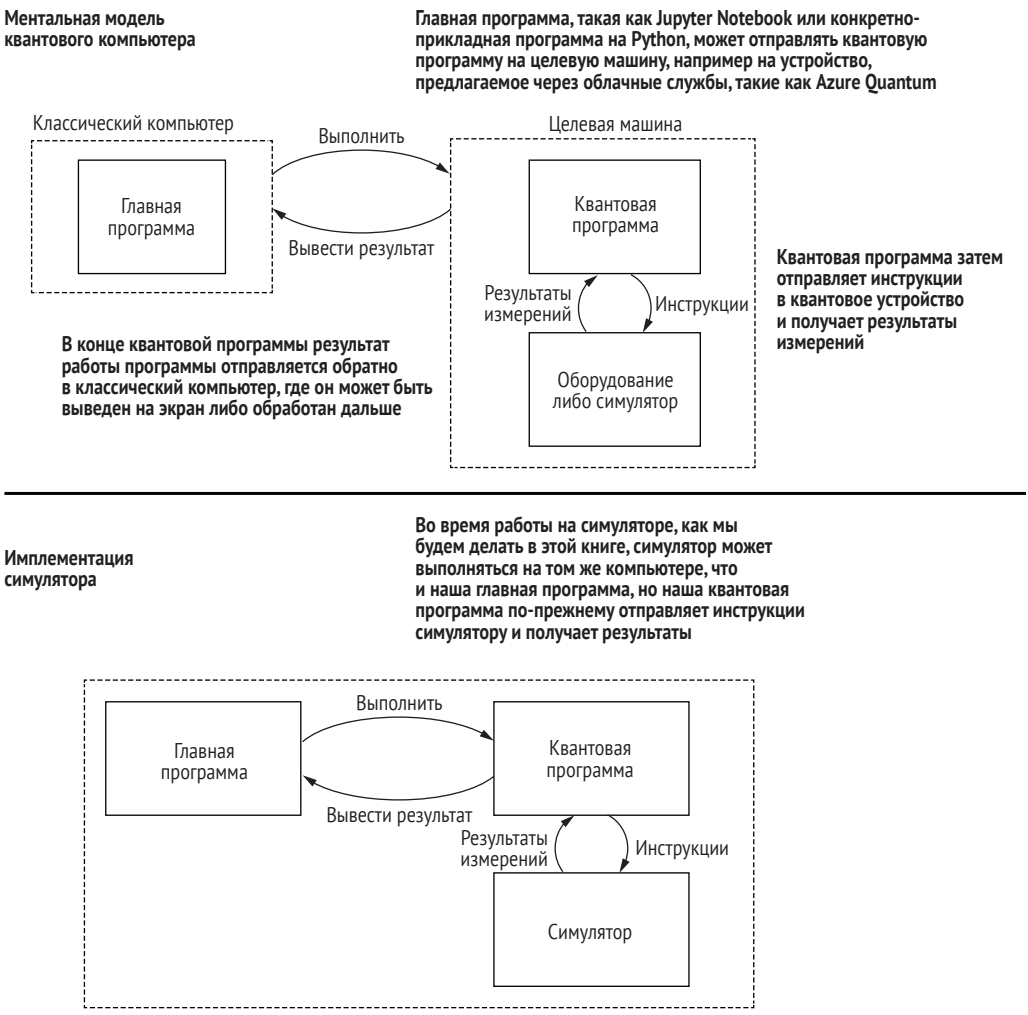


Рис. 7.1 Ментальная модель возможного использования квантового компьютера. Верхняя половина рисунка является общей моделью квантового компьютера. В этой книге мы будем использовать локальные симуляторы. Нижняя половина показывает то, что мы будем строить и использовать

7.1 Введение в Комплект инструментов для квантовой разработки

Комплект инструментов для квантовой разработки (Quantum Development Kit) предоставляет новый язык, Q#, для написания квантовых про-

грамм и их симулирования с использованием классических ресурсов. Квантовые программы, написанные на Q#, выполняются, рассматривая квантовые устройства как своего рода ускорители, аналогично тому, как мы могли бы выполнять исходный код на видеокарте.

ДЛЯ СПРАВКИ Если вы когда-либо использовали каркас программирования видеокарт, такой как CUDA или OpenCL, то это очень похожая модель.

Давайте взглянем на программный стек для языка Q# на рис. 7.2.

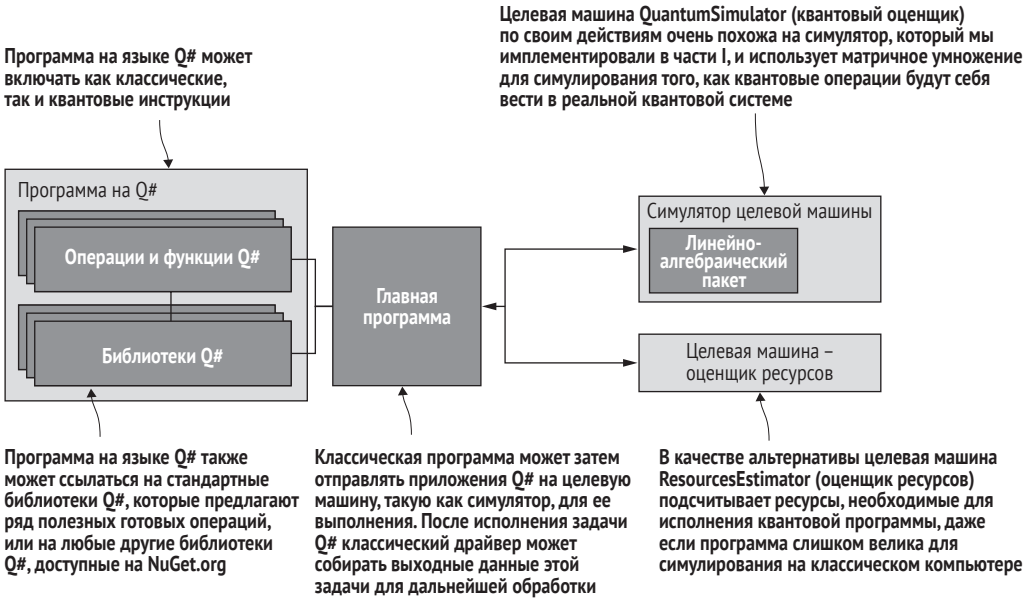


Рис. 7.2 Программный стек Комплекта инструментов для квантовой разработки от компании Microsoft на классическом компьютере. Мы можем написать программу на Q#, состоящую из функций и операций, ссылаясь на любые библиотеки Q#, которые мы хотим включить. Затем главная программа может координировать обмен информацией между нашей программой на Q# и целевой машиной (например, симулятором, работающим локально на нашем компьютере)

Программа на языке Q# состоит из операций и функций, которые инструктируют квантовое и классическое оборудование выполнять те или иные действия. Вместе с языком Q# также поставляется много библиотек, которые имеют полезные, готовые операции и функции для использования в наших программах.

После написания программы на Q# нам понадобится способ передачи инструкций оборудованию. Классическая программа, иногда называемая *драйвером*, или *главной программой*, отвечает за выделение целевой машины и выполнение операции Q# на этой машине.

Комплект инструментов для квантовой разработки включает в себя плагин для среды блокнотов Jupyter Notebook под названием IQ#, кото-

рый позволяет легко начать работу с Q#, автоматически предоставляя главные (хост-) программы. В главе 9 мы рассмотрим, как писать главные программы с использованием Python, но сейчас сосредоточимся на языке Q#. Инструкции по настройке среды Q# для работы в среде Jupyter Notebook см. в приложении А к книге.

Применяя плагин IQ# для среды Jupyter Notebook, мы можем использовать одну из двух разных целевых машин для выполнения исходного кода Q#. Первая – это целевая машина QuantumSimulator, которая очень похожа на симулятор, который мы разрабатываем на языке Python. Она будет намного быстрее, чем наш код на Python для симулирования кубитов.

Вторая – это целевая машина ResourcesEstimator, которая позволит нам оценивать количество кубитов и квантовых инструкций, которые нам потребуются для ее выполнения без полного ее симулирования. Это особенно полезно для получения представления о ресурсах, которые нам понадобятся для выполнения нашего приложения, написанного на Q#, в чем мы убедимся, когда будем рассматривать в книге более крупные программы Q# позже.

В целях получения представления о том, как все работает, давайте начнем с написания чисто классического приложения Q# «Привет, мир!». Во-первых, запустите Jupyter Notebook, выполнив следующие ниже действия в терминале[1]:

```
jupyter notebook
```

Эта команда автоматически откроет новую вкладку в браузере с домашней страницей для сеанса Jupyter Notebook. В меню **Создать** ↓ выберите Q#, чтобы создать новый блокнот Q#. Введите следующее ниже в первую пустую ячейку блокнота и нажмите **Ctrl**+**Enter** или ⌘+**Enter**, чтобы ее выполнить:

```
function HelloWorld() : Unit {                ❶
    Message("Hello, classical world!");       ❷
}
```

❶ **Определяет новую функцию, которая не принимает аргументов, и возвращает пустой кортеж, тип которого записан как Unit.**

❷ **Указывает целевой машине собрать диагностическое сообщение. Целевая машина QuantumSimulator выводит всю диагностику на экран, поэтому мы можем использовать Message как инструкцию print на Python.**

Вы должны получить нечто похожее на рис. 7.3.

ДЛЯ СПРАВКИ В отличие от Python, в языке Q# для завершения инструкций используются не символы новой строки, а точки с за-

[1] Альтернативной командой является: `python -m jupyterlab`, при условии что у вас установлен пакет jupyterlab (pip install jupyterlab). См. https://jupyterlab.readthedocs.io/en/stable/getting_started/installation.html. – *Прим. перев.*

пятой. Если вы получаете много ошибок компилятора, убедитесь, что вы не забыли про точки с запятой.

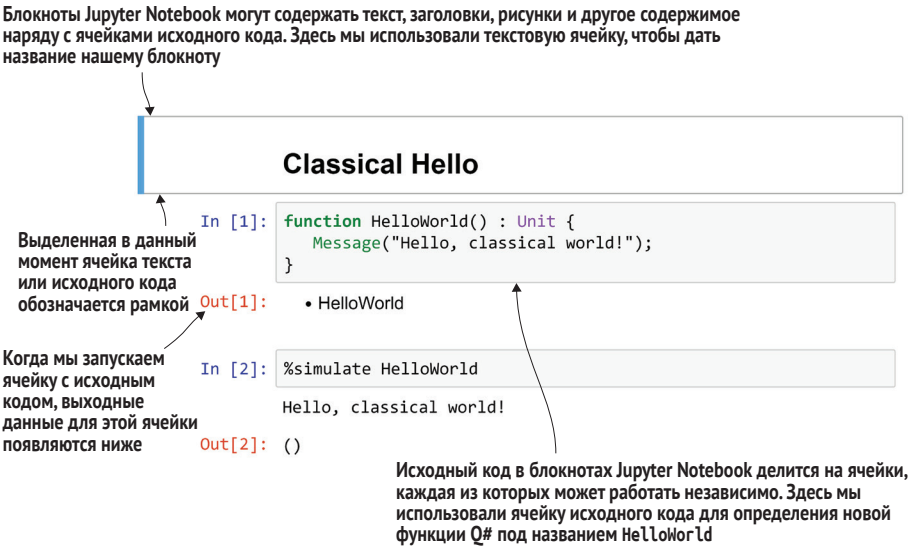


Рис. 7.3 Начало работы с IQ# и со средой блокнотов Jupyter Notebook. Здесь новая функция Q# под названием HelloWorld, определяется как первая ячейка в блокноте Jupyter Notebook, а результаты симулирования этой функции находятся во второй ячейке

Вы должны получить ответ, что функция `HelloWorld` была успешно скомпилирована. Для выполнения новой функции можно применить команду `%simulate` в новой ячейке.

Листинг 7.1 Использование волшебной команды `%simulate` в Jupyter

```
In [2]: %simulate HelloWorld
Hello, classical world!
```

Немного классического волшебства

Команда `%simulate` является примером *волшебной команды*, поскольку на самом деле она не является частью Q#, а является инструкцией для среды Jupyter Notebook. Если вы знакомы с плагином IPython для Jupyter, то вы, возможно, использовали аналогичные волшебные команды, чтобы сообщать Jupyter, как обрабатывать функции Python построения графиков.

В этой книге используемые нами волшебные команды начинаются с %, чтобы их было легко отличать от исходного кода Q#.

В данном примере `%simulate` выделяет для нас целевую машину и отправляет функцию или операцию Q# на эту новую целевую машину. В главе 9 мы увидим, как делать что-то подобное с помощью главных (хост-) программ Python вместо использования среды блокнотов Jupyter Notebook.

Программа Q# отправляется в симулятор, но в данном случае симулятор просто выполняет классическую логику, поскольку пока что нет квантовых инструкций, о которых нужно беспокоиться.

Упражнение 7.1: изменение приветствия

Измените определение функции `HelloWorld`, чтобы вместо «classical world» включить свое имя, а затем снова выполните команду `%simulate`, используя новое определение.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

7.2 Функции и операции в Q#

Теперь, когда у нас есть Комплект инструментов для квантовой разработки, который готов к работе в среде блокнотов Jupyter Notebook, давайте воспользуемся языком Q# для написания нескольких квантовых программ. Еще в главе 2 мы увидели, что с кубитом можно генерировать случайные числа по одному классическому биту за раз. Пересмотр этого приложения будет отличным местом для начала работы на языке Q#, тем более что случайные числа полезны, если мы хотим играть в игры.

7.2.1 Игры с квантовыми генераторами случайных чисел на Q#

Давным-давно, в Камелоте, Моргана страстно любила, как и мы, играть в игры. Будучи умным математиком с навыками, выходящими далеко за рамки ее собственного времени, Моргана время от времени использовала в своих играх даже кубиты. Однажды, когда сэр Ланселот спал под деревом, Моргана поймала его в ловушку, вызвав на небольшую игру: каждый из них должен попытаться угадать результат измерения одного из кубитов Морганы.

Если результат измерения по оси Z равен 0, то Ланселот выигрывает свою игру и возвращается к Гвиневре. Однако если результат равен 1, то Моргана выигрывает, и Ланселот должен остаться и сыграть еще один раунд. Обратите внимание на сходство с нашей предыдущей программой QRNG. Как и в главе 2, мы будем измерять кубит для генерирования

случайных чисел, на этот раз для игры. Разумеется, Моргана и Ланселот могли бы подбрасывать более традиционную монету, но что в этом интересного?

Две стороны одной... одного кубита?

В главе 2 мы разобрались в том, как генерировать случайные числа по одному биту за раз, подготавливая и измеряя кубиты. То есть кубиты можно использовать для имплементирования *монет*. Мы будем использовать ту же идею в этой главе и рассуждать о монете как о своего рода интерфейсе, который позволяет пользователю ее «подбрасывать» и получать случайный бит. То есть мы можем имплементировать интерфейс монеты, подготавливая и измеряя кубиты.

Вот шаги игры со стороны Морганы.

1. Подготовить кубит в состоянии $|0\rangle$.
2. Применить операцию Адамара (напомним, что унитарный оператор H переводит $|0\rangle$ в $|+\rangle$ и наоборот).
3. Измерить кубит по оси Z. Если результат измерения равен 0, то Ланселот может идти домой. В противном случае ему придется остаться и играть снова!

Сидя в кафе и наблюдая за тем, как мимо проходит мир, мы можем использовать наши ноутбуки, чтобы предсказывать, что произойдет в игре Морганы с Ланселотом, написав квантовую программу на Q#. В отличие от функции `HelloWorld`, о которой мы писали ранее, наша новая программа должна будет работать с кубитами, поэтому давайте на минутку посмотрим, как это сделать с помощью Комплекта инструментов для квантовой разработки.

Первостепенный способ взаимодействия с кубитами на Q# состоит в вызове *операций*, представляющих квантовые инструкции. Например, операция H в Q# представляет инструкцию Адамара, которую мы видели в главе 2. В целях понимания того, как эти операции работают, полезно знать разницу между операциями и функциями Q#, которые мы видели в примере `HelloWorld`:

- *функции* в Q# представляют *предсказуемую* классическую логику: такие вещи, как математические функции (`Sin`, `Log`). Функции всегда возвращают один и тот же результат, когда заданы одни и те же входные данные;
- *операции* в Q# представляют исходный код, который может иметь *побочные эффекты*, такие как отбор случайных чисел или выдача квантовых инструкций, изменяющих состояние одного или нескольких кубитов.

Это разделение помогает компилятору понимать то, как автоматически преобразовывать наш исходный код в рамках более крупных квантовых программ; мы подробнее поговорим об этом позже.

Еще одна точка зрения на разницу между функциями и операциями

На различие между функциями и операциями можно взглянуть и по-другому, как на то, что функции вычисляют вещи, но не могут ничего *вызывать*. Независимо от того, сколько раз мы вызываем функцию квадратного корня `Sqrt`, в нашей программе Q# ничего не поменяется. Напротив, если мы выполняем операцию `X`, то на наше квантовое устройство отправляется инструкция `X`, которая вызывает изменение состояния устройства. В зависимости от начального состояния кубита, к которому была применена инструкция `X`, мы можем сказать, что инструкция `X` была применена, измерив кубит. Поскольку функции ничего не делают в этом смысле, мы всегда можем точно предсказать их результат при наличии тех же входных данных.

Одним из важных следствий является то, что функции не могут вызывать операции, но операции могут вызывать функции. Это обусловлено тем, что у нас может быть операция, которая не обязательно является предсказуемой, и эта операция вызывает предсказуемую функцию, и при этом у нас по-прежнему есть что-то, что может быть или не быть предсказуемым. Однако предсказуемая функция не может вызывать потенциально непредсказуемую операцию и при этом оставаться предсказуемой.

Мы будем рассматривать разницу между функциями и операциями Q# подробнее по мере их использования на протяжении всей книги.

Поскольку мы хотим, чтобы квантовые инструкции влияли на наши квантовые устройства (и судьбу Ланселота), все квантовые операции в Q# определяются как операции (отсюда и название). Например, предположим, что Моргана и Ланселот готовят свой кубит в состоянии $|+\rangle$, используя инструкцию Адамара. Тогда мы можем предсказать исход их игры, записав пример квантового генератора случайных чисел (QRNG) из главы 2 как операцию Q#.

У этой операции побочные эффекты...

Когда мы хотим отправить инструкции на целевую машину, чтобы что-то сделать с нашими кубитами, нам нужно сделать это из операции, поскольку отправка инструкции – это своего рода *побочный эффект*. То есть когда мы выполняем операцию, мы не просто что-то вычисляем: мы что-то *делаем*. Выполнение операции дважды не то же самое, что выполнение ее один раз, даже если мы оба раза получаем один и тот же результат. Побочные эффекты не являются детерминированными или предопределенными, поэтому мы не можем использовать функции для отправки инструкций по манипулированию нашими кубитами.

В листинге 7.2 мы делаем именно это, начиная с написания операции под названием `GetNextRandomBit` для симулирования каждого раунда игры Морганы. Обратите внимание, что поскольку операция `GetNext-`

`RandomBit` должна работать с кубитами, она должна быть операцией, а не функцией. Мы можем запросить у целевой машины один или несколько свежих кубитов с помощью инструкции use.

Выделение кубитов на языке Q#

Инструкция use является одним из двух способов запрашивать у целевой машины кубиты. Нет никаких ограничений на число инструкций use, которые мы можем иметь в программах Q#, кроме числа кубитов, которые каждая целевая машина способна выделять.

В конце блока (т. е. тела `operation`, `for` или `if`), содержащего инструкцию `use`, кубиты возвращаются на целевую машину. Нередко кубиты выделяются в таком виде в начале тела операции, поэтому один из вариантов рассматривать операторы `use` – обеспечение того, чтобы каждый выделенный кубит «принадлежал» той или иной операции. Это делает невозможной «утечку» кубитов в программе Q#, что очень полезно, учитывая, что на реальном квантовом оборудовании кубиты, вероятно, будут очень дорогими ресурсами.

Если нам нужно больше контроля над тем, когда кубиты вывобождаются, то инструкции `use` также могут необязательно сопровождаться блоком, обозначаемым круглыми скобками { и }. В этом случае кубиты располагаются в конце блока, а не в конце операции.

Q# также предлагает еще один способ выделения кубитов, именуемый *заимствованием*. В отличие от выделения кубитов с помощью инструкций, `use` инструкция заимствования `borrow` позволяет нам заимствовать кубиты, принадлежащие разным операциям, не зная, в каком состоянии они начинаются. В этой книге мы не увидим много заимствований, но инструкция `borrow` работает очень похоже на инструкцию `use`, поскольку она не позволяет забыть о том, что мы заимствовали кубит.

По соглашению, все кубиты начинаются в состоянии $|0\rangle$ сразу после их получения, и мы обещаем целевой машине, что вернем их в состояние $|0\rangle$ в конце блока, чтобы они были готовы к тому, что целевая машина передаст их следующей операции, которая в них нуждается.

Листинг 7.2 Операция.qs: Q# для симулирования раунда игры Морганы

```
operation GetNextRandomBit() : Result {      ❶
    use qubit = Qubit();                     ❷
    H(qubit);                                ❸
    return M(qubit);                         ❹
}
```

❶ Объявляет операцию, так как мы хотим использовать кубиты и вернуть результат измерения источнику вызова.

❷ Ключевое слово use в Q# запрашивает у целевой машины один или несколько кубитов. Здесь мы запрашиваем одно значение типа Qubit, которое сохраняем в новой переменной qubit.

❸ После вызова H кубит находится в состоянии $H|0\rangle = |+\rangle$.

❹ **Использует операцию M для измерения нашего кубита в *Z*-базисе. Результат будет равен Zero (нулю) либо One (единице) с равной вероятностью. После измерения мы можем вернуть эти классические данные обратно источнику вызова.**

Здесь мы используем операцию `M` для измерения нашего кубита в *Z*-базисе, сохраняя результат в переменной `result`, которую мы объявили ранее. Поскольку состояние кубита находится в равной суперпозиции $|0\rangle$ и $|1\rangle$, результат будет равен `Zero` (нулю) либо `One` (единице) с равной вероятностью.

Автономные приложения Q#

Мы также можем писать приложения Q#, такие как QRNG-генератор, в качестве автономных приложений, вместо того чтобы вызывать их из Python или блокнотов IQ#. Для этого мы можем назначить одну из операций Q# в квантовом приложении в качестве *точки входа*, которая затем вызывается автоматически при запуске нашего приложения из командной строки.

В этой книге для работы с Q# мы будем придерживаться Jupyter и Python. Но вы можете обратиться к образцу по адресу https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp/tree/master/ch07/Qrng и получить информацию о написании автономных приложений Q#.

В листинге 7.2 мы также видим наш первый пример открытия *пространства имен* в Q#. Подобно пространствам имен в C++ и C# или пакетам в Java, пространства имен Q# помогают упорядочивать функции и операции. Например, операция `H`, вызываемая в листинге 7.2, определена в стандартной библиотеке Q# как `Microsoft.Quantum.Intrinsic.H`; т. е. она обитает в пространстве имен `Microsoft.Quantum.Intrinsic`. При применении `H` мы можем использовать ее полное имя `Microsoft.Quantum.Intrinsic.H` либо инструкцию `open`, позволяющую делать доступными все операции и функции в пространстве имен:

```
open Microsoft.Quantum.Intrinsic;          ❶
```

❶ **Выполняет все функции и операции, предоставляемые в Microsoft.Quantum.Intrinsic, доступные для использования в блокноте Q# либо исходном файле без необходимости указывать их полные имена.**

ДЛЯ СПРАВКИ При написании исходного кода Q# в среде блокнотов Jupyter Notebook пространства имен `Microsoft.Quantum.Intrinsic` и `Microsoft.Quantum.Canon` из стандартной библиотеки Q# всегда открываются для нас автоматически, так как они используются в большинстве исходного кода Q#. Например, когда мы вызывали функцию `Message` ранее, она находилась в пространстве имен `Microsoft.Quantum.Intrinsic`, которое ядро Q# открыло автоматически. Для проверки того, какие пространства имен открыты в блокноте Q#, мы можем применить волшебную команду `%lsopen`.

Упражнение 7.2: генерирование большего числа битов

Примените вольшебную команду `%simulate`, чтобы выполнить операцию `GetNextRandomBit` несколько раз. Получаете ли вы результаты, которых ожидали?

Далее мы увидим, сколько ходов Ланселоту потребуется, чтобы получить `Zero`, необходимый ему для возвращения домой. Давайте напишем операцию, чтобы играть раунды до тех пор, пока мы не получим `Zero`. Поскольку эта операция симулирует игру Морганы, мы назовем ее `PlayMorganasGame`.

ДЛЯ СПРАВКИ Все переменные в Q# по умолчанию немутируемые.

Листинг 7.3 Операции.qs: симулирование многих раундов игры Морганы с использованием языка Q#

```
operation PlayMorganasGame() : Unit {
    mutable nRounds = 0;                                    ❶
    mutable done = false;
    repeat {                                                ❷
        set nRounds = nRounds + 1;
        set done =
            (GetNextRandomBit() == Zero);                   ❸
    }
    until done;                                             ❹

    Message($"Ланселоту потребовалось {nRounds} ход(ов),
    ➥ чтобы вернуться домой.");                            ❺
}
```

❶ Использует ключевое слово mutable для объявления переменной, указывающей на число сыгранных раундов. Мы можем изменять значение этой переменной позже с помощью ключевого слова «set».
❷ Q# позволяет операциям использовать цикл повторять-до-достижения-успеха (RUS).
❸ Внутри цикла мы вызываем QRNG-генератор, который написали ранее как операцию GetNextRandomBit. Мы выполняем проверку результата на равенство Zero, и если это так, то устанавливаем done равным true.
❹ Если мы получили Zero, то можем остановить цикл.
❺ Мы снова используем функцию Message, чтобы напечатать число раундов на экране. Для этого мы используем строковые литералы $"", которые позволяют нам включать переменные в диагностическое сообщение, используя местозаполнители {} внутри строкового литерала.

ДЛЯ СПРАВКИ Строковые литералы Q#, обозначаемые символом `$""`, называются *интерполированными строковыми литералами* и работают очень похоже на строковые литералы `f""` в Python.

Листинг 7.3 включает в себя поток управления Q#, именуемый циклом повторять-до-достижения-успеха (*repeat-until-success*, аббр. *RUS*). В отличие от цикла `while`, циклы RUS также позволяют нам указывать отла-

дочный блок fixup, который выполняется, если условие выхода из цикла не удовлетворено.

Когда возникает потребность сбрасывать кубиты?

При выделении нового кубита в Q# с помощью инструкции `use` мы обещаем целевой машине, что вернем его в состояние |0⟩ перед его высвобождением. На первый взгляд это кажется ненужным, так как целевая машина может просто сбрасывать состояние кубитов, когда они высвобождаются, – в конце концов, мы часто вызываем операцию `Reset` в конце операции или блока `use`. И действительно, это происходит автоматически, когда кубит измеряется, прямо перед его высвобождением!

Важно отметить, что операция `Reset` работает путем выполнения измерения в *Z*-базисе и переворачивания кубита с помощью операции `X`, если измерение возвращает `One`. Во многих квантовых устройствах операция измерения обходится намного дороже, чем другие операции, поэтому если мы сможем избежать вызова Reset, то мы сможем снизить стоимость наших квантовых программ. Особенно учитывая ограничения среднесрочных устройств, такой вид оптимизации может иметь решающее значение для того, чтобы сделать квантовую программу практически полезной.

Позже в этой главе мы увидим примеры, в которых мы знаем состояние кубита, когда он нуждается в высвобождении, вследствие чего мы можем «откатывать подготовку» (unprepare) кубита, вместо того чтобы его измерять. В этих случаях у нас нет окончательного измерения, поэтому компилятор Q# не добавляет для нас автоматический сброс, избегая необходимости в потенциально дорогостоящей операции измерения.

Мы можем выполнить эту новую операцию с помощью команды `%simulate` в стиле, очень похожем на пример `HelloWorld`. Сделав это, мы увидим, на сколько ходов игры Ланселоту придется остаться:

```
In []: %simulate PlayMorganasGame
Ланселоту потребовалось 1 ход(ов), чтобы вернуться домой.
Out[]: ()
Похоже, в тот раз Ланселоту повезло! Или, возможно, не повезло, если ему было скучно
болтаться за столом в Камелоте.
```

Глубокое погружение: `open` против `import`

На первый взгляд инструкция `open` в Q# может показаться очень похожей на инструкцию `import` в Python, JavaScript или TypeScript. Как `open`, так и `import` делают исходный код из библиотек доступным для использования в наших программах и приложениях. Главное отличие заключается в том, что открытие пространства имен с помощью инструкции `open` делает только это: она открывает пространство имен и делает его доступным, но не приводит к выполнению какого-либо исходного кода и не изменяет способ, которым наш исходный код компилирует и производит поиск библиотек.

В принципе, мы могли бы писать каждую программу Q# без единой инструкции open, явно обращаясь к каждой функции и операции по ее полному имени, включая ее пространство имен (т. е. вызывая `Microsoft.Quantum.Intrinsic.H` вместо просто `H`). В этом смысле `open` аналогична инструкции `using` в C++ или C#, инструкции `open` в F# и инструкции `import` в Java.

Напротив, `import` в Python, JavaScript и TypeScript не только делает имена из библиотеки доступными для нашего исходного кода, но и приводит к выполнению частей этих библиотек. При первом импорте Python'овского модуля интерпретатор Python использует имя модуля, чтобы найти местоположение, где этот модуль определен, а затем выполняет модуль вместе с любым исходным кодом инициализации, который у него есть. Нередко, для того чтобы использовать эти модули, нам нужно сначала установить один или несколько *пакетов*, используя такой инструмент, как pip или conda для пакетов Python или npm для JavaScript и TypeScript.

Та же концепция может использоваться для добавления новых библиотек в наши программы Q# с помощью *менеджера пакетов NuGet*. В блокноте Q# команда %package инструктирует ядро Q# загрузить заданную библиотеку Q# и добавить ее в наш сеанс. Например, когда мы открываем новый блокнот, стандартная библиотека Q# автоматически загружается и устанавливается из пакета `Microsoft.Quantum.Standard` из nuget.org (https://www.nuget.org/packages/Microsoft.Quantum.Standard). Аналогичным образом, при написании консольных приложений Q# выполнение команды `dotnet add package` добавляет необходимые метаданные в компилятор Q# нашего проекта, чтобы отыскать нужный нам пакет.

Более подробную информацию можно получить, ознакомившись с документацией Комплекта инструментов для квантовой разработки, находящейся по адресу https://docs.microsoft.com/azure/quantum/user-guide/libraries/additional-libraries.

7.3 Передача операций в качестве аргументов

Предположим, что в игре Морганы нас интересовал отбор случайных битов с неравномерной вероятностью. В конце концов, Моргана не обещала Ланселоту рассказать, *как* она готовила кубит, который она измеряет; она может заставить беднягу играть дольше, если вместо справедливой монеты она сделает из своего кубита смещенную монету.

Модифицировать игру Морганы можно очень просто, взяв на входе операцию, воплощающую то, что Моргана делает, чтобы подготовиться к своей игре, вместо прямого вызова операции `H`. Для взятия операции на входе нам нужно написать *тип* данных на входе, подобно тому, как мы пишем `qubit : Qubit` при объявлении входного кубита `qubit` с типом `Qubit`. Типы операций обозначаются толстыми стрелками (`=>`) от типа данных на входе к типу данных на выходе. Например, операция `H` имеет тип `Qubit => Unit`, так как `H` принимает один кубит на входе и возвращает пустой кортеж на выходе.

ДЛЯ СПРАВКИ В Q# функции обозначаются тонкими стрелками (->), а операции – толстыми стрелками (=>).

Листинг 7.4 Использование операций на входе для предсказания игры Морганы

```
operation PrepareFairCoin(qubit : Qubit) : Unit {
    H(qubit);
}

operation GetNextRandomBit(
        statePreparation : (Qubit => Unit)                    ❶
) : Result {
    use qubit = Qubit();
    statePreparation(qubit);                                  ❷
    return Microsoft.Quantum.Measurement.MResetZ(qubit);      ❸
}
```

❶ Мы добавили новое входное значение под названием statePreparation в операцию GetNextRandomBit, воплотив операцию, которую мы хотим использовать для подготовки состояния, используемого нами как монета. В данном случае Qubit => Unit является типом операции, которая берет один кубит и возвращает пустой кортеж типа Unit.

❷ В операции GetNextRandomBit операция, переданная как statePreparation, может быть вызвана, как и любая другая операция.

❸ Стандартные библиотеки Q# предоставляют операцию Microsoft.Quantum.Measurement.MResetZ для удобства измерения и сбрасывания кубита за один шаг. В этом случае операция MResetZ выполняет то же самое, что и инструкция return M(qubit); в приведенном выше примере. Разница здесь в том, что операция MResetZ всегда сбрасывает свой входной кубит, а не только тогда, когда она используется перед высвобождением кубита. Подробнее об этой операции мы поговорим позже в данной главе, а также о том, как использовать более короткое имя при вызове этой операции.

Упражнение 7.3: тип операции GetNextRandomBit

Каков тип нашего нового определения операции `GetNextRandomBit`?

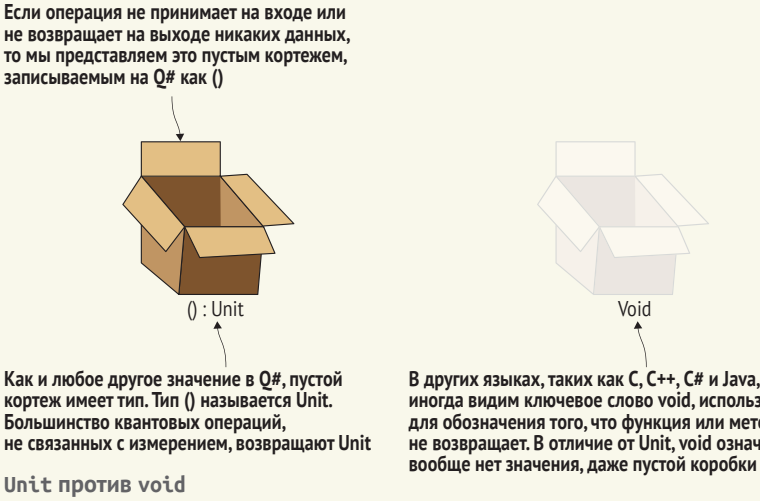
Кортеж вошел, кортеж вышел

Все функции и операции в Q# принимают один кортеж на входе и возвращают один кортеж на выходе. Например, функция, объявленная как функция `Pow(x : Double, y : Double) : Double {...}`, принимает кортеж `(Double, Double)` на входе и возвращает кортеж `(Double)` на выходе. Этот принцип работает из-за свойства, именуемого *эквивалентностью одноэлементных кортежей*. Для любого типа `'T` кортеж `('T)`, содержащий один-единственный `'T`, эквивалентен самому `'T`. В примере `Pow` мы можем рассуждать о выходном значении как о кортеже `(Double)`, который эквивалентен `Double`.

Представление операций с помощью одного входного и одного выходного значений

С учетом этого функция или операция, которая не возвращает выходных данных, может рассматриваться как возвращающая кортеж без элементов, `()`. Тип таких кортежей называется `Unit`, аналогично другим языкам, основанным на кортежах, таким как F#. Если рассуждать о кортеже как о своего рода коробке, то это отличается от void, используемого в C, C++ или C#, потому что там все же есть что-то: коробка с тем, чего нет.

В Q# мы всегда возвращаем коробку, *даже если эта коробка пуста*. В Q# нет смысла для функции или операции, которая возвращает «ничего». Более подробную информацию см. в разделе 7.2 книги «Начинаем программировать на F#» Исаака Абрахама (*Get Programming with F#*, Isaac Abraham, Manning, 2018).



`Unit` против `void`

В листинге 7.4 мы видим, что операция `GetNextRandomBit` трактует свой аргумент `statePreparation` как «черный ящик». Единственный способ узнать что-либо о стратегии подготовки Морганы – это ее *выполнить*.

Говоря по-другому, мы не хотим ничего делать с `statePreparation`, из чего вытекает, что мы знаем, что конкретно она делает или какой она является. Единственный способ, которым операция `GetNextRandomBit` может взаимодействовать с `statePreparation`, – вызвать ее, передав ей Qubit для применения к нему действия.

Это позволяет нам использовать логику в операции `GetNextRandomBit` многократно для самых разных процедур подготовки состояния, которые Моргана может использовать, чтобы создать Ланселоту немного проблем. Например, предположим, что ей нужна смещенная монета, которая возвращает `Zero` в трех четвертях случаев и `One` в одной четверти случаев. Мы могли бы выполнить что-то вроде следующего ниже, чтобы предсказать эту новую стратегию.

Листинг 7.5 Передача разных стратегий подготовки состояния в `PlayMorganasGame`

```
open Microsoft.Quantum.Math;                              (1)

operation PrepareQuarterCoin(qubit : Qubit) : Unit {
    Ry(2.0 * PI() / 3.0, qubit);                          (2)
}
```

(1) **Классические математические функции, такие как Sin, Cos, Sqrt и ArcCos, а также константы, такие как PI(), предоставляются пространством имен Microsoft.Quantum.Math.**

(2) **Операция Ry имплементирует поворот по оси *Y*. Q# использует радианы вместо градусов для выражения поворотов, и, значит, это поворот на 120° вокруг оси *Y*.**

Операция `Ry` имплементирует поворот по оси Y, которую мы встречали в главе 2. Отсюда, если при поворачивании состояния кубита qubit на 120° в листинге 7.5 кубит начинается в состоянии $|0\rangle$, то она подготавливает кубит в состоянии $R_y(120°)|0\rangle = \sqrt{3/4}\ |0\rangle + \sqrt{1/4}\ |1\rangle$. В результате вероятность наблюдать 1 при измерении кубита равна $\sqrt{1/4}^2 = 1 / 4$.

Мы можем сделать этот пример еще более общим, позволив Моргане указывать для своей монеты произвольное смещение (которое имплементируется их совместным кубитом).

Листинг 7.6 Передача операций для имплементирования произвольных смещений монеты

```
operation PrepareBiasedCoin(
    morganaWinProbability : Double, qubit : Qubit
) : Unit {
    let rotationAngle = -2.0 * ArcCos(
        Sqrt(1.0 - morganaWinProbability));                (1)
    Ry(rotationAngle, qubit);
}
operation PrepareMorganasCoin(qubit : Qubit)
: Unit {                                                   (2)
     PrepareBiasedCoin(0.62, qubit);
}
```

❶ **Выясняет, под каким углом повернуть входной кубит, чтобы получить правильную вероятность увидеть One в качестве результата. Это потребует немного тригонометрии; см. следующую ниже врезку для получения подробной информации.**

❷ **Эта операция имеет правильную сигнатуру типа (Qubit => Unit), и мы видим, что вероятность выигрыша Морганы в каждом раунде составляет 62 %.**

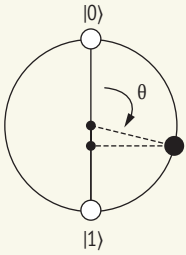
Разработка тригонометрии

Как мы уже видели несколько раз, в квантовых вычислениях расширенно задействуются повороты. Для того чтобы выяснить, какие углы нам нужны для наших поворотов, мы опираемся на тригонометрию (буквально «науку о треугольниках»), т. е. раздел математики, который описывает углы поворота. Например, как мы видели в главе 2, поворачивание $|0\rangle$ на угол θ вокруг оси Y приводит к состоянию $\cos(-\theta/2)\,|0\rangle + \sin(-\theta/2)\,|1\rangle$. Мы знаем, что хотим выбрать θ такой, что $\cos(-\theta/2) = \sqrt{100\,\% - 62\,\%}$, поэтому мы получаем 62%-ную вероятность иметь результат One. Это означает, что нам нужно «откатить» функцию косинуса, чтобы выяснить, каким должен быть θ .

В тригонометрии обратной косинусу функцией является функция арккосинус и записывается как arccos. Взяв арккосинус обеих сторон уравнения $\cos(-\theta/2) = \sqrt{100\,\% - 62\,\%}$, мы получаем $\arccos(\cos(-\theta/2)) = \arccos(\sqrt{100\,\% - 62\,\%})$. Мы можем выполнить взаимоотмену arccos и cos и найти угол поворота, который дает нам то, что нам нужно, $-\theta/2 = \arccos(\sqrt{100\,\% - 62\,\%})$.

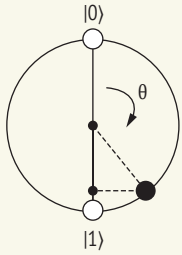
Наконец, мы умножаем обе стороны на –2 и получаем уравнение, используемое в строке 4 листинга 7.6. Эта аргументация наглядно показана на следующем ниже рисунке.

Если измерение по оси *Z* приведет к нулю, то Ланселот победит и сможет вернуться домой



Моргана может контролировать степень вероятности получить результат один путем выбора θ, угла, под которым она поворачивает вокруг оси *Y*

Если Моргана выберет θ, близкий к π/2 (90°), то в результате ноль и единица будут приблизительно равновероятными



С другой стороны, чем ближе Моргана выбирает к (180°), тем ближе проекция состояния на ось *Z* к |1⟩

Если она выберет θ в точности равным π, то в результате всегда будет получаться единица: она может продолжать играть с Ланселотом до бесконечности

Если измерение по оси *Z* приведет к единице, то Моргана выиграет, и Ланселот должен сыграть еще один раунд

Принцип, по которому Моргана может выбирать θ для управления игрой. Чем ближе к π она выберет θ, тем дольше Моргана сможет держать бедного Ланселота в игре

Однако это несколько неудовлетворительно, потому что операция `PrepareMorganasCoin` вводит много шаблонного кода только для того, чтобы изолировать значение `0.62` для входного аргумента `morganaWinProbability` для операции `PrepareBiasedCoin`. Если Моргана изменит свою

стратегию, чтобы иметь другое смещение, то, используя этот подход, нам понадобится еще одна шаблонная операция, чтобы ее представить. Сделав шаг назад, давайте посмотрим, что на самом деле *делает* операция `PrepareMorganasCoin`. Она начинается с `PrepareBiasedCoin : (Double, Qubit) => Unit` и обертывает ее в операцию типа `Qubit => Unit`, изолируя аргумент типа `Double` как `0.62`. То есть он удаляет один из аргументов для операции `PrepareBiasedCoin`, исправляя это входное значение на `0.62`.

К счастью, язык Q# обеспечивает удобную укороченную нотацию для создания новых функций и операций, изолируя некоторые (но не все!) значения на входе. Используя эту укороченную нотацию, именуемую *частичным применением*, мы можем переписать операцию `PrepareMorganasCoin` из листинга 7.6 в более удобочитаемой форме:

```
let flip = GetNextRandomBit(PrepareBiasedCoin(0.62, _));
```

Символ `_` указывает на то, что часть входных данных для операции `PrepareBiasedCoin` *отсутствует*. Мы говорим, что операция `PrepareBiasedCoin` была применена частично. В отличие от типа `(Double, Qubit) => Unit`, который имела операция `PrepareBiasedCoin`, теперь, после заполнения части `Double` входных аргументов, операция `PrepareBiasedCoin(0.62, _)` имеет тип `Qubit => Unit`, что делает ее совместимой с нашими модификациями операции `GetNextRandomBit`.

ДЛЯ СПРАВКИ Частичное применение в Q# похоже на функцию `functools.partial` в Python и ключевое слово `_` в Scala.

На частичное применение можно взглянуть и по-другому, как на способ создания новых функций и операций путем конкретизации существующих функций и операций.

Листинг 7.7 Использование частичного применения для конкретизации операции

```
function BiasedPreparation(headsProbability : Double)
: (Qubit => Unit) {                          ❶
    return PrepareBiasedCoin(
        headsProbability, _);                ❷
}
```

❶ Выходной тип представляет собой операцию, которая берет кубит и возвращает пустой кортеж. То есть BiasedPreparation – это функция, которая создает новые операции!

❷ Создает новую операцию, передавая headsProbability, но оставляя местозаполнитель (_) для целевого кубита. В результате мы получаем операцию, которая берет один кубит и подставляет его в местозаполнитель.

ДЛЯ СПРАВКИ В листинге 7.7 мы возвращаем `PrepareBiasedCoin(headsProbability, _)` как самостоятельное значение, подобно тому, как мы можем вернуть `42` из функции или операции с типом `Int` на выходе. В Q# функции и операции являются значениями

в том же самом смысле, что и `42`, `true` и `(3.14, "привет")`, и, подобно Python'овской функции `(lambda x: x ** 2)`, являются значением. Формально мы говорим, что в Q# функции и операции являются *первоклассными значениями*.

То, что функция `BiasedPreparation` возвращает операцию из функции, возможно, кого-то смутит, но это полностью согласуется с разделением между описанными ранее функциями и операциями, поскольку функция `BiasedPreparation` по-прежнему предсказуема. В частности, вызов `Biased-Preparation(p)` всегда возвращает одну и ту же операцию для данного `p`, независимо от того, сколько раз мы вызываем эту функцию. Мы можем в этом убедиться, заметив, что функция `BiasedPreparation` всего лишь частично применяет операции, но никогда их не вызывает.

Упражнение 7.4: частичное применение

Частичное применение работает как для функций, так и для операций! Попробуйте это сделать, написав функцию `Plus`, которая складывает два целых числа, `n` и `m`, и еще одну функцию `PartialPlus`, которая на входе принимает `n` и возвращает функцию, которая прибавляет `n` к своему входу.

Подсказка: вы можете начать использовать следующий ниже фрагмент кода в качестве заготовки:

```
function Plus(n : Int, m : Int) : Int {
    // заполнить эту часть
}
function PartialPlus(n : Int) : (Int -> Int) {
    // заполнить эту часть
}
```

7.4 Игра Морганы на Q#

С первоклассными операциями и частичным применением наготове теперь мы можем написать более полную версию игры Морганы.

Стандартные библиотеки Q#

Комплект инструментов для квантовой разработки поставляется с различными стандартными библиотеками, которые мы увидим в остальной части книги. Например, в листинге 7.8 мы используем операцию `MResetZ`, которая одновременно измеряет кубит (аналогично `M`) и сбрасывает его (аналогично `Reset`). Эта операция предлагается пространством имен `Microsoft.Quantum.Measurement`, одной из главных стандартных библиотек, входящих в Комплект инструментов для квантовой разработки.

Полный список операций и функций, доступных в этом пространстве имен, можно найти по адресу https://docs.microsoft.com/qsharp/api/qsharp/microsoft.quantum.measurement. На данный момент, однако, не беспокойтесь об этом слишком сильно; мы увидим больше стандартных библиотек Q# по мере продвижения.

Листинг 7.8 Полный листинг на Q# для смещенной игры PlayMorganasGame

```
open Microsoft.Quantum.Math;                                    ❶
open Microsoft.Quantum.Measurement;

operation PrepareBiasedCoin(winProbability : Double, qubit : Qubit)
: Unit {
    let rotationAngle = 2.0 * ArcCos(
        Sqrt(1.0 - winProbability));                            ❷
    Ry(rotationAngle, qubit);
}

operation GetNextRandomBit(statePreparation : (Qubit => Unit))
: Result {
    use qubit = Qubit();
    statePreparation(qubit);                                    ❸
    return MResetZ(qubit);                                      ❹
}

operation PlayMorganasGame(winProbability : Double) : Unit {
    mutable nRounds = 0;
    mutable done = false;
    let prep = PrepareBiasedCoin(
        winProbability, _);                                     ❺
    repeat {
        set nRounds = nRounds + 1;
        set done = (GetNextRandomBit(prep) == Zero);
    }
    until done;

    Message($"Ланселоту потребовалось {nRounds} ход(ов), чтобы вернуться домой.");
}
```

❶ Открывает пространства имен из стандартной библиотеки Q#, чтобы помочь с классической математикой и измерением кубитов.

❷ Угол поворота определяет смещение монеты.

❸ Использует операцию, которую мы передали как statePreparation, и применяет ее к кубиту.

❹ Операция MResetZ определена в пространстве имен Microsoft.Quantum.Measurement, которое мы открываем в начале образца. Она измеряет кубит в *Z*-базисе, а затем применяет операции, необходимые для возвращения кубита в состояние |0⟩.

❺ Использует частичное применение, для того чтобы указать смещение для нашей процедуры подготовки состояния, но не целевой кубит. В то время как операция PrepareBiasedCoin имеет тип (Double, Qubit) => Unit, выражение PrepareBiasedCoin(0.2, _) «заполняет» один из двух входных аргументов, оставляя в итоге операцию с типом Qubit => Unit, который EstimateBias ожидает получить на входе.

Документирование функций и операций Q#

Документирование функций и операций Q# обеспечивается путем написания небольших, специально отформатированных текстовых документов в комментариях с тройной косой чертой (///) перед объявлением функции или операции. Эта документация пишется в формате упрощенной разметки Markdown, простом языке форматирования текста, используемом на таких сайтах, как GitHub, Azure DevOps, Reddit и Stack Exchange, а также генераторами веб-сайтов, такими как Jekyll. Информация в комментариях /// выводится на экран при наведении курсора мыши на вызовы этой функции или операции и может использоваться для создания справочных материалов по API, аналогичных материалам на https://docs.microsoft.com/qsharp/api/.

Различные части комментариев /// обозначаются заголовками разделов, такими как /// # Сводная информация. Например, мы можем задокументировать операцию PrepareBiasedCoin из листинга 7.8 следующим образом:

```
/// # Сводная информация
/// Подготавливает состояние, представляющее монету с заданным смещением.
///
/// # Описание
/// Получая кубит, первоначально находящийся в состоянии |0⟩, применяет операции
/// к этому кубиту, такому что он имеет состояние √p |0⟩ + √(1 − p) |1⟩,
/// где p предоставляется на входе.
/// Измерение этого состояния возвращает результат One с вероятностью p.
///
/// # Вход
/// ## winProbability
/// Вероятность, с которой измерение кубита должно вернуть One.
/// ## qubit
/// Кубит, на котором нужно подготовить состояние √p |0⟩ + √(1 − p) |1⟩.
operation PrepareBiasedCoin(
        winProbability : Double, qubit : Qubit
) : Unit {
    let rotationAngle = 2.0 * ArcCos(Sqrt(1.0 - winProbability));
    Ry(rotationAngle, qubit);
}
```

При использовании IQ# мы можем просматривать документационные комментарии с помощью команды ?. Например, мы можем просмотреть документацию по операции X, выполнив X? во входной ячейке.

Для получения полной справочной информации обратитесь по адресу: https://docs.microsoft.com/azure/quantum/user-guide/language/statements/iterations#documentation-comments.

Для того чтобы оценить смещение конкретной операции подготовки состояния, мы можем выполнить операцию PlayMorganasGame многократно и подсчитать количество полученных нами нулей (Zero). Давайте выберем значение для winProbability и выполним операцию PlayMorganasGame с этим значением, чтобы посмотреть, насколько долго Ланселот застрянет:

```
In []: %simulate PlayMorganasGame winProbability=0.9                    ❶
Ланселоту потребовалось 5 ход(ов), чтобы вернуться домой.
```

❶ **Передавать входные данные в операции можно с помощью команды `%simulate`, указывая их после имени операции, которую мы хотим просимулировать.**

Попробуйте поиграть с разными значениями вероятности выигрыша, `winProbability`. Обратите внимание, что если Моргана действительно склонит чашу весов в свою пользу, то мы можем подтвердить, что Ланселоту потребуется довольно много времени, чтобы вернуться к Гвиневре:

```
In []: %simulate PlayMorganasGame winProbability=0.999
Ланселоту потребовалось 3255 ход(ов), чтобы вернуться домой.
```

В следующей главе мы будем развивать полученные здесь навыки, вернувшись в Камелот, чтобы найти наш первый пример квантового алгоритма: алгоритм Дойча–Йожи.

Резюме

- Q# – это язык квантового программирования, поставляемый вместе с Комплектом инструментов с открытым исходным кодом для квантовой разработки от Microsoft.
- Квантовые программы на Q# подразделены на функции, представляющие классическую и детерминированную логику, и операции, которые могут иметь побочные эффекты, такие как отправка инструкций квантовым устройствам.
- Функции и операции в Q# являются первоклассными значениями и могут передаваться на вход других функций и операций. Мы можем использовать этот механизм для комбинирования различных частей квантовых программ.
- Передавая операции Q# в нашей программе, мы можем расширить наш пример QRNG из главы 2, чтобы разрешить передачу операций, которые готовят состояния, отличные от $|+\rangle$; это, в свою очередь, позволяет создавать смещенные случайные числа.

8 Что такое квантовый алгоритм

Эта глава охватывает следующие ниже темы:

- понимание того, что такое квантовый алгоритм;
- разработка оракулов для представления классических функций в квантовых программах;
- работа с полезными техническими приемами квантового программирования.

Одним из важных применений квантовых алгоритмов является получение ускорений для решения задач, в которых нам нужно искать входные данные для функции, которую мы пытаемся усвоить. Такие функции могут быть туманными и сбивать с толку (например, хеш-функции) или вычислительно трудными для оценивания (что распространено при изучении математических задач). В любом случае применение квантовых компьютеров к таким задачам требует от нас понимания того, как мы программируем и предоставляем входные данные для квантовых алгоритмов. В целях ознакомления с тем, как это делается, мы запрограммируем и выполним имплементацию *алгоритма Дойча–Йожи*, который позволит нам быстро усваивать свойства неизвестных функций с помощью квантовых устройств.

8.1 Классические и квантовые алгоритмы

> *Алгоритм (сущ.): пошаговая процедура решения задачи либо достижения какой-либо цели.*
>
> – Словарь Merriam-Webster

Говоря о классическом программировании, мы иногда говорим, что программа имплементирует *алгоритм*, а именно последовательность шагов, которые используются для решения задачи. Например, если мы хотим отсортировать список, то мы можем говорить об алгоритме быстрой сортировки (quicksort) независимо от того, какой язык или операционную систему мы используем. Мы часто указываем эти шаги на высоком уровне. В примере быстрой сортировки мы могли бы перечислить шаги примерно следующим образом.

1. Если сортируемый список пуст или содержит только один элемент, то вернуть его как есть.
2. Выбрать элемент списка для сортировки, именуемый *опорным*.
3. Подразделить все остальные элементы списка на те, которые меньше опорного элемента, и те, которые больше его.
4. Рекурсивно выполнить быструю сортировку каждого нового списка.
5. Вернуть первый список, затем опорный элемент и, наконец, второй список.

Эти шаги служат руководством для написания имплементации на конкретном интересующем нас языке. Скажем, мы хотим написать алгоритм быстрой сортировки на Python.

Листинг 8.1 Пример имплементации алгоритма quicksort

```
def quicksort(xs):
    if len(xs) > 1:                                  (1)
        pivot = xs[0]                                (2)
        left = [x in xs[1:] if x <= pivot]           (3)
        right = [x in xs[1:] if x > pivot]
        return quicksort(left) +
➥ [pivot] + quicksort(right)                        (4)
    else:
        return xs
```

(1) Проверяет наличие базового случая, глядя, есть ли в списке по крайней мере два элемента или нет.
(2) Выбирает первый элемент, который будет нашим опорным элементом для шага 2.
(3) Код Python, который создает два новых списка, как описано в шаге 3.
(4) Конкатенирует все вместе, как описано в шагах 4 и 5.

Хорошо написанный алгоритм помогает в написании имплементаций, создавая четкие шаги выполнения. Квантовые алгоритмы в этом отношении одинаковы: они перечисляют шаги, которые мы должны выполнить в любой имплементации.

ОПРЕДЕЛЕНИЕ *Квантовая программа* – это имплементация квантового алгоритма, состоящего из *классической программы*, которая посылает инструкции *квантовому устройству* для подготовки определенного состояния или результата измерения.

Как мы видели в главе 7, при написании программы на Q# мы пишем классическую программу, которая отправляет инструкции одной из нескольких разных целевых машин от нашего имени, как показано на рис. 8.1, возвращая измерения в нашу классическую программу.

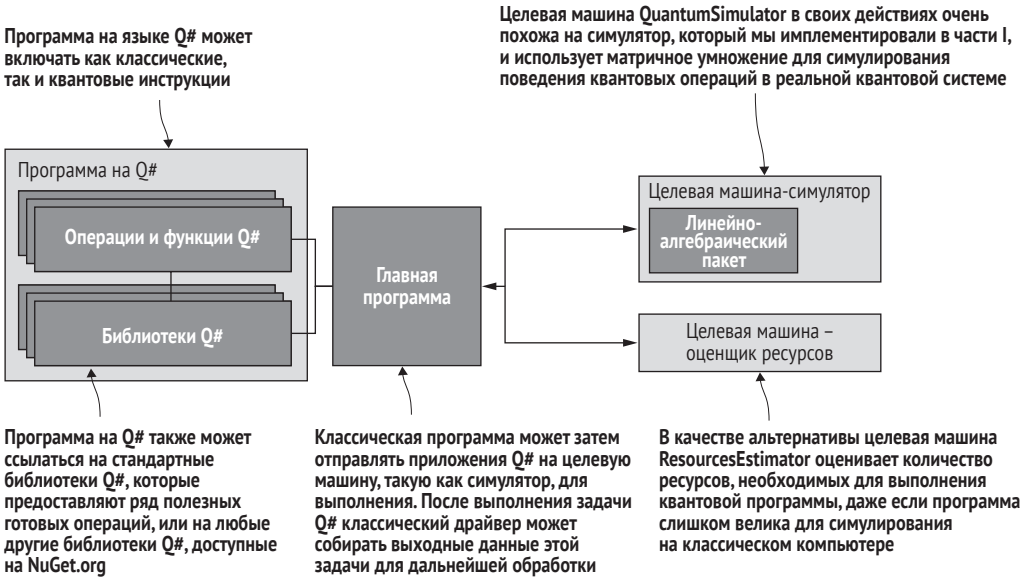


Рис. 8.1 Программный стек Комплекта инструментов для квантовой разработки от Microsoft на классическом компьютере. Мы можем написать программу на Q#, состоящую из функций и операций и ссылающуюся на любые библиотеки Q#, которые мы хотим включить. Затем главная программа может координировать обмен информацией между нашей программой на Q# и целевой машиной (например, симулятором, работающим локально на нашем компьютере)

Искусство квантового программирования

Мы не можем копировать квантовые состояния, но если они возникли в результате выполнения программы, то мы можем сказать кому-то другому, какие шаги ему необходимо предпринять для подготовки тех же состояний. Как мы видели ранее, квантовые программы – это особый вид классических программ, поэтому мы можем копировать их с безрассудной самоотверженностью. Как мы увидим в остальной части книги, любое квантовое состояние может быть либо аппроксимировано, либо расписано точно результатом на выходе из квантовой программы, которая начинается только с копий состояния $|0\rangle$. Например, в главе 2 мы подготовили начальное состояние $|+\rangle$ QRNG-генератора с помощью программы, состоящей из одной-единственной инструкции H.

Говоря по-другому, мы можем рассуждать о программе как о рецепте приготовления кубита. Имея кубит, мы не можем определить, какой рецепт был использован для его приготовления, при этом мы можем копировать сам рецепт столько, сколько нам нравится.

В то время как шаги при исполнении программы `quicksort` предписывают интерпретатору Python сравнивать значения и перемещать их в памяти, шаги в программе на Q# предписывают нашим целевым машинам применять повороты и измерения к кубитам на устройстве. Как показано на рис. 8.1, мы можем использовать главную программу для отправки приложений Q# на каждую отдельную целевую машину для ее выполнения. На данный момент мы продолжим использовать плагин IQ# для среды блокнотов Jupyter Notebook в качестве главной программы; в следующей главе мы научимся использовать Q# для написания наших собственных главных программ.

В этой книге нас по большей части интересует симулирование квантовых программ, поэтому мы используем целевую машину QuantumSimulator. Этот симулятор работает очень похоже на те, которые мы разработали в главах 2 и 4, поскольку он выполняет такие инструкции, как инструкция Адамара, путем умножения квантовых состояний с унитарными операторами, такими как *H*.

ДЛЯ СПРАВКИ Как и в предыдущих главах, мы используем шрифты, чтобы отличать инструкции, подобные `H`, от унитарных матриц, подобных *H*, которые мы используем для симулирования этих инструкций.

Целевая машина `ResourcesEstimator` позволяет нам не выполнять квантовую программу, а получать оценки числа кубитов, которые потребуются для ее выполнения. Это полезно для более крупных программ, которые невозможно просимулировать классически или выполнить на доступном оборудовании, чтобы помочь нам узнать количество требуемых для этого кубитов; мы узнаем больше об этой целевой машине позже.

Поскольку приложения Q# отправляют инструкции на целевые машины, которые мы используем для их выполнения, не представляет труда использовать исходный код Q# многократно позже на разных целевых машинах, в которых используется один и тот же набор инструкций. Например, в целевой машине `QuantumSimulator` используются те же инструкции, которые, как мы ожидаем, будут выполняться реальным квантовым оборудованием, как только оно станет доступным, и, следовательно, мы можем тестировать написанные на Q# программы на симуляторах уже сейчас, используя небольшие примеры задач, а затем выполнять те же программы на квантовом оборудовании позже.

В этих разных целевых машинах и приложениях общим остается то, что нам нужно написать программу, которая отправляет инструкции на целевую машину для достижения какой-то цели. Следовательно, наша

задача как квантовых программистов состоит в обеспечении того, чтобы эти инструкции имели эффект решения полезной задачи.

> **ДЛЯ СПРАВКИ** Подход, на основе которого мы используем симуляторы для тестирования программ Q#, немного похож на то, как мы используем симуляторы для тестирования программ для другого специализированного оборудования, такого как программируемые в условиях эксплуатации вентильные массивы (*field-programmable gate arrays*, аббр. *FPGA*) или эмуляторы для тестирования приложений для мобильных устройств с наших настольных компьютеров и ноутбуков. Главное различие заключается в том, что в случае симулирования квантового компьютера мы можем использовать классический компьютер только для очень небольшого числа кубитов или ограниченных видов программ.

Это делать гораздо проще, когда у нас есть алгоритм, направляющий нас в организации шагов, которые должны осуществляться в классических и квантовых устройствах. При разработке новых квантовых алгоритмов мы можем использовать квантовые эффекты, такие как запутывание, которые мы встречали в главе 4.

> **ДЛЯ СПРАВКИ** В целях получения каких-либо преимуществ от нашего квантового оборудования мы *обязаны* использовать уникальные квантовые свойства оборудования. В противном случае у нас просто будет более дорогой, более медленный классический компьютер.

8.2 Алгоритм Дойча–Йожи: умеренные улучшения для проведения поиска

Итак, что может стать хорошим примером *квантового алгоритма*, в котором используются преимущества нашего блестящего нового квантового оборудования? В главах 4 и 7 мы узнали, что размышления об играх часто помогают получить ответ на данный вопрос, и это не исключение. В целях отыскания подходящей игры для этой главы давайте вернемся в Камелот, где Мерлин сталкивается с испытанием.

8.2.1 Владычица (квантового) озера

Мерлин, знаменитый и мудрый маг, только что встретился с Нимуэ, Владычицей озера. Нимуэ, ищущая способного наставника для следующего короля Англии, решила проверить способность Мерлина справиться с этой задачей. Два непримиримых соперника, Артур и Мордред, борются за трон, и если Мерлин примет вызов Нимуэ, то он должен выбрать, кого из них наставлять в качестве короля.

Со своей стороны Нимуэ все равно, кто станет королем, коль скоро Мерлин может давать им мудрые советы. Но ее беспокоит вопрос, будет ли Мерлин, назначенный наставником нового короля, надежным и последовательным в своем руководстве.

Поскольку Нимуэ разделяет нашу страсть к играм, она решила поиграть в игру с Мерлином, чтобы проверить, что он будет хорошим наставником. Игра Нимуэ «Воспитатель короля» проверяет, насколько Мерлин *последователен* в своей роли советника короля. Для того чтобы сыграть в Воспитателя короля, Нимуэ дает Мерлину имя одного из двух непримиримых соперников, и Мерлин должен ответить, должен кандидат Нимуэ быть истинным наследником трона или нет. Правила таковы:

- в каждом раунде Нимуэ задает Мерлину один-единственный вопрос в форме «Должен ли *потенциальный наследник* стать королем?»;
- Мерлин должен ответить либо «да», либо «нет», не сообщая никакой дополнительной информации.

Каждый раунд дает Нимуэ больше информации о царстве смертных, поэтому она хочет задать как можно меньше вопросов, чтобы уличить Мерлина, если он не заслуживает доверия. Ее цели заключаются в следующем:

- убедиться, что Мерлин будет хорошим наставником для нового короля Англии;
- задать как можно меньше вопросов для проверки;
- избегать знаний о том, кому Мерлин скажет «да» на вопрос о наставничестве.

На данный момент у Мерлина есть четыре возможные стратегии:

1. сказать «да», когда ему задается вопрос о том, должен ли Артур стать королем, и «нет» в противном случае (хороший наставник);
2. сказать «да», когда ему задается вопрос о том, должен ли Мордред стать королем, и «нет» в противном случае (хороший наставник);
3. сказать «да» независимо от того, о ком Нимуэ спрашивает (плохой наставник);
4. сказать «нет» независимо от того, о ком Нимуэ спрашивает (плохой наставник).

Мы можем рассуждать о стратегиях Мерлина, снова воспользовавшись понятием таблицы истинности. Предположим, например, что Мерлин решил быть исключительно бесполезным и отрицать права на трон любого кандидата. Мы могли бы записать это, используя таблицу истинности, как в табл. 8.1.

Таблица 8.1 Таблица истинности одной из возможных стратегий Воспитателя короля

Вход (Нимуэ)	Выход (Мерлин)
Должен ли Мордред стать королем?	Нет
Должен ли Артур стать королем?	Нет

В этот момент Нимуэ была бы права, если бы пожаловалась на мудрость Мерлина как наставника! Мерлин не был последователен в исполнении поручения выбирать между Артуром и Мордредом. Хотя Нимуэ, возможно, все равно, кого выберет Мерлин, он, безусловно, должен выбрать *кого-то* для наставничества и подготовки к трону.

Нимуэ нужна стратегия, чтобы определить за минимальное число раундов игры, какая у Мерлина стратегия: стратегия 1 или 2 (хороший наставник) либо Мерлин играет в соответствии со стратегией 3 или 4 (плохой наставник). Она могла бы просто задать оба вопроса: «Должен ли Мордред стать королем?» и «Должен ли Артур стать королем?», а затем сравнить его ответы, но это привело бы к тому, что Нимуэ будет точно знать, кого он выбрал королем. С каждым вопросом владычица Нимуэ все больше узнаёт о смертных делах королевства – какая неприятность!

Хотя, казалось бы, игра Нимуэ обречена на то, чтобы заставлять ее узнавать о выборе Мерлином наследника, ей все же повезло. Поскольку речь идет о квантовом озере, в оставшейся части этой главы мы увидим, что Нимуэ имеет возможность задать *единственный* вопрос, который даст ей ответ, *только* если Мерлин привержен своей роли наставника, а не кого он выбрал.

Поскольку в нашем распоряжении нет квантового озера, давайте попробуем смоделировать то, что Нимуэ делает с квантовыми инструкциями Q# на нашем классическом компьютере, а затем это просимулируем. Представим стратегию Мерлина классической функцией *f*, которая принимает вопрос Нимуэ как *x* на входе. То есть мы напишем *f*(Артур) в значении «Что отвечает Мерлин, когда его спрашивают, должен ли Артур стать королем». Обратите внимание, что, поскольку Нимуэ задаст только один из двух вопросов, вопрос, который она задаст, является примером бита. Иногда удобно записывать этот бит, используя метки «0» и «1», в других же случаях полезно помечать входной бит Нимуэ, используя булевы значения `False` и `True`. В конце концов, ответ «1» был бы довольно странным на такой вопрос, как «Должен ли Мордред стать королем?».

Используя биты, мы пишем *f*(0) = 0, что означает, что если Нимуэ спросит Мерлина «Должен ли Мордред стать королем?», то его ответом будет «Нет». В табл. 8.2 показано, каким образом можно соотнести вопросы Нимуэ с булевыми значениями.

Таблица 8.2 Кодирование вопроса Нимуэ в виде бита

Вопрос Нимуэ	Представление в виде бита	Представление в виде булева значения
Должен ли Мордред стать королем?	0	`False`
Должен ли Артур стать королем?	1	`True`

Если бы у нее не было никаких квантовых ресурсов, чтобы убедиться в стратегии Мерлина, Нимуэ пришлось бы пробовать в *f* оба входных значения; т. е. ей пришлось бы задать Мерлину оба вопроса.

Перепробовав все входные данные, она получила бы полную стратегию Мерлина; и, как уже отмечалось, Нимуэ на самом деле в этом не заинтересована.

Вместо того чтобы спрашивать Мерлина и о Мордреде, и об Артуре, мы можем имплементировать квантовый алгоритм на Q#, в котором используются квантовые эффекты, чтобы узнать, каким наставником, хорошим либо плохим, является Мерлин, задав ему только *один* вопрос. Используя симуляторы, поставляемые с Комплектом инструментов для квантовой разработки, мы даже можем выполнить нашу новую программу Q# на наших ноутбуках или настольных компьютерах! В остальной части этой главы мы рассмотрим пример написания этого квантового алгоритма, именуемого алгоритмом Дойча–Йожи (см. рис. 8.2).

Рис. 8.2 В этой главе мы будем работать в программном стеке Комплекта инструментов для квантовой разработки от Microsoft, составляя программы на Q#, которые выполняются через главную программу в среде блокнотов Jupyter Notebook на целевой машине-симуляторе

Давайте сделаем набросок того, как будет выглядеть наша квантовая программа. Возможными входами и выходами для *f*(стратегия Мерлина) является `True` и `False`. Мы можем написать таблицу истинности для *f*, используя входы и выходы, которые мы получаем при вызове *f*. Например, если *f* является классической операцией NOT (часто обозначаемой как ¬), то мы увидим, что *f*(`True`) равно `False`, и наоборот. Как показано в табл. 8.3, использование классической операции NOT в качестве стратегии в нашей игре соответствует выбору королем Мордреда.

Таблица 8.3 Таблица истинности для классической операции NOT

Вход	Выход
`True`(Должен ли Артур стать королем?)	`False`(«Нет»)
`False`(Должен ли Мордред стать королем?)	`True`(«Да»)

Существует четыре возможных варианта определения нашей функции *f*, каждый из которых представляет одну из четырех стратегий, доступных Мерлину, как показано на рис. 8.3.

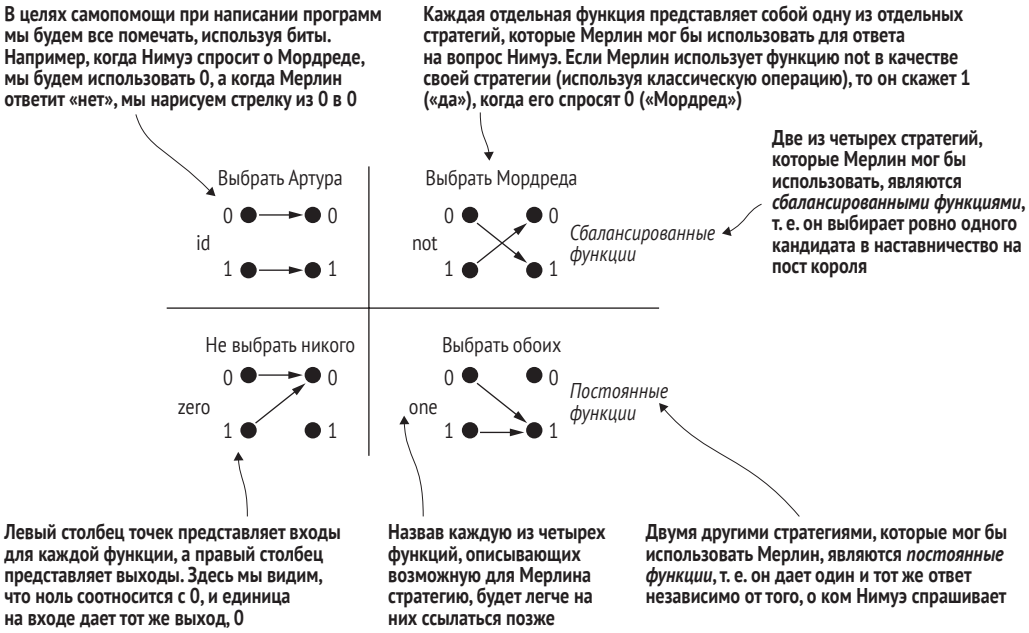


Рис. 8.3 Четыре разные функций из одного бита в один бит. Мы называем две функции в верхней строке сбалансированными, потому что равновеликое число входов соотносится с 0, как и число входов, которое соотносится с 1. Мы называем две функции в нижней строке постоянными, поскольку все входы соотносятся с одним-единственным выходом

Две из этих функций, для удобства помеченные `id` и `not`, отправляют каждый из двух входов 0 и 1 на разные выходы; мы называем эти функции *сбалансированными*. В нашей маленькой игре они представляют случаи, когда Мерлин выбирает в качестве короля ровно одного человека. Все случаи перечислены в табл. 8.4.

Таблица 8.4 Классифицирование стратегий Мерлина как постоянных либо сбалансированных

Стратегия Мерлина	Функция	Тип	Проходит вызов Нимуэ?
Выбрать Артура	`id`	Сбалансированая (f(0) ≠ f(1))	Да
Выбрать Мордреда	`not`	Сбалансированая (f(0) ≠ f(1))	Да
Никого не выбрать	`zero`	Постоянная (f(0) = f(1))	Нет
Выбрать обоих	`one`	Постоянная (f(0) = f(1))	Нет

С другой стороны, функции, которые мы называем `zero` и `one`, являются *постоянными* функциями, поскольку они отправляют оба входа на один и тот же выход. Постоянные функции тогда представляют стратегии, в которых Мерлин решительно бесполезен, поскольку в качестве короля он либо выбрал обоих (хороший способ начать плохую войну), либо не выбрал ни того, ни другого.

В классическом плане, в целях определения того, какой, постоянной либо сбалансированной, функция является (Мерлин соответственно –

плохой либо хороший наставник), мы должны знать всю функцию, построив ее таблицу истинности. Напомним, что Нимуэ хочет убедиться, что Мерлин – надежный наставник. Если Мерлин следует стратегии, представленной постоянной функцией, то он не будет хорошим наставником. Глядя на таблицы истинности для функций `id` и `one`, соответственно табл. 8.5 и 8.6, мы видим, как они описывают случай, когда Мерлин следует стратегии, которая позволит ему быть хорошим либо плохим наставником.

Таблица 8.5 Таблица истинности для функции id, пример сбалансированной функции

Вход	Выход
`True`(Должен ли Артур стать королем?)	`True`(«Да»)
`False`(Должен ли Мордред стать королем?)	`False`(«Нет»)

Таблица 8.6 Таблица истинности для функции one, пример постоянной функции

Вход	Выход
`True`(Должен ли Артур стать королем?)	`True`(«Да»)
`False`(Должен ли Мордред стать королем?)	`True`(«Да»)

Затруднение Нимуэ в попытке узнать, каким наставником, хорошим либо плохим, является Мерлин (т. е. является ли *f* сбалансированной либо постоянной), заключается в том, что качество наставничества Мерлина является своего рода *глобальным свойством* его стратегии. Невозможно посмотреть на один-единственный выход из *f* и понять, что именно *f* будет выдавать на выходе для разных входов. Если у нас есть доступ только к *f*, то Нимуэ застряла: она должна восстановить всю таблицу истинности, чтобы решить, какой из двух, постоянной либо сбалансированной, является стратегия Мерлина.

С другой стороны, если мы сможем представить стратегию Мерлина как часть квантовой программы, то мы сможем применить квантовые эффекты, о которых мы узнали в книге ранее. Используя квантовые вычисления, Нимуэ сможет получать информацию только в том случае, если его стратегия является постоянной либо сбалансированной, без необходимости знать точно, какую стратегию он использует. Поскольку нас не интересует информация, содержащаяся в таблице истинности, помимо того, каким наставником, хорошим либо плохим, является Мерлин, использование квантовых эффектов поможет нам узнать то, что нас волнует более непосредственно. С помощью нашего квантового алгоритма мы можем сделать это одним вызовом функции и без необходимости узнавать какую-либо дополнительную информацию, которая нас не интересует. Не спрашивая всех деталей таблицы истинности, а только ища более общие свойства нашей функции, мы можем эффективно задействовать наши квантовые ресурсы.

Мощь квантовых вычислений

Если мы хотим применить *классический компьютер*, чтобы узнать, с какой функцией мы имеем дело: постоянной либо сбалансированной, мы должны сначала решить более сложную задачу: определить, какая именно функция у нас есть. Напротив, квантовая механика позволяет нам решать только ту задачу, которая нас волнует (постоянная либо сбалансированная), не решая более сложную задачу, которую должен решать классический компьютер.

Это пример закономерности, которую мы будем встречать на протяжении всей книги, в которой квантовая механика позволяет нам определять менее мощные алгоритмы, чем мы можем выразить классически.

Для этого мы применим алгоритм Дойча–Йожи, в котором используется *один-единственный запрос* к нашему квантовому представлению стратегии Мерлина, чтобы узнать, каким наставником, хорошим либо плохим, он является. Это преимущество не очень практично (экономия только на одном вопросе), но это нормально; мы увидим в книге более практичные алгоритмы позже. На данный момент алгоритм Дойча–Йожи является отличным местом, чтобы начать учиться имплементировать квантовые алгоритмы и, что еще важнее, определять инструменты, которые мы можем использовать для понимания принципа работы квантовых алгоритмов.

8.3 Оракулы: представление классических функций в квантовых алгоритмах

Давайте посмотрим, как все выглядит из квантового озера Нимуэ. Когда мы плюхаемся в воду, чтобы поплавать, мы сталкиваемся с довольно непосредственным вопросом: как использовать кубиты для имплементирования функции f, которая представляет стратегию Мерлина? Из предыдущего раздела мы узнали, что классическая функция f является нашим описанием стратегии, которую Мерлин использует для игры в каждом раунде Воспитателя короля. Поскольку f является классической, легко перевести ее обратно в набор действий, которые предпримет Мерлин: Нимуэ дает Мерлину один классический бит (свой вопрос), а Мерлин отдает Нимуэ классический бит обратно (свой ответ).

В целях невмешательства в дела смертных Нимуэ теперь хочет использовать алгоритм Дойча–Йожи. Поскольку она живет в квантовом озере, Нимуэ может легко выделить кубит и отдать его Мерлину. К счастью для нас, Мерлин умеет общаться с кубитами, но нам по-прежнему нужно выяснить, что конкретно Мерлин будет делать с кубитами Нимуэ, чтобы действовать в соответствии со своей стратегией.

Проблема в том, что мы не можем передать кубиты в функцию f, которую используем для представления стратегии Мерлина: f принимает

и возвращает не кубиты, а классические биты. Для того чтобы Мерлин применял свою стратегию, которая направляла бы его в том, что он делает с кубитами Нимуэ, мы хотим превратить стратегию Мерлина *f* в своего рода квантовую программу, именуемую *оракулом*. К нашему удобству, Мерлин довольно хорошо играет роль оракула.

ПРИМЕЧАНИЕ Из трактовки персонажа Мерлина Теренсом Х. Уайтом мы знаем, что он проживает свою жизнь назад во времени. Мы представим это, обеспечив, что все, что делает Мерлин, является *унитарным*. Как мы видели в главе 2, одним из следствий этого представления является то, что применяемые Мерлином преобразования *обратимы*. В частности, Мерлин не может измерять кубиты Нимуэ, поскольку операция измерения не является обратимой. Эта привилегия принадлежит только Нимуэ.

В целях понимания того, что нам нужно сделать, чтобы просимулировать действия Мерлина как оракула, мы должны выяснить две вещи:

1. какое преобразование Мерлин должен применить к кубитам Нимуэ, основываясь на своей стратегии?
2. какие квантовые операции Мерлину потребуется применить для имплементирования этого преобразования?

Унитарные матрицы и таблицы истинности

На вопрос, что именно нам нужно сделать на шаге 1, можно ответить и по-другому, сказав, что нам нужно найти *унитарную матрицу*, которая представляет то, что делает Мерлин, аналогично тому, как мы использовали классические функции, такие как *f*, для представления того, что делал Мерлин, когда Нимуэ давала ему классические биты. Как мы видели в главе 2, унитарные матрицы относятся к квантовым вычислениям так же, как таблицы истинности к классическим: они говорят нам о том, каким является эффект квантовой операции для всех возможных входных данных. После того как мы найдем правильный унитарный оператор, на шаге 2 мы выясним последовательность подлежащих выполнению нами квантовых операций, которые будут описываться этим унитарным оператором.

8.3.1 Преобразования Мерлина

В целях завершения шага 1 нам нужно превратить такие функции, как *f*, в унитарные матрицы, поэтому давайте начнем с повторения того, чем может быть *f*. Возможные стратегии, которые Мерлин может использовать, представлены функциями `id`, `not`, `zero` и `one` (см. рис. 8.4).

Для двух сбалансированных функций `id` и `not` на рис. 8.4 мы можем легко ответить на вопрос 1. Квантовые программы для `id` и `not` могут быть имплементированы в виде поворотных операций, что позволяет легко превратить их в квантовые операции. Операция квантового NOT,

например, представляет собой поворот на 180° вокруг оси X, обмениваясь состояниями $|0\rangle$ и $|1\rangle$ друг с другом.

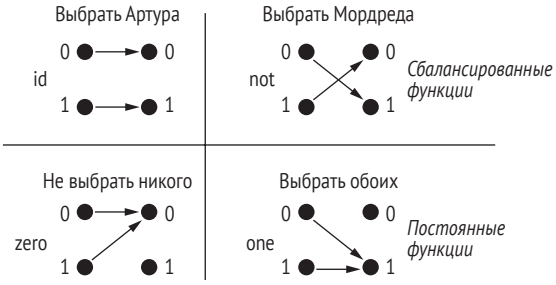
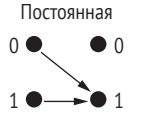


Рис. 8.4 Четыре разные функции из одного бита в один бит

ДЛЯ СПРАВКИ Вспомните из главы 3, что квантовая операция `X`, представленная унитарной матрицей $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, применяет поворот на 180° вокруг оси X. Эта операция имплементирует квантовое NOT: поскольку $X|0\rangle = |1\rangle$ и $X|1\rangle = |0\rangle$, мы можем написать его с помощью оператора ¬ (NOT) из главы 2, как $X|x\rangle = |\neg x\rangle$.

В то время как любой поворот можно откатить, повернув на ту же величину в противоположном направлении, мы сталкиваемся с еще большим количеством проблем с постоянными функциями `zero` и `one`. Ни `zero`, ни `one` невозможно имплементировать непосредственно в виде поворотов, поэтому нам предстоит еще немного поработать. Например, если f является `zero`, то выходы $f(0)$ и $f(1)$ равны 0. Если у нас есть только выход 0, то мы не сможем сказать, каким образом мы получили это значение на выходе: в результате передачи на вход в f 0 либо 1? (см. рис. 8.5).

Пример 1: по тому, как соотносятся входы и выходы, мы видим, что если мы знаем выход функции, то можем проследить его вплоть до того, каким был вход!



Пример 2: в случае с постоянными функциями мы видим, что хотя мы знаем значение на выходе, не ясно, каким было значение на входе. Им могло быть либо 0, либо 1, но мы не знаем...

Рис. 8.5 Почему мы не можем обратить постоянные функции zero или one? В принципе, если все входные значения соотносятся с одним-единственным выходным значением, то мы теряем информацию о том, с какого входного значения мы начали

ПРИМЕЧАНИЕ Как только мы применяем `zero` или `one`, мы теряем любую информацию о входе.

Поскольку и `one`, и `zero` необратимы, а валидные операции на кубитах обратимы, Мерлину нужен еще один способ представления функций, подобных *f*, в квантовых алгоритмах, таких как испытание Нимуэ. С другой стороны, если мы сможем представить стратегию Мерлина с помощью обратимой классической функции вместо *f*, будет гораздо проще написать квантовое представление его стратегии. Ниже приведена наша стратегия представления классических функций в виде квантовых оракулов.

1. Найти способ представить нашу необратимую классическую функцию с помощью обратимой классической функции.
2. Написать преобразование на квантовых состояниях, используя нашу обратимую классическую функцию.
3. Выяснить, какие мы можем выполнять квантовые операции, которые приводили бы к такому преобразованию.

Давайте воспользуемся проверенным и верным подходом – угадыванием и проверкой, сможем ли мы сконструировать действительно обратимую классическую функцию. Самый простой способ выяснить, что конкретно, 0 либо 1, мы получили на входе, состоит в том, чтобы это где-то зарегистрировать. Итак, давайте создадим новую функцию, которая возвращает два бита вместо одного.

В качестве нашей первой попытки давайте зарегистрируем и сохраним входные данные:

$$g(x) = (x, f(x)).$$

Например, если Мерлин использует стратегию `one` (т. е. он говорит Нимуэ «да» независимо от того, что она спрашивает), то $f(x) = 1$, и $g(x) = (x, f(x)) = (x, 1)$.

Это намного приближает нас к решению, так как теперь мы можем сказать, с какого входного значения, 0 либо 1, мы начали. Но мы не совсем у цели, так как *g* имеет два значения на выходе и одно значение на входе (см. рис. 8.6).

В целях использования *g* в качестве стратегии Мерлину пришлось бы вернуть Нимуэ больше кубитов, чем она ему передала, но она является хранителем как мечей, так и кубитов. Точнее говоря, в техническом плане обращение *g* вспять уничтожило бы информацию, так как оно потребовало бы два входа и вернуло бы один выход!

Пробуя еще раз, давайте определим новую классическую функцию *h*, которая на входе принимает два значения и на выходе возвращает тоже два значения, $h(x, y)$. Давайте еще раз рассмотрим пример, в котором мы описываем стратегию Мерлина с функцией $f(x) = 1$. Поскольку *g* привела нас почти до цели, мы выберем *h* такую, что $h(x, 0) = g(x)$. С нашей первой попытки мы увидели, что когда Мерлин использует стратегию $f(x) = 1$, то $g(x) = (x, 1)$, поэтому мы имеем, что $h(x, 0) = (x, 1)$. Теперь нам просто нужно определить, что происходит при передаче $y = 1$ в *h*. Если мы хо-

тим, чтобы h была обратимой, нам нужно, чтобы $h(x, 1)$ было назначено чему-то другому, чем $(x, 1)$. Это можно сделать, позволив $h(x, y) = (x, \neg y)$ так, чтобы $h(x, 1) = (x, 0) \neq (x, 1)$. Этот выбор особенно удобен, так как двухкратное применение h возвращает нам наши изначальные входные данные, $h(h(x, y)) = h(x, \neg y) = (x, \neg\neg y) = (x, y)$. Если это выглядит немного многословно, то взгляните на визуальное представление указанной аргументации на рис. 8.7.

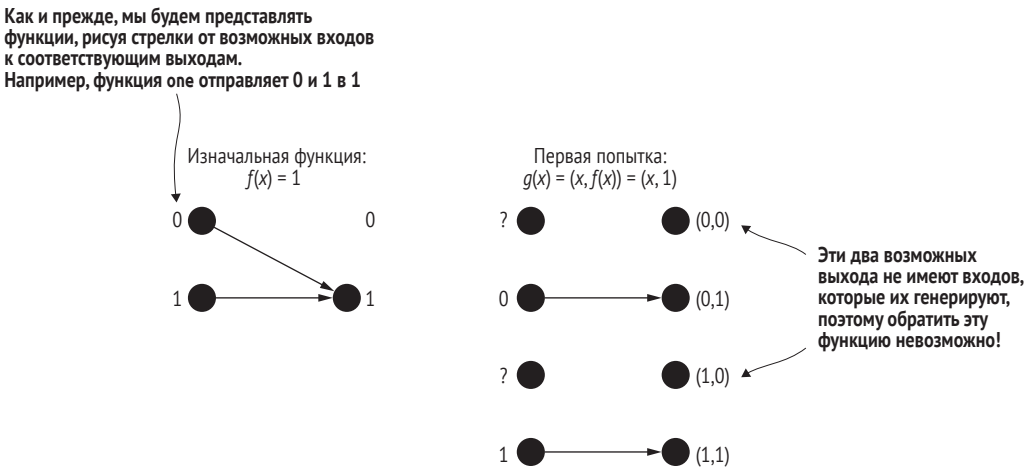


Рис. 8.6 Первая попытка: сохранение входа и выхода с помощью $g(x)$. Используя этот подход, некоторые выходные комбинации недостижимы ни с одного входа (например, ни один вход не производит выход (1, 0)). Отсюда обратить функцию невозможно, так как нет входа, соответствующего этим выходам

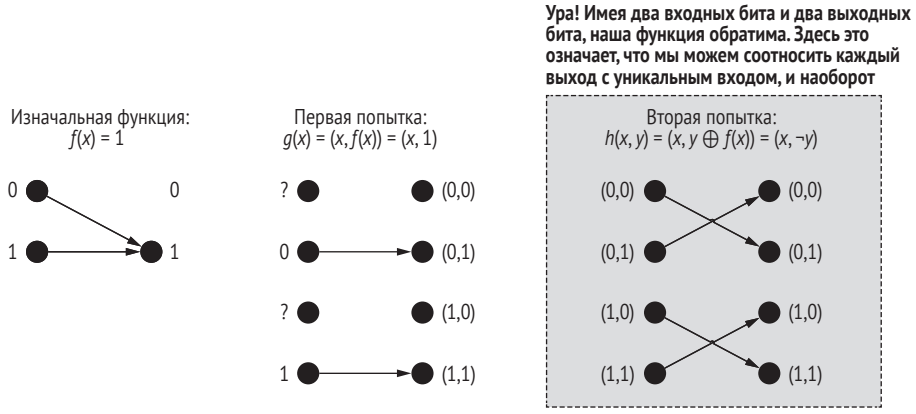


Рис. 8.7 Вторая попытка: функция $h(x, y)$ является обратимой и имеет одинаковое число входов и выходов

Теперь, когда мы знаем, как сделать *обратимую* классическую функцию из каждой стратегии, давайте закончим тем, что создадим кван-

товую программу из нашей обратимой функции. В случае функции one мы видели, что h переворачивает свое второе входное значение, $h(x, y) = (x, \neg y)$. Отсюда мы можем написать квантовую программу, которая делает то же самое, что и наша обратимая классическая функция, просто переворачивая второй из двух входных кубитов. Как мы видели в главе 4, мы можем сделать это с помощью инструкции X, так как $X|x\rangle = |\neg x\rangle$.

8.3.2 Обобщение наших результатов

В более общем случае мы можем сделать обратимую квантовую операцию точно так же, как мы сделали обратимую классическую функцию h, перевернув выходной бит, основываясь на выходном значении необратимой функции f. Мы можем определить унитарную матрицу (т. е. квантовый аналог таблицы истинности) U_f для f для каждого входного состояния точно в таком же ключе. На рис. 8.8 показано сравнение того, как можно выполнить это определение.

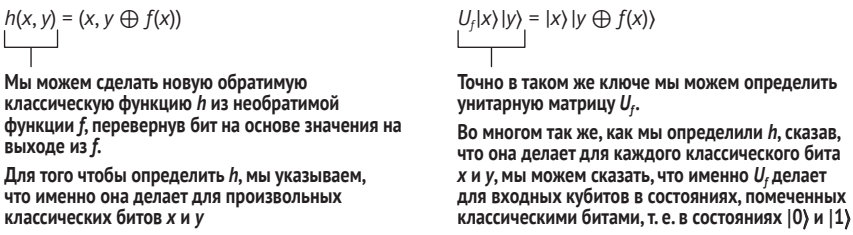


Рис. 8.8 Строительство обратимых классических функций и унитарных матриц из необратимых классических функций. Слева мы видим, что мы можем построить обратимую классическую функцию из необратимой, если будем отслеживать входные данные, которые мы передаем в необратимую функцию. Мы можем сделать то же самое для унитарной матрицы, описывающей квантовую операцию, имея два реестра кубитов: один реестр отслеживает вход, а другой содержит выход нашей необратимой функции

Определение U_f в таком ключе позволяет легко делать откат вызова f, так как двукратное применение U_f дает нам тождество $\mathbb{1}$ (т. е. унитарную матрицу для инструкции «ничего не делать»). Когда мы определяем унитарную матрицу таким способом, применяя функцию f условно к меткам для кубитовых состояний, мы называем эту новую операцию *оракулом*.

ОПРЕДЕЛЕНИЕ *Оракул* – это квантовая операция, представляемая унитарной матрицей U_f, которая преобразовывает свое входное состояние как

$$U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle.$$

Символ $\oplus$ воплощает оператор исключающего OR из обычной булевой логики.

Осталось лишь выяснить, какую последовательность инструкций нам нужно отправить для имплементирования каждой унитарной U_f. Мы встречали инструкции, необходимые для имплементирования оракула для функции one: инструкцию X для второго кубита. Теперь давайте посмотрим на то, как писать оракулы для других возможных функций f. Благодаря этому Мерлин будет знать, что ему следует делать, независимо от его стратегии.

Глубокое погружение: почему она зовется оракулом?

До этого мы встречали несколько разных примеров причудливых имен, которыми квантовые вычисления обязаны своей истории физики, таких как бра, кет и телепортация. Однако не только физики любят чуть-чуть повеселиться! Одна из отраслей теоретической информатики, именуемая *теорией сложности*, занимается разведывательным анализом того, что можно делать эффективно, хотя бы в принципе, при наличии разных типов вычислительных машин. Возможно, вы слышали, например, о проблеме «**P** против **NP**», классической головоломке в теории сложности, которая задается вопросом, так ли уж трудно решать задачи в классе **P**, как задачи в классе **NP**. Класс сложности **P** – это группа вопросов, ответы на которые можно получить с помощью алгоритма, занимающего полиномиальное время. Напротив, **NP** – это группа вопросов, потенциальный ответ на которые можно проверить за полиномиальное время, но мы не знаем, сможем ли найти ответ за полиномиальное время с чистого листа.

Многие другие задачи в теории сложности ставятся путем введения небольших игр или историй, в помощь исследователям, чтобы те могли помнить, какие определения и где использовать. Наша собственная маленькая история о Мерлине и Нимуэ является данью этой традиции. На самом деле одна из самых известных историй в квантовых вычислениях обозначается аббревиатурой **MA** от «Мерлин-Артур». Задачи в классе **MA**, как полагают, связаны с использованием истории, в которой Артур задает Мерлину, всемогущему, но ненадежному магу, серию вопросов. Задача выбора альтернативного решения типа «да/нет» возникает в **MA**, если всякий раз, когда дан ответ «да», существует доказательство, которое Мерлин может дать Артуру и которое Артур может легко проверить с помощью машины **P** и генератора случайных чисел.

Название оракула вписывается в этот тип повествования, поскольку любой класс сложности **A** может быть превращен в новый класс сложности $\mathbf{A}^{\mathbf{B}}$, позволяя машинам **A** решать задачу **B** за один шаг, как если бы они консультировались с оракулом. Бóльшая часть истории таких задач, как задача Дойча–Йожи, связана с попытками понять то, как квантовые вычисления влияют на вычислительную сложность, и вследствие этого в квантовых вычислениях были приняты многие соглашения об именах и бóльшая часть терминологии.

Подробнее о теории сложности и ее связи с квантовыми вычислениями, черными дырами, свободной волей и греческой философией читайте в книге Скотта Ааронсона «Квантовые вычисления со времен Демокрита» (*Quantum Computing Since Democritus*, Scott Aaronson, Cambridge University Press, 2013).

В общем случае задача отыскания последовательности инструкций, начиная с унитарной матрицы, является математически сложной и называется *унитарным синтезом*. С учетом этого в данном случае мы можем ее решить, подставив каждую стратегию Мерлина f в наше определение для U_f и идентифицировав инструкции, которые будут иметь такой эффект – мы можем выдвинуть догадку и ее проверить так же, как мы делали, чтобы превратить функцию `one` в оракула. Давайте попробуем этот подход для функции `zero`.

Упражнение 8.1: попробуйте написать оракула!

Какой была бы оракульная операция (U_f) для f, если бы f была функцией `zero`?

Решение: давайте разберемся с этим пошагово.

1. Из определения U_f мы знаем, что $U_f|xy\rangle = |x\rangle\ |y \oplus f(x)\rangle$.
2. Подставив `zero` в f, $f(x) = 0$, получаем, что $U_f|xy\rangle = |x\rangle|y \oplus 0\rangle$.
3. Мы можем использовать, что $y \oplus 0 = y$, и упростить еще больше, получив, что $U_f|xy\rangle = |x\rangle|y\rangle$.
4. В этой точке мы замечаем, что U_f ничего со своим входным состоянием не делает, поэтому мы можем ее имплементировать… ничего не делая.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

Функция f = `id` является чуть-чуть изощреннее, чем в случае функций `zero` и `one`, потому что невозможно упростить $y \oplus f(x)$ без того, чтобы зависеть от x. Как резюмировано в табл. 8.7, нам нужно $U_f|x\rangle\ |y\rangle = |x\rangle\ |y \oplus f(x)\rangle = |x\rangle\ |y \oplus x\rangle$. То есть нам нужно действие оракула на входное состояние ($|x\rangle\ |y\rangle$), чтобы оставить x нетронутым и заменить y на исключающее OR между x и y.

На это можно взглянуть и по-другому, а именно просто вспомнить, что $y \oplus 1 = \neg y$, поэтому при $x = 1$ нам нужно перевернуть y. Как раз так мы определили инструкцию контролируемого-NOT (CNOT) в главе 6, поэтому мы понимаем, что при f, равной `id` U_f, можно имплементировать путем применения CNOT.

В связи с этим нам остается определить оракула для f = `not`. Подобно тому, как оракул для `id` переворачивает выходной (целевой) кубит, когда входной (контрольный) кубит находится в состоянии $|1\rangle$, та же аргументация дает, что нам нужно, чтобы оракул для `not` перевернул второй кубит, когда входной кубит находится в состоянии $|0\rangle$. Самый простой для этого подход состоит в том, чтобы сначала перевернуть входной кубит

с помощью инструкции `X`, применить инструкцию `CNOT`, а затем откатить первый переворот еще одной `X`.

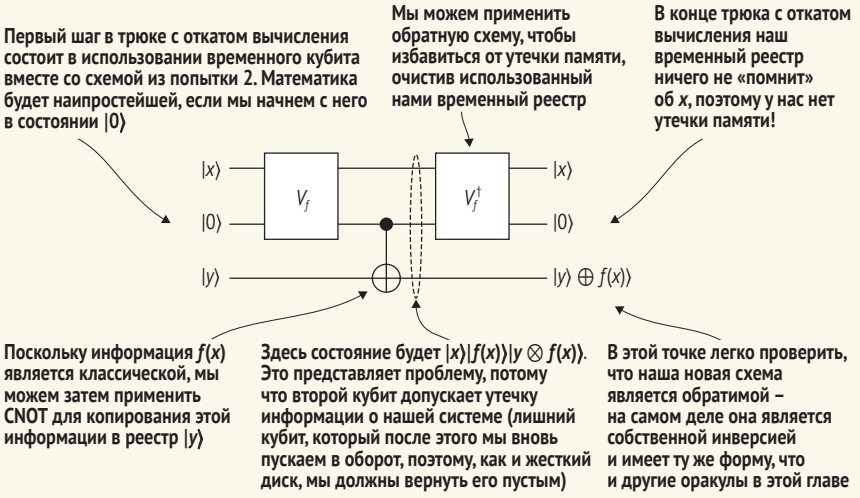
ДЛЯ СПРАВКИ Если вы хотите узнать больше о том, как строить других оракулов и использовать пакет QuTiP для доказательства того, что они делают то, что вы хотите, то рекомендуем ознакомиться с приложением D к книге.

Трюк с откатом вычисления: превращение функций в квантовых оракулов

В нынешнем виде может показаться, что для разработки U_f для каждой функции f требуется много работы. К счастью, есть хороший трюк, который позволяет нам строить оракула, начиная с несколько более простого требования.

Напомним, что ранее мы пытались определить обратимую версию f, возвращая $(x, f(x))$ на выходе, когда на входе задано $(x, 0)$. Аналогичным образом предположим, что дана квантовая операция V_f, которая правильно преобразовывает $|x\rangle|0\rangle$ в $|x\rangle|f(x)\rangle$. Мы всегда можем сделать оракула U_f, используя один дополнительный кубит и дважды вызывая V_f, используя технический прием под названием трюк с откатом вычисления (uncompute), как показано на следующем ниже рисунке.

Трюк с откатом вычисления: если добавить еще один вход $|0\rangle$, то мы можем сделать операцию, которая дает нам возможность сделать обратимой операцию из $|x\rangle|y\rangle$ в $|x\rangle|f(x)\rangle$! Это работает для любой функции $f(x)$



Использование трюка с откатом вычисления, чтобы превратить операцию, которая работает только тогда, когда мы добавляем лишний входной кубит $|0\rangle$, в операцию, которую можно использовать в качестве оракула

Хотя это не особо помогает в случае Дойча–Йожи, но все же показывает, что концепция оракула является очень общей, так как часто гораздо проще начать с операции в форме V_f.

В целях ревизии оракулов, которые мы научились определять, мы собрали всю работу этого раздела в приведенной ниже табл. 8.7.

Таблица 8.7 Выходные данные из оракула для каждой однобитовой функции *f*

Имя функции	Функция	Выход из оракула
`zero`	$f(x) = 0$	$\|x\rangle\|y \oplus 0\rangle = \|x\rangle\|y\rangle$
`one`	$f(x) = 1$	$\|x\rangle\|y \oplus 1\rangle = \|x\rangle\|\neg y\rangle$
`id`	$f(x) = x$	$\|x\rangle\|y \oplus x\rangle$
`not`	$f(x) = \neg x$	$\|x\rangle\|y \oplus \neg x\rangle$

ПРИМЕЧАНИЕ Строительство оракула работает и для многокубитовых функций. В качестве мысленного упражнения: если у нас есть функция $f(x_0, x_1) = x_0$ AND x_1, то как оракул U_f будет преобразовывать входное состояние $|x_0\, x_1\rangle$? В последующих главах мы научимся кодировать этого оракула.

Таким образом, мы применили оракульное представление для решения проблемы, связанной с тем, что функции, подобные `zero` и `one`, невозможно представить в виде поворотов. Разобравшись с этим, мы можем продолжить написание остальной части алгоритма, который Нимуэ использует, чтобы бросить вызов Мерлину.

Глубокое погружение: другие способы представления функций в виде оракулов

Это не единственный способ, которым мы могли бы определить U_f. Мерлин мог бы переворачивать знак x, вводимого владычицей Нимуэ, когда $f(x)$ является функцией `one`:

$$U_f|x\rangle = (-1)^{f(x)}|x\rangle.$$

Это представление оказывается полезнее в некоторых случаях, например в алгоритмах градиентного спуска. Указанные алгоритмы распространены в машинном обучении и минимизируют функции путем поиска по направлениям, в которых функция изменяется быстрее всего. Дополнительные сведения см. в разделе 4.10 книги Эндрю Траска «Грокаем глубокое обучение» (*Grokking Deep Learning*, Andrew Trask, Manning, 2019).

Подбор правильного подхода к представлению классической информации для конкретного приложения, такого как вызовы подпрограмм внутри квантового алгоритма, является частью искусства квантового программирования. Пока что мы будем использовать определение оракула, введенное ранее.

Имея на руках оракульное представление f, первые несколько шагов алгоритма Дойча–Йожи могут быть записаны в том же псевдокоде, который мы использовали для быстрой сортировки ранее.

1. Подготовить два кубита с метками `control` (контрольный) и `target` (целевой) в состоянии $|0\rangle \otimes |0\rangle$.
2. Применить операции к кубитам `control` и `target`, чтобы подготовить следующее состояние: $|+\rangle \otimes |-\rangle$.
3. Применить оракула U_f к входному состоянию $|+\rangle \otimes |-\rangle$. Напомним, что $U_f|x\rangle|y\rangle = |x\rangle\ |y \oplus f(x)\rangle$.
4. Измерить кубит `control` в X-базисе. Если мы наблюдаем 0, то функция является постоянной; в противном случае функция является сбалансированной.

ДЛЯ СПРАВКИ Измерение кубита в X-базисе всегда возвращает 0 либо 1, точно так же, как если бы мы измеряли в Z-базисе. Вспомните из главы 3, что если состояние кубита равно $|+\rangle$, то мы всегда получаем 0 при измерении в X-базисе, тогда как мы всегда получаем 1, если кубит находится в $|-\rangle$.

Эти шаги показаны на рис. 8.9. В конце главы мы узнаем причину, *почему* этот алгоритм работает, а пока что давайте приступим и начнем его программировать. Для этого мы будем использовать язык Q#, предоставляемый Комплектом инструментов для квантовой разработки, поскольку он дает возможность легко видеть структуру квантового алгоритма, глядя на его исходный код.

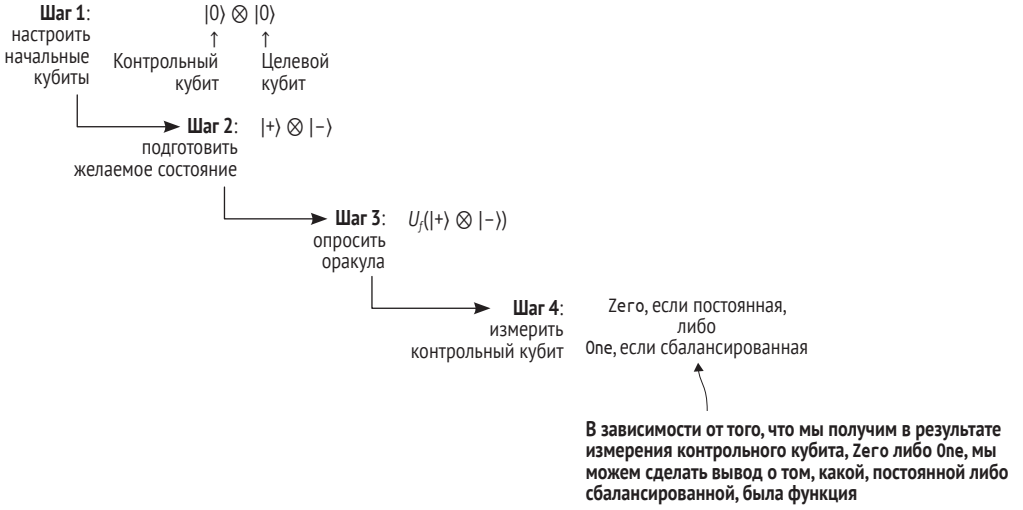


Рис. 8.9 Шаги алгоритма Дойча–Йожи. Мы начинаем с подготовки состояния $|+-\rangle$, затем опрашиваем оракула (т. е. задаем вопрос Мерлину), а потом измеряем контрольный кубит, чтобы узнать, какую функцию, постоянную либо сбалансированную, оракул представляет

8.4 Симулирование алгоритма Дойча–Йожи на Q#

В главе 7 мы попытались передавать операции в качестве аргументов в программах Q#. Мы можем использовать тот же подход, передавая операции на вход в оракулы, чтобы помочь предсказать исход испытания Мерлина. Для этого давайте вспомним, что в этой задаче мы можем рассматривать четыре возможные функции, каждая из которых представляет возможную стратегию, которую Мерлин мог бы использовать (см. табл. 8.8).

Таблица 8.8 Представление однобитовых функций в виде двухкубитовых оракулов

Имя функции	Функция	Результат работы оракула	Операция Q#
`zero`	$f(x) = 0$	$\|x\rangle\|y \oplus 0\rangle = \|x\rangle\|y\rangle$	`NoOp(control, target);`
`one`	$f(x) = 1$	$\|x\rangle\|y \oplus 1\rangle = \|x\rangle\|\neg y\rangle$	`X(target);`
`id`	$f(x) = x$	$\|x\rangle\|y \oplus x\rangle$	`CNOT(control, target)`
`not`	$f(x) = \neg x$	$\|x\rangle\|y \oplus \neg x\rangle$	`X(control); CNOT(control, target); X(control);`

Если мы представим каждую функцию $f(x)$ с помощью оракула (квантовой операции), который отображает $|x\rangle|y\rangle$ в $|x\rangle|y \oplus f(x)\rangle$, то мы сможем идентифицировать каждую из функций `zero`, `one`, `id` и `not` из рис. 8.3.

Каждый из четырех приведенных выше оракулов транслируется непосредственно в Q#:

```
namespace DeutschJozsa {
    open Microsoft.Quantum.Intrinsic;

    operation ApplyZeroOracle(control : Qubit, target : Qubit) : Unit {
    }

    operation ApplyOneOracle(control : Qubit, target : Qubit) : Unit {
        X(target);
    }

    operation ApplyIdOracle(control : Qubit, target : Qubit) : Unit {
        CNOT(control, target);
    }

    operation ApplyNotOracle(control : Qubit, target : Qubit) : Unit {
        X(control);
        CNOT(control, target);
        X(control);
    }
}
```

Разве мы не можем просто взглянуть на исходный код?

В Oracles.qs мы написали исходный код для каждого из четырех однокубитовых оракулов `ApplyZeroOracle`, `ApplyOneOracle`, `ApplyIdOracle` и `ApplyNotOracle`. Глядя на этот исходный код, мы можем сказать, каким, постоянным либо сбалансированным, является каждый из них без необходимости его вызывать, так почему же мы должны беспокоиться об алгоритме Дойча–Йожи? С точки зрения Нимуэ, у нее не обязательно есть исходный код, который Мерлин использует для применения операций к своим кубитам. Но даже если она его имеет, пути Мерлина неисповедимы, поэтому она не сможет легко предсказывать, что делает Мерлин, даже учитывая исходный код, который он использует.

Хотя с практической точки зрения трудно столь сильно затуманить двухкубитового оракула, алгоритм Дойча–Йожи демонстрирует технический прием, который полезен в более общем случае. Например, у нас может быть доступ к исходному коду операции, но задача извлечения ответа на вопрос об этой операции является математически или вычислительно сложной. Все криптографические хеш-функции обладают этим свойством по своему дизайну, независимо от того, используются ли они для обеспечения правильного скачивания файла, для проверки правильности подписи приложения разработчиком либо в рамках выращивания блокчейна посредством майнинга коллизий.

В главе 10 мы рассмотрим пример, в котором используются технические приемы, подобные тем, которые были разработаны в алгоритме Дойча–Йожи, чтобы задавать вопросы о таких функциях быстрее.

Имплементировав этих оракулов на Q#, мы можем написать весь алгоритм Дойча–Йожи (а также стратегию Нимуэ для Воспитателя короля)! См. рис. 8.10, чтобы освежить свои знания о шагах алгоритма Дойча–Йожи.

Рис. 8.10 Шаги алгоритма Дойча–Йожи. Мы начинаем с подготовки состояния |+−⟩, затем опрашиваем оракула (т. е. задаем вопрос Мерлину), а потом измеряем контрольный кубит

Листинг 8.2 Algorithm.qs: операция Q# для выполнения алгоритма Дойча–Йожи

```
operation CheckIfOracleIsBalanced(oracle : ((Qubit, Qubit) => Unit))
: Bool {
    use control = Qubit();
    use target = Qubit();                    ❶

    H(control);                              ❷
    X(target);
    H(target);

    oracle(control, target);                 ❸

    H(target);                               ❹
    X(target);

    return MResetX(control) == One;          ❺
}
```

❶ Просит целевую машину дать нам два кубита, контрольный и целевой, каждый из которых начинается в состоянии |0⟩.
❷ Подготавливает входное состояние |+−⟩ = (|00⟩ – |01⟩ + |10⟩ – |11⟩)/2 на контрольном и целевом кубитах, как показано на шаге 2 рис. 8.10.
❸ Вызывает оракула, переданного во входном аргументе. Обратите внимание, что оракул вызывается только один раз!
❹ Мы знаем, что целевой кубит все еще находится в |−⟩, поэтому мы можем откатить последовательность операций X(target); H(target); для сброса целевого кубита.
❺ Измеряет, в каком состоянии, |+⟩ либо |−⟩, находится контрольный кубит, что соответствует результату Zero либо One в *X*-базисе.

Как и предоставляемая стандартными библиотеками Q# операция `MResetZ`, операция `MResetX` выполняет желаемое *X*-базисное измерение и сбрасывает измеренный кубит в состояние |0⟩. Теперь мы хотим убедиться, что наша имплементация работает как надо, поэтому давайте ее протестируем!

Результаты измерения на Q#

Теперь мы увидели операции `MResetX` и `MResetZ` на языке Q#, которые измеряют и сбрасывают кубит соответственно в *X*- и *Z*-базисе. Обе эти операции возвращают результирующее значение, `Result`, которое поначалу кажется немного туманным. В конце концов, *X*-базисное измерение говорит нам о том, в каком состоянии, |+⟩ либо |−⟩, мы находились, так почему же в Q# используются метки `Zero` и `One`?

Условные обозначения, используемые для результатов *X*- и *Z*-базисного измерений на языке Q#

Значение `Result`	*X*-базис	*Z*-базис
`Zero`	⟨+|	⟨0|
`One`	⟨−|	⟨1|

Мы подробнее поговорим об этом позже, но короткий ответ таков: значение типа `Result` сообщает нам число фаз (–1), применяемых к состоянию разными инструкциями. Например, $Z|1\rangle = -|1\rangle = (-1)^1\ |1\rangle$, тогда как $X|-\rangle = (-1)^1\ |-\rangle$. Поскольку в обоих случаях мы возводим (–1) в степень 1, $|1\rangle$ и $|-\rangle$ назначаются результату `One` при измерении соответственно в Z- и X-базисах. Аналогичным образом, поскольку $Z|0\rangle = (-1)^0|0\rangle$, мы назначаем $|0\rangle$ результату `Zero` при Z-измерении.

Ранее мы сказали, что Нимуэ хотела бы знать как можно меньше о делах человеческих. Поэтому она просит Мерлина сделать с ее кубитами что-то только один раз, когда мы вызываем `oracle(control, target)`. Нимуэ получает только один классический бит информации из вызова `MResetX`, которого ей недостаточно, чтобы определить разницу между стратегией `id` (Мерлин выбирает Артура для наставничества на титул короля) и стратегией `not` (Мерлин выбирает Мордреда для наставничества на титул короля).

В целях обеспечения того, чтобы она все же могла узнать, что ее на самом деле волнует – является ли стратегия Мерлина постоянной либо сбалансированной, – мы можем применить функцию `Fact`, поставляемую со стандартными библиотеками Q#, чтобы проверить, что наша имплементация работает. Функция `Fact` принимает две булевы переменные в качестве первых двух аргументов, проверяет их на равенство и, если они не равны, выдает сообщение.

ДЛЯ СПРАВКИ Позже мы увидим, как использовать эти инструкции подтверждения истинности при написании модульных тестов для квантовых библиотек.

Прежде всего мы передаем операцию `ApplyZeroOracle`, которую мы написали ранее, в качестве оракула для функции `Zero`. Поскольку функция `Zero` не является сбалансированной, мы ожидаем, что вызов `CheckIfOracleIsBalanced(ApplyZeroOracle)` на выходе вернет `false`; это ожидание можно проверить с помощью функции `Fact`.

Листинг 8.3 Algorithm.qs: тестирование операции Q# в алгоритме Дойча–Йожи

```
operation RunDeutschJozsaAlgorithm() : Unit {
    Fact(not CheckIfOracleIsBalanced(ApplyZeroOracle),
        "Отказ теста для оракула zero.");                    ❶
    Fact(not CheckIfOracleIsBalanced(ApplyOneOracle),
        " Отказ теста для оракула one.");                    ❷
    Fact(CheckIfOracleIsBalanced(ApplyIdOracle),
        " Отказ теста для оракула id.");
    Fact(CheckIfOracleIsBalanced(ApplyNotOracle),
        " Отказ теста для оракула not.");

    Message("Все тесты прошли успешно!");                    ❸
}
```

❶ Выполняет алгоритм Дойча–Йожи для случая, когда Мерлин использует стратегию Zero.
❷ Делает в точности то же самое для стратегии One, на этот раз вместо этого вызывая CheckIfOracleIsBalanced(ApplyOneOracle).
❸ Если все четыре подтверждения истинности прошли, то мы можем быть уверены в том, что наша программа для алгоритма Дойча–Йожи работает независимо от того, какую стратегию использует Мерлин.

Если мы выполним этот фрагмент кода волшебной командой `%simulate`, то сможем подтвердить, что с помощью алгоритма Дойча–Йожи Нимуэ сможет узнать именно то, что она хочет узнать о стратегии Мерлина:

```
In [ ]: %simulate RunDeutschJozsaAlgorithm
Все тесты прошли успешно!
```

8.5 Размышления о квантово-алгоритмических техниках

Фу – мы проделали здесь парочку довольно серьезных шагов:

- мы применили классические обратимые функции для симулирования стратегии Мерлина так, чтобы мы могли написать ее в виде квантового оракула;
- мы использовали Q# и Комплект инструментов для квантовой разработки в целях имплементирования алгоритма Дойча–Йожи и проверки того, что мы можем узнать стратегию Мерлина с помощью одного оракульного вызова.

В этом месте полезно поразмышлять о том, что мы узнали, искупавшись в квантовом озере Нимуэ, поскольку технические приемы, которые мы использовали в этой главе, будут полезны на протяжении всей остальной части книги.

8.5.1 Ботинки и носки: применение и откат квантовых операций

Первый шаблон, над которым полезно поразмышлять, – это тот, который мы, возможно, заметили в Algorithm.qs. Давайте еще раз посмотрим на порядок, в котором операции применяются к целевому кубиту.

Листинг 8.4 Инструкции из алгоритма Дойча–Йожи для целевого объекта

```
// ...
X(target);
H(target);
oracle(control, target);
H(target);
X(target);
// ...
```

На эту последовательность можно вглянуть как на то, что инструкции `X(target); H(target);` подготавливают `target` в состоянии $|-\rangle$, тогда как инструкции `H(target); X(target);` делают «откат подготовки» $|-\rangle$, возвращая `target` обратно в состояние $|0\rangle$. Мы должны обратить порядок назад в силу того, что часто называется *принципом ботинок и носков*. Если мы хотим надеть ботинки и носки, то мы будем иметь более приемлемые результаты, если сначала наденем носки; но если мы хотим их снять, то нам нужно сначала снять ботинки. Иллюстрация этой процедуры приведена на рис. 8.11.

Рис. 8.11 Мы не можем снять носки раньше, чем ботинки

Язык Q# упрощает выполнение преобразований нашего кода в стиле ботинок и носков с помощью функционального средства под названием *функторы*. Функторы позволяют нам легко описывать новые варианты операции, которую мы уже определили. Давайте сразу перейдем к примеру и представим новую операцию `PrepareTargetQubit`, которая инкапсулирует последовательность `X(target); H(target);`.

Листинг 8.5 Подготовка состояния из листинга 8.4

```
operation PrepareTargetQubit(target : Qubit)
: Unit is Adj {                    ❶
    X(target);
    H(target);
}
```

❶ Написав «is Adj» в рамках сигнатуры, мы сообщаем компилятору Q#, чтобы он автоматически вычислял обратную операцию – т. е. адъюнкт – этой операции.

Затем мы можем вызвать сгенерированную компилятором обратную операцию, используя `Adjoint`, один из двух функторов, предлагаемых языком Q# (другой мы увидим в главе 9).

Листинг 8.6 Использование ключевого слова Adjoint для применения инструкций

```
PrepareTargetQubit(target);
oracle(control, target);
Adjoint PrepareTargetQubit(target);          ❶
```

❶ Adjoint PrepareTargetQubit применяет функтор Adjoint к PrepareTargetQubit, возвращая операцию, которая «откатывает» PrepareTargetQubit. Следуя мышлению в стиле ботинок и носков, эта новая операция работает, сначала вызывая Adjoint H(target);, а затем Adjoint X(target);.

Самоадъюнктные операции

В этом случае `X` и `Adjoint X` являются одной и той же операцией, так как переворачивание бита, а затем переворачивание его снова всегда возвращает нас к тому, с чего мы начали. Говоря по-другому, `X` откатывает `X`. Аналогичным образом `Adjoint H` совпадает с `H`, поэтому приведенный выше фрагмент дает нам последовательность `H(target); X(target);`. Мы говорим,что инструкции `X` и `H` являются *самоадъюнктными*.

Однако не все операции являются их собственными адъюнктами! Например, `Adjoint Rz(theta, _)` совпадает с операцией `Rz(-theta, _)`.

В более практическом плане функтор `Adjoint` на операции U совпадает с операцией, которая обращает назад эффекты, или делает откат эффектов, U. Название «адъюнкт» относится к конъюгатной транспозиции U^+ унитарной матрицы U. На словах U^+ называется *адъюнктом U*. Ключевое слово `Adjoint` в Q# гарантирует, что если операция U описывается унитарной U, то при условии существования адъюнкта U, т. е. `Adjoint U`, она описывается с помощью U^+.

Указанный шаблон выполнения инструкции используется настолько часто, что стандартные библиотеки Q# предлагают операцию `ApplyWith` для выражения этого шаблона выполнения и затем откатывания операции.

ПРИМЕЧАНИЕ Операция `ApplyWith` предоставляется пространством имен `Microsoft.Quantum.Canon` в стандартной библиотеке Q#. Как и стандартная библиотека на других языках, стандартная библиотека Q# предоставляет массу базовых инструментов, необходимых для написания программ на Q#. По мере прочтения остальной части книги вы увидите целый ряд способов, которыми стандартная библиотека Q# будет облегчать вашу жизнь как разработчика квантовых систем.

Используя операцию `ApplyWith` и частичное применение, мы можем переписать операцию `CheckIfOracleIsBalanced` компактно.

Листинг 8.7 ApplyWith и частичное применение помогают упорядочивать в стиле ботинок и носков

```
H(control);
ApplyWith(PrepareTargetQubit, oracle(control, _), target);
set result = MResetX(control);
```

Операция `ApplyWith` в этом примере автоматически применяет адъюнкт операции `PrepareTargetQubit` после выполнения `oracle(control, _)`. Обратите внимание, что `_` используется для частичного применения оракула к контрольному кубиту.

Давайте шаг за шагом расширим листинг 8.7, чтобы увидеть, как он работает. Вызов операции `ApplyWith` применяет свой первый аргумент, затем применяет свой второй аргумент, а потом применяет адъюнкт своего первого аргумента к кубиту, переданному в последнем аргументе.

Листинг 8.8 Расширение операции ApplyWith из листинга 8.7

```
H(control);
PrepareTargetQubit(target);
(oracle(control, _))(target);
Adjoint PrepareTargetQubit(target);
set result = MResetX(control);
```

Частичное применение в третьей строке кода затем может быть заменено путем подстановки `target` вместо `_`.

Листинг 8.9 Урегулирование частичного применения из листинга 8.8

```
H(control);
PrepareTargetQubit(target);
oracle(control, target);
Adjoint PrepareTargetQubit(target);
set result = MResetX(control);
```

Использование таких операций, как `ApplyWith`, помогает применять общие шаблоны в квантовом программировании многократно, в особенности для того, чтобы убедиться, что мы не забыли взять адъюнкт, `Adjoint`, в большой квантовой программе.

Q# также предоставляет еще один способ представления шаблона «ботинки и носки» с использованием блоков инструкций вместо пересылки операций туда-сюда. Например, мы могли бы написать листинг 8.7, используя вместо этого ключевые слова `within` и `apply`, как показано ниже.

Листинг 8.10 Использование ключевых слов within и apply для упорядочения в стиле ботинок и носков

```
H(control);
within {
    PrepareTargetQubit(target);
} apply {
```

```
    oracle(control, target);
}
set result = MResetX(control);
```

Обе формы бывают полезны в разных контекстах, поэтому не стесняйтесь использовать то, что подходит для вас лучше всего!

8.5.2 Использование инструкций Адамара для переворачивания управления и цели

Мы можем применить варианты мышления в стиле ботинок и носков из предыдущего раздела, чтобы изменить то, какие кубиты играют роль контрольных и целевых в таких инструкциях, как CNOT. В целях понимания принципа работы важно иметь в виду, что квантовые инструкции преобразовывают все состояния реестров, на которые они действуют. В таких случаях, как алгоритм Дойча–Йожи, на контрольный кубит можно повлиять, применив вентили к контрольному и целевому кубитам вместе, а не только к целевому кубиту. Это пример более общего шаблона: контрольный и целевой кубиты операции CNOT меняются ролями, когда мы применяем инструкцию CNOT в X-базисе вместо Z-(вычислительного) базиса.

В целях ознакомления давайте рассмотрим унитарный оператор (квантовый аналог классических таблиц истинности) для ответа на вопрос, что произойдет, если мы применим инструкции H для преобразования в X-базис, применим инструкцию CNOT, а затем применим еще инструкции H, чтобы вернуться к Z-базису.

Листинг 8.11 Проверка того, что H переворачивает управление и цель операции CNOT

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> H = hadamard_transform()
>>> HH = qt.tensor(H, H)
>>> HH                                    ❶
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
➥ isherm = True
Qobj data =                               ❷
[[ 0.5  0.5  0.5  0.5]
 [ 0.5 -0.5  0.5 -0.5]
 [ 0.5  0.5 -0.5 -0.5]
 [ 0.5 -0.5 -0.5  0.5]]
>>> HH * qt.cnot(2, 0, 1) * HH            ❸
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
➥ isherm = True
Qobj data =                               ❹
[[1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]]
```

```
>>> qt.cnot(2, 1, 0)                    ❺
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
➥ isherm = True
Qobj data =
[[1. 0. 0. 0.]
 [0. 0. 0. 1.]                        ❻
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]]
```

❶ **Полезно определить укороченную нотацию для унитарного оператора $H \otimes H$, который симулирует последовательность инструкций H(control); H(target);.**

❷ **Глядя на унитарный оператор $H \otimes H$, мы видим, что $|00\rangle$ преоразовывается в ($|00\rangle + |01\rangle + |10\rangle + |11\rangle$) / 2, универсальную суперпозицию по всем четырем вычислительно-базисным состояниям.**

❸ **Дает унитарный оператор, представляющий H(control); H(target); CNOT(control, target); H(control); H(target);. Эта последовательность инструкций может рассматриваться как применение CNOT в *X*-базисе вместо *Z*-базиса.**

❹ **Унитарный оператор для этой последовательности немного похож на CNOT, но в которой несколько строк перевернуто. Что же случилось?**

❺ **В попытке выяснить, что конкретно применение инструкций H к каждому кубиту сделало с инструкцией CNOT, давайте рассмотрим унитарный оператор для CNOT(target, control).**

❻ **Реверсирование роли контрольного и целевого кубитов в вызове инструкции CNOT дает нам точно такой же унитарный оператор, как и использование инструкций H для применения инструкции CNOT в *X*-базисе.**

На рис. 8.12 приведено визуальное представление исходного кода на Python, который мы только что выполнили.

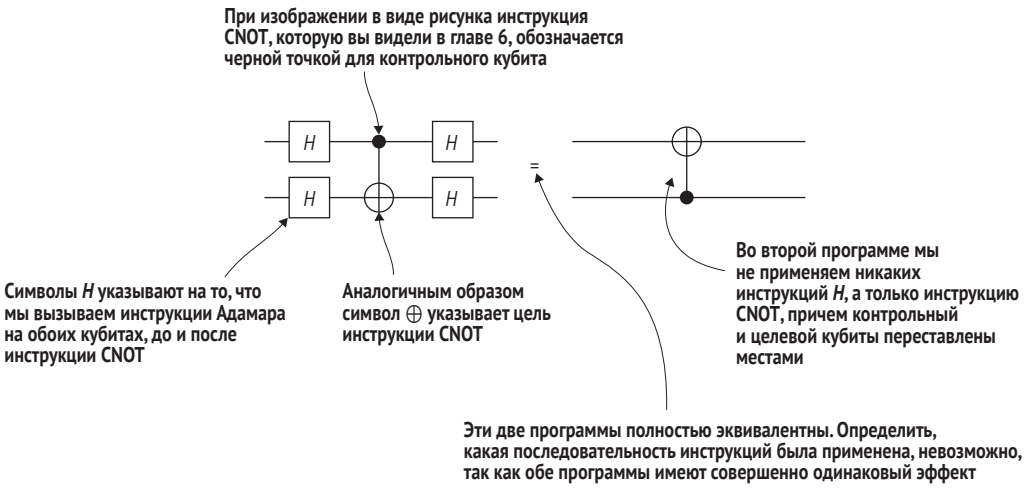


Рис. 8.12 Обмен ролями между контрольным и целевым кубитами для инструкции CNOT с использованием инструкций Адамара. Применяя Адамары к каждому кубиту, как до, так и после CNOT, мы можем переворачивать роли контрольного и целевого кубитов

Из предыдущего вычисления мы можем заключить, что `CNOT(target, control)` делает в точности то же самое, что и `H(control); H(target);`

`CNOT(control, target); H(control); H(target);`. Подобно тому как `H` меняет роль X- и Z-базисов, инструкции `H` могут переворачивать между использованием кубита в качестве контрольного либо целевого.

8.6 Фазовая отдача: ключ к нашему успеху

Помятуя об этих технических приемах, мы теперь готовы заняться разведкой того, что заставляет алгоритм Дойча–Йожи делать свое дело: технический прием квантового программирования под названием *фазовая отдача* (phase kickback)[1]. Указанный прием позволяет нам написать операцию `CheckIfOracleIsBalanced`, чтобы та работала для нескольких разных оракулов, раскрывая только один бит, который мы хотим знать (о том, действовал ли Мерлин как хороший наставник или нет).

С целью ознакомления с использованием фазовой отдачи в алгоритме Дойча–Йожи для работы в целом давайте вернемся к нашим трем образам мышления и применим математику, чтобы предсказать, что произойдет при вызове любого оракула. Напомним, что мы определили оракула U_f, который мы построили из каждой классической функции f такой, что для всех классических битов x и y $U_f|xy\rangle = |x\rangle|f(x) \oplus y\rangle$.

> **ДЛЯ СПРАВКИ** Здесь мы используем x и y для представления классических битов, которые помечают двухкубитовые состояния. Это еще один пример использования вычислительного базиса для рассуждений о поведении квантовых программ.

Давайте начнем так же, как мы делали в наших программах на основе пакета QuTiP, с разворачивания входного состояния $|+-\rangle = |+\rangle \otimes |-\rangle$ в вычислительном базисе. Начиная с разворачивания состояния контрольного кубита, мы имеем, что $|+-\rangle = |+\rangle \otimes |-\rangle = (|0\rangle + |1\rangle)/\sqrt{2} \otimes |-\rangle = (|0-\rangle + |1-\rangle)/\sqrt{2}$. Как и прежде, мы можем проверить нашу математику с помощью пакета QuTiP.

Листинг 8.12 Использование пакета QuTiP для проверки $(|0-\rangle + |1-\rangle)/\sqrt{2} = |+-\rangle$

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> from numpy import sqrt
```

[1] Для справки: фазовая отдача (phase kickback) широко используется для решения черно-ящичных задач в квантовых вычислениях. Цель таких задач состоит в том, чтобы выяснять некоторую информацию о неизвестной функции f, применяя f к любому валидному значению на входе и получая результат на выходе без необходимости знать механизм работы внутри ящика, кодируя выходное значение $f(\psi)$ в фазе квантового состояния. Такое объяснение помогает понять, почему этот термин переводится именно как отдача. См. https://www.quora.com/What-is-phase-kickback-and-how-does-it-occur. – *Прим. перев.*

```
>>> H = hadamard_transform()
>>> ket_0 = qt.basis(2, 0)                    ❶
>>> ket_1 = qt.basis(2, 1)
>>> ket_plus = H * ket_0
>>> ket_minus = H * ket_1
>>> qt.tensor(ket_plus, ket_minus)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]
 [-0.5]]
>>> (
...      qt.tensor(ket_0, ket_minus) +
...      qt.tensor(ket_1, ket_minus)
... ) / sqrt(2)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]                                       ❷
 [-0.5]
 [ 0.5]
 [-0.5]]
```

❶ **Начинается с полезной укороченной нотации.**
❷ **Оба вектора одинаковы, говоря нам о том, что $(|0-\rangle + |1-\rangle)/\sqrt{2}$ является еще одним способом написания $|+-\rangle$.**

Далее, как мы видели в главе 2, мы можем применить линейность, чтобы предсказать, как U_f преобразовывает это входное состояние.

Еще раз о матрице

Ранее в этом разделе мы неявно применили линейность, когда использовали матрицы для симулирования работы алгоритма Дойча–Йожи. Как описано в главе 2, матрицы – это один из способов написания линейных функций.

Поскольку U_f является унитарной матрицей, мы знаем, что для *любого* состояния $|\psi\rangle$ и $|\varphi\rangle$ и для *любых* чисел α и β $U_f(\alpha|\psi\rangle + \beta|\varphi\rangle) = \alpha U_f|\psi\rangle + \beta U_f|\varphi\rangle]$. Используя это свойство с вычислительным базисом, мы имеем, что таким же образом $|+-\rangle$ и $(|0-\rangle + |1-\rangle)/\sqrt{2}$ являются одним и тем же состоянием, $U_f|+-\rangle$ и $U_f(|0-\rangle + |1-\rangle)/\sqrt{2}$ тоже являются одним и тем же состоянием.

ДЛЯ СПРАВКИ Использование нашей укороченной нотации для многокубитовых состояний $|+-\rangle = |+\rangle \otimes |-\rangle$, $|0-\rangle = |0\rangle \otimes |-\rangle$ и $|1-\rangle = |1\rangle \otimes |-\rangle$.

На рис. 8.13 показано визуальное изображение линейности.

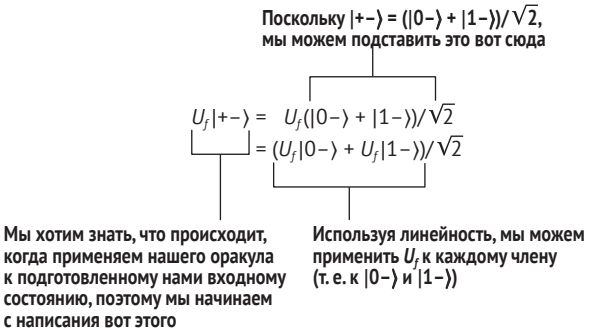


Рис. 8.13 Применение линейности для понимания принципа преобразования нашим оракулом входного состояния

Написав в таком виде, не сразу понятно, какое преимущество мы получили, применив U_f к $|0-\rangle$ и $|1-\rangle$. Давайте посмотрим, как оракульная операция применяется к первому члену, вынеся за скобки контрольный (первый) кубит, чтобы рассмотреть эффект на целевой кубит.

При этом мы снова будем использовать линейность, чтобы понять, как работает U_f, передавая нашему оракулу по одному состоянию за раз. Как мы узнали из главы 2, линейность является очень мощным инструментом, который позволяет нам разбивать даже довольно сложные квантовые алгоритмы на части, которые становится легче понимать и анализировать. В данном случае мы можем понять, как U_f действует на $|0-\rangle$, используя линейность (т. е. разбив $|0-\rangle$ на суперпозицию между $|00\rangle$ и $|01\rangle$).

Мы начинаем с использования, что $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$.

$$U_f|0-\rangle = U_f(|00\rangle - |01\rangle)/\sqrt{2}$$

Мы можем применить U_f к каждому члену в силу линейности.

$$= (U_f|00\rangle - U_f|01\rangle)/\sqrt{2}$$

Определение оракула говорит нам о том, что U_f преобразовывает состояния в вычислительном базисе.

$$= (|0(0 \oplus f(0))\rangle - |0(1 \oplus f(0))\rangle)/\sqrt{2}$$

Выполнение XOR чего-то с 0 ничего не делает.

$$= (|0\, f(0)\rangle - |0(1 \oplus f(0))\rangle)/\sqrt{2}$$

Выполнение XOR чего-то с 1 является тем же самым, что и NOT.

$$= (|0\, f(0)\rangle - |0(\neg f(0))\rangle)/\sqrt{2}$$

Поскольку контрольный кубит находится в обоих членах, мы можем его вынести за скобки.

$$= |0\rangle \oplus (|f(0)\rangle - |\neg f(0)\rangle)/\sqrt{2}$$

Например, если мы рассматриваем функцию `zero`, то $f(0) = 0$. Отсюда $|f(0)\rangle = |0\rangle$ и $|f(0)\rangle = |1\rangle$, поэтому $U_f|0-\rangle = |0-\rangle$.

С другой стороны, если $f(0) = 1$, то $U_f|0-\rangle = |0\rangle \otimes (|1\rangle - |0\rangle)/2 = -|0-\rangle$. То есть U_f переворачивает знак $|0-\rangle$.

ДЛЯ СПРАВКИ $(|1\rangle - |0\rangle)/\sqrt{2}$ также можно записать как $-|-\rangle$ либо как $X|-\rangle$.

Обратите внимание, что U_f либо поворачивает целевой кубит на X, либо нет, в зависимости от значения $f(0)$:

$$U_f|0-\rangle = (-1)^{f(0)}|0-\rangle/\sqrt{2}$$

Мы можем использовать тот же технический прием, что и раньше, чтобы понять, что делает U_f, если вместо этого контрольный кубит находится в состоянии $|1\rangle$. Поступая в таком ключе, мы получаем фазу $(-1)^{f(1)}$ вместо $(-1)^{f(0)}$, так что $U_f|1-\rangle = (-1)^{f(1)}|1-\rangle$.

Используя линейность снова, чтобы скомбинировать члены для двух состояний контрольного кубита, теперь мы знаем, как U_f преобразовывает состояние обоих кубитов, когда контрольный кубит находится в $|+\rangle$:

$$U_f|+-\rangle = \frac{1}{\sqrt{2}}((-1)^{f(0)}|0-\rangle + (-1)^{f(1)}|1-\rangle).$$

На последнем шаге мы должны отметить, что, как мы видели в главе 4, мы не можем наблюдать *глобальные фазы*. Следовательно, мы можем вынести за скобки $(-1)^{f(0)}$, чтобы выразить выходное состояние в членах $f(0) \oplus f(1)$, вопрос, который нас интересовал с самого начала, как показано на рис. 8.14.

Если $f(0) \oplus f(1) = 0$ (постоянная), то выходное состояние равно $|+-\rangle$; но если $f(0) \oplus f(1) = 1$ (сбалансированная), то выходное состояние равно $|--\rangle$. Посредством одного вызова U_f мы узнаём, является ли f постоянной либо сбалансированной, *даже если мы не узнаём, какой является $f(x)$ для любого конкретного входного x*.

То, что происходит, когда мы применяем U_f с входным кубитом в состоянии $|+\rangle$, можно рассматривать как то, что состояние входного кубита представляет вопрос, который мы задаем об f. Если мы задаем вопрос $|0\rangle$, то получим ответ $f(0)$, а если мы задаем вопрос $|1\rangle$, то получим ответ $f(1)$. Вопрос $|+\rangle$ тогда говорит нам о $f(0) \oplus f(1)$, не говоря ни о $f(0)$, ни о $f(1)$, взятых сами по себе.

Однако когда мы задаем вопросы в суперпозиции, подобной этой, роли «входа» и «выхода» не так сразу ясны, как они ясны классически. В частности, входы $|0\rangle$ и $|1\rangle$ приводят к переворачиванию выходного кубита, в то время как вход $|+\rangle$ приводит к переворачиванию *входного кубита*, при условии что мы начинаем выходной кубит в состоянии $|-\rangle$. В общем случае двухкубитовые операции, такие как U_f, преобразовывают все пространство кубитов, на которое они воздействуют, – наше разделение на входы и выходы является одним из способов интерпретирования действия U_f.

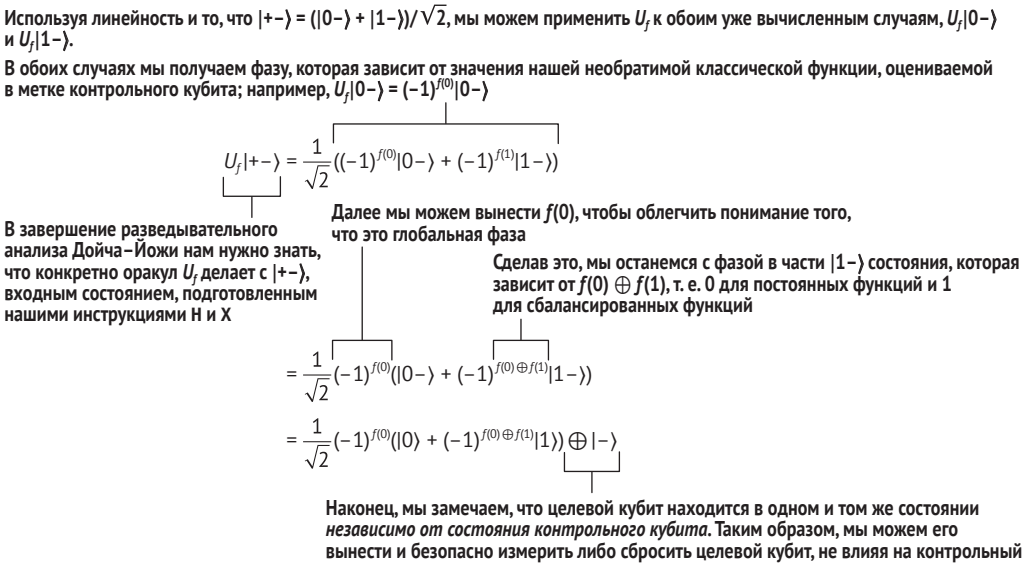


Рис. 8.14 Отработка последних двух шагов алгоритма Дойча–Йожи. Записав действия оракула на состоянии |+−⟩, мы видим, как измерение контрольного кубита в конце говорит о том, какую функцию, постоянную либо сбалансированную, оракул представляет.

Тот факт, что состояние входного кубита изменяется, основываясь на преобразованиях, определенных в выходном кубите, является примером фазовой отдачи, квантового эффекта, используемого алгоритмом Дойча–Йожи. В следующих двух главах мы будем применять фазовую отдачу для разведывательного анализа новых алгоритмов, таких как алгоритмы, используемые в квантовом обнаружении и квантово-химических симуляциях.

Глубокое погружение: расширение алгоритма Дойча–Йожи

Хотя мы рассмотрели здесь функции только с однобитовыми входами, алгоритму Дойча–Йожи всегда требуется лишь один запрос для данных любого размера на входе в нашу функцию или выходе из нее.

Для кодирования двухкубитовой функции $f(x_0, x_1)$ мы можем ввести трехкубитового оракула $U_f|x_0\, x_1\, y\rangle = |x_0\, x_1\rangle \otimes / f(x_0, x_1) \oplus y\rangle$. Например, возьмем $f(x_0, x_1) = x_0 \oplus x_1$. Эта функция является сбалансированной, так как $f(0, 0) = f(1, 1) = 0$, но $f(0, 1) = f(1, 0) = 1$. При применении U_f к трехкубитовому состоянию |++−⟩ = (|00⟩ + |01⟩ + |10⟩ + |11⟩) ⊗ |−⟩ мы получаем (|00⟩ – |01⟩ – |10⟩ + |11⟩) ⊗ |−⟩ = |−−−⟩. Используя X-базисное измерение, мы можем различить это по постоянной функции, такой как $f(x_0, x_1) = 0$, которая на выходе даст нам |−⟩.

Коль скоро нам обещано, что f является либо постоянной, либо сбалансированной, один и тот же шаблон сохраняется независимо от того, сколько битов f принимает на входе: мы можем узнать один бит данных о поведении f с помощью одного вызова U_f. Прекрасный повод поговорить об O(1)! Если вы незнакомы с нотацией «О большое», то обратитесь к книге «Грокаем алгоритмы» Адитьи Бхаргавы (*Grokking Algorithms*, Aditya Bhargava, Manning, 2016).

В следующей главе мы будем опираться на полученные здесь навыки, занявшись выяснением того, как *алгоритм фазового оценивания* позволяет использовать побочные технологии, такие как квантовые сенсоры.

Резюме

- Квантовые алгоритмы – это последовательность шагов, которым мы можем следовать, чтобы решать задачу с помощью квантовых компьютеров. Мы можем имплементировать квантовые алгоритмы путем написания квантовых программ на Q#.
- Алгоритм Дойча–Йожи является примером квантового алгоритма, который позволяет нам решать вычислительную задачу с меньшими ресурсами, чем любой возможный классический алгоритм.
- Если мы хотим встраивать классические функции в квантовый алгоритм или программу, то нам нужно делать это *обратимо*. Мы можем строить специальные виды квантовых операций, именуемые оракулами, которые позволяют нам представлять классические функции, применяемые к квантовым данным.
- Алгоритм Дойча–Йожи позволяет нам проверять, являются ли оба выходных значения однобитового оракула одинаковыми либо разными, используя только один вызов этого оракула; мы не узнаём какое-либо конкретное выходное значение; вместо этого непосредственно узнаём глобальное свойство, которое нас интересует.
- Алгоритм Дойча–Йожи демонстрирует шаблоны «ботинки и носки», которые обычно также встречаются в других квантовых алгоритмах. Нам часто нужно будет применять внешнюю операцию, применять внутреннюю операцию, а затем откатывать внешнюю операцию (либо брать ее адъюнкт).
- Технический прием «фазовая отдача» позволяет нам ассоциировать применяемую квантовой операцией фазу с контрольным кубитом вместо целевого кубита. Мы увидим больше примеров этого приема в алгоритмах, которые узнаем далее.

9 Квантовая телеметрия[1]: это не просто фаза

Эта глава охватывает следующие ниже темы:

- как квантовые операции могут получать полезную информацию о неизвестных операциях с помощью фазовой отдачи;
- создание новых типов на языке Q#;
- выполнение исходного кода Q# из главной программы на Python;
- понимание важных свойств и форм поведения собственных состояний и фазы;
- программирование контролируемых квантовых операций на языке Q#.

В предыдущей главе мы имплементировали наш первый квантовый алгоритм, Дойча–Йожи, на языке Q#. Помогая Нимуэ и Мерлину играть в Воспитателя короля, мы увидели, как технические приемы квантового программирования, такие как *фазовая отдача*, снабжают нас преимуществами в решении задач. В этой главе мы рассмотрим алгоритмы фазового оценивания, которые мы можем использовать в наших квантовых программах для решения различных типов задач. И мы снова вернемся

[1] Квантовая телеметрия (quantum sensing), или зондирование, обычно используется для описания одного из следующих: 1) использование квантового объекта для измерения физической величины (классической или квантовой); квантовый объект характеризуется квантованными энергетическими уровнями; 2) использование квантовой когерентности (т. е. волнообразных состояний пространственной или временной суперпозиции) для измерения физической величины; 3) использование квантовой запутанности для повышения чувствительности или точности измерения за пределами того, что возможно классически. – *Прим. перев.*

в Камелот; на этот раз мы будем использовать игру между Ланселотом и Дагонетом, чтобы проиллюстрировать поставленную задачу.

9.1 Фазовое оценивание: использование полезных свойств кубитов для измерения

На протяжении всей книги мы видели, что игры бывают полезным способом усвоения концепций квантовых вычислений. Например, в предыдущей главе игра Нимуэ с Мерлином позволила нам разведать наш первый квантовый алгоритм: алгоритм Дойча–Йожи. В этой главе мы будем использовать еще одну игру, чтобы узнать, как узнавать фазы квантовых состояний с помощью фазовой отдачи, технического приема квантовой разработки, используемого в алгоритме Дойча–Йожи и многих других квантовых алгоритмах.

Для игры в этой главе давайте вернемся назад и посмотрим, чем занимался Ланселот. Пока Нимуэ и Мерлин решают судьбу королей, мы находим Ланселота и придворного шута Дагонета, играющих в игру на угадывание. Поскольку у них было время поиграть, Дагонету стало скучно, и он хочет «позаимствовать» несколько квантовых инструментов Нимуэ, чтобы сделать их игру чуть-чуть интереснее.

9.1.1 Часть и частичное применение

В новой игре Дагонета, вместо того чтобы Ланселот угадывал число, Дагонет просит Ланселота угадывать, что конкретно квантовая операция делает с одним кубитом, давая Ланселоту возможность вызывать его с разными входными данными. Учитывая, что все однокубитовые операции являются поворотами, это хорошо подходит для игры. Дагонет выбирает угол поворота вокруг конкретной оси, а Ланселот вводит число в операцию Дагонета, которая изменяет масштаб поворота, применяемого Дагонетом. То, какую ось выбирает Дагонет, на самом деле не имеет значения, так как игра заключается в том, чтобы угадывать угол поворота. Для удобства здесь поворот Дагонета всегда происходит вокруг оси Z. Наконец, Ланселот может измерить кубит и использовать свое измерение, чтобы угадать изначальный угол поворота Дагонета. На рис. 9.1 приведена блок-схема следующих ниже шагов.

1. Дагонет подбирает секретный угол для однокубитовой операции поворота.
2. Дагонет готовит операцию для использования Ланселотом, которая скрывает секретный угол и дает Ланселоту ввести одно дополнительное значение в форме числа (мы назовем его масштабом), которое будет умножено на секретный угол, чтобы получить общий угол операции поворота.
3. Наилучшая стратегия Ланселота в игре состоит в том, чтобы выбирать много масштабных значений и оценивать вероятность из-

мерить One для каждого значения. Для этого ему нужно выполнить следующие ниже шаги много раз для каждого значения масштаба из множества:

a подготовить состояние |+⟩ и ввести значение масштаба в поворот Дагонета. Он использует состояние |+⟩, потому что знает, что Дагонет поворачивает вокруг оси *Z*; и для этого состояния его повороты приведут к изменению локальной фазы, которую он может измерить;

b после подготовки каждого состояния |+⟩ Ланселот может поворачивать его с помощью секретной операции, измерять кубит и записывать измерение.

4 Теперь у Ланселота есть данные, касающиеся его масштабного фактора и вероятности того, что он измерил One для этого масштабного фактора. Он может выполнить подгонку этих данных в своей голове и получить угол Дагонета на основе подогнанных параметров (он и в самом деле – величайший рыцарь в стране). А мы задействуем Python, который поможет нам сделать то же самое!

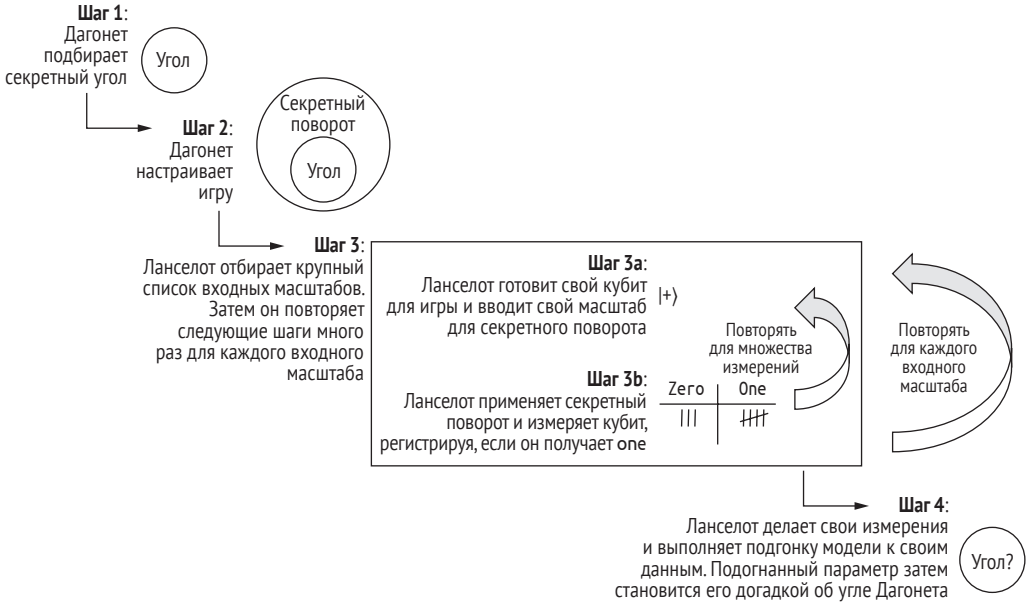


Рис. 9.1 Шаги в игре Дагонета и Ланселота. Дагонет скрывает секретный угол поворота в операции, а Ланселот должен выяснить, что это за угол

Обратите внимание, что *это* действительно игра, так как Ланселот не может измерить этот поворот непосредственно с помощью всего одного измерения. Если бы он мог, то это нарушило бы теорему о запрете клонирования, и он вышел бы за пределы законов физики. Как рыцарь Круглого стола Ланселот связан не только долгом и честью, но и законами физики, поэтому он должен играть в игру Дагонета по правилам.

Глубокое погружение: усвоение оси с помощью гамильтонианова обучения

В этой главе мы сосредоточимся на случае, когда ось поворота Дагонета известна, но нам нужно узнать его угол. Этот случай соответствует общей задаче в физике, где нам поручено усвоить ларморову прецессию кубита в магнитном поле. Изучение *прецессий Лармора* полезно не только для строительства кубитов; оно также позволяет обнаруживать очень малые магнитные поля и создавать очень точные сенсоры.

В более общем случае, однако, мы можем усвоить гораздо больше, чем один угол поворота. Случай, когда ось также неизвестна, является примером общей задачи, именуемой *гамильтониановым обучением*, богатой областью исследований в области квантовых вычислений. В гамильтониановом обучении мы реконструируем физическую модель кубита или реестра кубитов, используя игру, очень похожую на ту, которую мы рассматриваем в этой главе.

Давайте перейдем к прототипированию этой игры на языке Q#. Будет полезно иметь доступ к разным частям стандартных библиотек Q#, поэтому мы начнем с добавления следующих ниже инструкций `open` в верхнюю часть нашего файла Q#, operations.qs.

Листинг 9.1 operations.qs: открытие пространств имен Q# для игры

```
namespace PhaseEstimation {                                    ❶
    open Microsoft.Quantum.Intrinsic;                          ❷
    open Microsoft.Quantum.Convert as Convert;                 ❸
    open Microsoft.Quantum.Measurement as Meas;                ❹
    open Microsoft.Quantum.Arrays as Arrays;                   ❺
    // ...
```

❶ Все инструкции open в файле Q# появляются сразу после объявления пространства имен.
❷ Как и прежде, открытие Microsoft.Quantum.Intrinsic предоставляет нам доступ ко всем основным инструкциям (R1, Rz, X и т. д.), которые мы можем отправлять на квантовое устройство.
❸ Мы также можем дать пространствам имен псевдоним при их открытии подобно тому, как мы создаем псевдонимы пакетов и модулей Python при их импортировании. Здесь мы сокращаем Microsoft.Quantum.Convert, чтобы позже иметь возможность использовать функции конверсии типов в этом пространстве имен путем добавления к ним приставки Convert.
❹ Мы можем сделать операцию MResetZ из предыдущих глав доступной в виде Meas.MResetZ для документирования источника операции.
❺ Последнее пространство имен, которое нам нужно открыть в этой главе, – это Microsoft.Quantum.Arrays, которое предоставляет полезные функции и операции для работы с массивами.

В листинге 9.2 показан пример квантовой операции, необходимой Дагонету для имплементирования поворота, с которым он и Ланселот будут играть. Как и другие повороты, наша новая поворотная операция возвращает `Unit` (тип пустого кортежа `()`), указывая на отсутствие значимого поворота от операции. В фактическом теле операции мы можем найти угол поворота, умножив угол скрытого угла, `angle`, Дагонета на масштабный фактор, `scale`, Ланселота.

Листинг 9.2 operations.qs: операция по настройке игры

```
operation ApplyScaledRotation(
    angle : Double, scale : Double,          ❶
    target : Qubit)
: Unit is Adj + Ctl {                        ❷
    R1(angle * scale, target);               ❸
}
```

❶ В целях игры на угадывание нам нужна квантовая операция, которая принимает два классических аргумента: один, который передается Дагонетом, и один, который передается Ланселотом.
❷ Часть сигнатуры Adj + Ctl указывает на то, что эта операция поддерживает функтор Adjoint, который мы впервые увидели в главе 8, а также функтор Controlled, который мы увидим позже в этой главе.
❸ Поворотная операция R1 здесь почти идентична операции Rz, которую мы уже встречали несколько раз. Разница между R1 и Rz станет важной, когда мы добавим функтор Controlled.

ПРИМЕЧАНИЕ При написании исходного кода Q# в отдельном файле (т. е. не из среды блокнотов Jupyter Notebook) все операции и функции должны быть определены в пространстве имен. Это помогает упорядочивать наш исходный код и затрудняет появление конфликтов нашего кода с исходным кодом в различных библиотеках, которые мы используем в квантовом приложении. Для краткости мы нередко не показываем объявления пространств имен, однако полный исходный код всегда можно найти в репозитории примеров из этой книги на Github: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

В целях ознакомления с визуальным представлением настройки игры Дагонетом обратитесь к рис. 9.2.

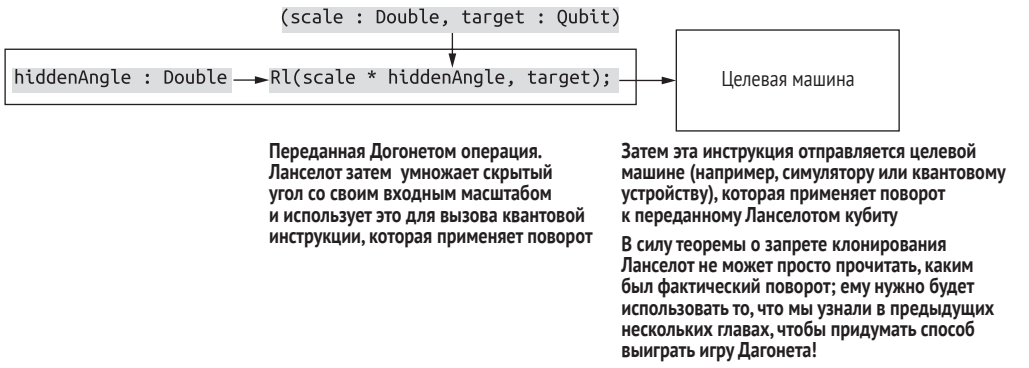


Рис. 9.2 Использование частичного применения для сокрытия секретного угла во время игры Дагонета на угадывание. Оракул, получаемый Ланселотом, имеет вход для его масштабного параметра scale, и затем он может выбрать целевую машину для использования операции, но не может «заглянуть внутрь» операции, чтобы увидеть секретный угол

Глубокое погружение: почему бы просто не измерить угол?

Может показаться, что для того, чтобы угадать скрытый угол Дагонета, Ланселоту приходится перепрыгивать через множество обручей. В конце концов, что, кроме долга и чести, мешает Ланселоту передать масштаб 1.0, а затем просто прочитать угол из фазы, примененной к его кубиту? Оказывается, в этом случае теорема о запрете клонирования снова выставляет нам страйк, говоря о том, что Ланселот никогда не сможет узнать фазу из одной операции измерения.

Легче всего понять это, если на мгновение притвориться, что Ланселот может это сделать, а затем посмотреть, что пойдет не так. Предположим, что Дагонет скрывает угол $\pi/2$ и что Ланселот готовит свой кубит в состоянии $|+\rangle = 1/\sqrt{2}\,(|0\rangle + |1\rangle)$. Если Ланселот затем передаст 1.0 в качестве своего масштаба, то его кубит окажется в состоянии $1/\sqrt{2}\,(|0\rangle + i|1\rangle)$. Если бы Ланселот мог непосредственно измерить фазу $e^{i\pi/2} = +i$, чтобы угадать угол Дагонета одним-единственным измерением, то он мог бы использовать это для подготовки еще одной копии состояния $1/\sqrt{2}\,(|0\rangle + i|1\rangle)$, даже если бы Ланселот не знал правильного базиса для измерения. И действительно, волшебное измерительное устройство Ланселота также должно работать, если Дагонет скрывает угол π, и в этом случае Ланселот оказывается с кубитом в состоянии $1/\sqrt{2}\,(|0\rangle - |1\rangle)$.

Говоря по-другому, если бы Ланселот мог вычислить угол Дагонета, непосредственно измеряя фазы, то он мог бы делать копии произвольных состояний вида $1/\sqrt{2}\,(|0\rangle + e^{i\varphi}|1\rangle)$ для любого угла φ, *не зная* φ *заранее*. Это довольно сильно нарушает теорему о запрете клонирования, поэтому мы можем с уверенностью заключить, что Ланселоту потребуется немного больше работы, чтобы выиграть игру Дагонета.

После того как мы определили операцию в таком виде, Дагонет может применить характерную для Q# функциональность частичного применения, которую мы впервые увидели в главе 8, чтобы скрыть свое входное значение. Затем Ланселот получает операцию, которую он может применить к своим кубитам, но не так, чтобы непосредственно увидеть угол, который он пытается угадать.

Используя `ApplyScaledRotation`, Дагонет может легко сделать операцию, для того чтобы Ланселот мог ее вызвать. Например, если Дагонет выбирает угол 0.123, то он может его «скрыть», предоставив Ланселоту операцию `ApplyScaledRotation(0.123, _, _)`. Как и в примерах частичного применения в главе 7, символ _ обозначает слот для будущих входных данных.

Как показано на рис. 9.3, поскольку `ApplyScaledRotation` имеет тип `(Double, Double, Qubit) => Unit is Adj + Ctl`, предоставление только первого входного значения приводит к операции типа `(Double, Qubit) => Unit is Adj + Ctl`. Это означает, что Ланселот может предоставить входное значение типа `Double`, кубит, к которому он хочет применить свою операцию, и использовать ключевое слово `Adjoint` и функтор, который мы видели в главе 6.

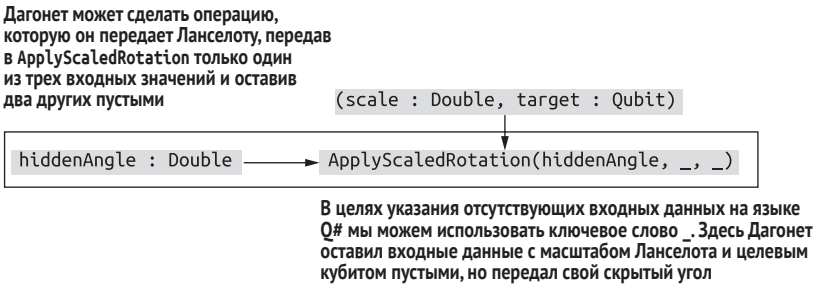


Рис. 9.3 Частичное применение операции ApplyScaledRotation с целью создания операции для Ланселота

Тот факт, что мы можем видеть значение угла в синтаксисе, вовсе *не означает*, что Ланселот может это сделать тоже. И вправду, единственное, что Ланселот может сделать с частично примененной операцией или функцией, – это вызвать ее, частично применить ее дальше либо передать в другую функцию или операцию. С точки зрения Ланселота, `ApplyScaledRotation(0.123, _, _)` представляет собой черный ящик. Благодаря этому трюку с частичным применением у него будет просто операция, которая принимает его масштабное значение и может использоваться для поворота кубита.

Мы можем упростить нашу жизнь как разработчиков на Q#, дав типу операции Ланселота имя, которое немного легче читать, чем `(Double, Qubit) => Unit is Adj + Ctl`.

В следующем далее разделе мы узнаем, как язык Q# позволяет нам аннотировать сигнатуры типов, которые мы используем для игры на угадывание между Дагонетом и Ланселотом.

9.2 Пользовательские типы

Мы уже видели, как типы играют свою роль в языке Q#, в особенности в сигнатурах функций и операций. Мы также видели, что и функции, и операции строятся по принципу «кортеж вошел, кортеж вышел». В этом разделе мы научимся создавать свои собственные типы на языке Q# и узнаем, почему это бывает удобно.

В языке Q# (как и во многих других языках) несколько типов определяются как часть самого языка: такие типы, как `Int`, `Qubit` и `Result`, которые мы уже встречали.

> **ДЛЯ СПРАВКИ** Полный список этих базовых типов см. в документации по языку Q# по адресу: https://docs.microsoft.com/azure/quantum/user-guide/language/typesystem/#primitive-types.

Основываясь на этих базовых типах, мы можем создавать массивоподобные типы, добавляя `[]` после типа. Например, в игре этой главы нам, вероятно, потребуется ввести массив чисел двойной точности, чтобы

представить несколько входных значений Ланселота в операцию Дагонета. Мы можем использовать `Double[]` для указания массива значений типа `Double`.

Листинг 9.3 Определение массивоподобного типа из значений `Double` длиной 10

```
let scales = EmptyArray<Double>(10);          ❶
```

❶ emptyArray расположен в пространстве имен Microsoft.Quantum.Arrays. Не забудьте открыть это пространство имен перед выполнением указанного фрагмента кода.

Язык Q# также позволяет нам определять свои собственные типы, используя для этого инструкцию `newtype`. Эта инструкция позволяет нам объявлять новые *пользовательские типы* (*user-defined type*, аббр. *UDT*). Есть две главные причины для применения пользовательских типов:

- удобство;
- доведение своего намерения.

Первая причина является вопросом *удобства*. Иногда сигнатура типа для функции или операции может быть довольно длинной, поэтому мы можем определить свой собственный тип как своего рода укороченную нотацию. Еще одна причина, по которой мы, возможно, захотим дать имя нашему типу, вызвана желанием сообщить о *намерении*. Скажем, наша операция принимает кортеж из двух значений тип `Double`, который представляет комплексное число. Определение нового типа `Complex`, возможно, напомнит нам и нашим товарищам по команде о том, что конкретно этот кортеж представляет. Комплект инструментов для квантовой разработки предлагает несколько разных функций, операций и UDT с библиотеками Q#, таких как показано в следующем ниже примере, в котором определяется тип `Complex`.

Листинг 9.4 Как определяются комплексные числа во время выполнения исходного кода на Q#

```
namespace Microsoft.Quantum.Math {          ❶
    newtype Complex = (                     ❷
       Real: Double,
        Imag: Double
    );
}
```

❶ Комплексные числа имплементированы как UDT в пространстве имен Microsoft.Quantum.Math. Мы можем использовать их путем включения инструкции «open Microsoft.Quantum.Math;» в наше квантовое приложение.

❷ Тип Complex определяется как кортеж из двух значений типа Double, где первый элемент называется Real, а второй – Imag.

ДЛЯ СПРАВКИ Комплект инструментов для квантовой разработки имеет открытый исходный код, поэтому, если вам интересно, вы всегда можете посмотреть, как работают различные части языка

Q#, среды выполнения, компилятора и стандартных библиотек. Например, определение UDT Complex находится в файле src/Simulation/QSharpFoundation/Math/Types.qs в репозитории исполнимых файлов Q# по адресу: https://github.com/microsoft/qsharp-runtime.

Как показано на рис. 9.4, существует два способа получать разные элементы из UDT. Мы можем либо использовать именованные элементы вместе с оператором `::`, либо «разворачивать» UDT с помощью оператора `!` для возврата к изначальному типу, завернутому в UDT:

```
function TakesComplex(complex : Complex) : Unit {
    let realFromNamedItem = complex::Real;          ❶
    let (real, imag) = complex!;                    ❷
}
```

❶ Поскольку UDT Complex определяется с помощью именованного элемента Real, мы можем получить доступ к этому элементу как ::Real, чтобы вернуть реальную часть наших входных данных.

❷ В качестве альтернативы, поскольку Complex определяется как обертывание кортежа типа (Double, Double), оператор развертывания ! возвращает нас к реальной и мнимой частям комплексного числа без обертки UDT.

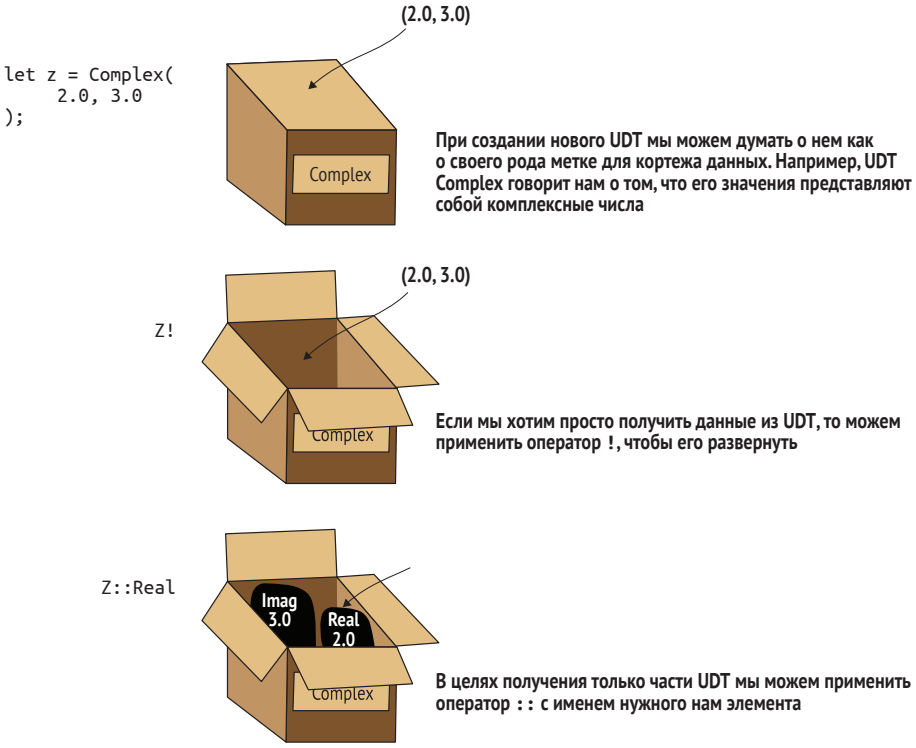


Рис. 9.4 Применение операторов `::` и `!` с пользовательскими типами. UDT можно рассматривать как помеченный кортеж данных. Оператор `::` позволяет получать доступ к этим данным по имени, в то время как оператор `!` распаковывает UDT в его базовый тип

ДЛЯ СПРАВКИ Оба способа работы с UDT полезны в разных случаях, но в большинстве из них в этой книге мы будем придерживаться использования именованных элементов и оператора `::`.

После определения нового UDT он также может выступать в качестве способа создания нового экземпляра этого типа. Например, тип `Complex` действует как функция, которая создает новое комплексное число с входом в виде кортежа из двух значений типа `Double`. Это похоже на Python, где типы также являются функциями, создающими экземпляры этого типа.

Листинг 9.5 Создание комплексного числа с помощью UDT Complex

```
let imaginaryUnit = Complex(0.0, 1.0);          ❶
```

❶ **Определение UDT с помощью newtype также определяет новую функцию с тем же именем, что и тип, которая возвращает значения нашего нового UDT. Например, мы можем вызвать Complex как функцию с двумя входными значениями типа Double, представляющими действительную и мнимую части нового комплексного значения. Здесь мы определяем комплексное значение, представляющее 0 + 1.0*i* (0+1j в нотации языка Python), также именуемое мнимой единицей.**

Упражнение 9.1: пользовательские типы для стратегий

В главе 4 мы использовали Python'овские аннотации типов для представления концепции *стратегии* в игре CHSH. Пользовательские типы в языке Q# можно применять аналогичным образом. Попробуйте определить новый UDT для стратегий CHSH, а затем примените новый UDT для обертки постоянной стратегии из главы 4.

Подсказка: наша и Евина части стратегии могут быть представлены как операции, которые берут `Result` и выдают `Result`: т. е. в виде операций типа `Result => Result`.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

Для игры в этой главе мы определили новый UDT для обозначения того, как мы намереваемся его использовать, и в качестве удобной укороченной записи типа операции, которую Ланселот получает как свою часть игры на угадывание. В следующем ниже листинге мы видим определение этого нового типа под названием `ScalableOperation` в виде кортежа с одним именованным входным значением `Apply`.

Листинг 9.6 operations.qs: настройка для квантовой игры на угадывание

```
newtype ScalableOperation = (                          ❶
    Apply: ((Double, Qubit) => Unit is Adj + Ctl)      ❷
);

function HiddenRotation(hiddenAngle : Double)
: ScalableOperation {                                  ❸
    return ScalableOperation(
        ApplyScaledRotation(hiddenAngle, _, _)         ❹
    );
}
```

❶ Мы можем объявить новый UDT с помощью оператора newtype, указав имя нового типа и определив базовый тип, на котором основан новый тип.

❷ Мы можем давать имена различным элементам в UDT, используя тот же синтаксис, что и объявление сигнатуры для операции или функции. Здесь новый UDT имеет один элемент с именем Apply, который позволяет вызывать операцию, обернутую операцией ScalableOperation.

❸ После определения мы можем использовать новый UDT, как и любой другой тип. Здесь мы определяем функцию, которая выдает значения типа ScalableOperation.

❹ Мы можем легко выдавать выходные значения, вызывая ScalableOperation с операцией, которая будет завернута в наш новый UDT. В данном примере мы можем создать новые экземпляры ScalableOperation, используя то же частичное применение операции ApplyScaledRotation, которое мы встречали ранее в этой главе.

ДЛЯ СПРАВКИ При определении входных значений для функций и операций на Q# эти входные значения имеют имена, начинающиеся со строчных букв. Однако в листинге 9.6 именованный элемент `Apply` в `ScalableOperation` начинается с заглавной буквы. Это связано с тем, что входные значения для функции или операции имеют смысл только в пределах этого вызываемого объекта, в то время как именованные элементы несут некий более широкий смысл. Мы можем использовать написание входных значений и именованных элементов заглавными буквами, чтобы делать самоочевидным то, где искать их определения.

Функция `HiddenRotation`, определенная в листинге 9.6, помогает нам имплементировать игру Ланселота и Дагонета, предлагая нам способ дать Дагонету возможность скрыть свой угол. Вызов функции `HiddenRotation` с углом Дагонета возвращает новую операцию `ScalableOperation`, которую Ланселот может вызвать, чтобы собрать данные, необходимые ему для угадывания скрытого угла. На рис. 9.5 показана новая настройка игры Дагонета.

Имея несколько новых типов и способ, которым Дагонет скрывает свой угол, давайте продолжим имплементирование остальной части игры! У нас есть все необходимое для следующего шага: оценивания вероятности каждого измерения, которое мы делаем во время игры Ланселота и Дагонета. Это очень похоже на то, как мы оцениваем вероятность подбрасывания монеты; см. рис. 9.6.

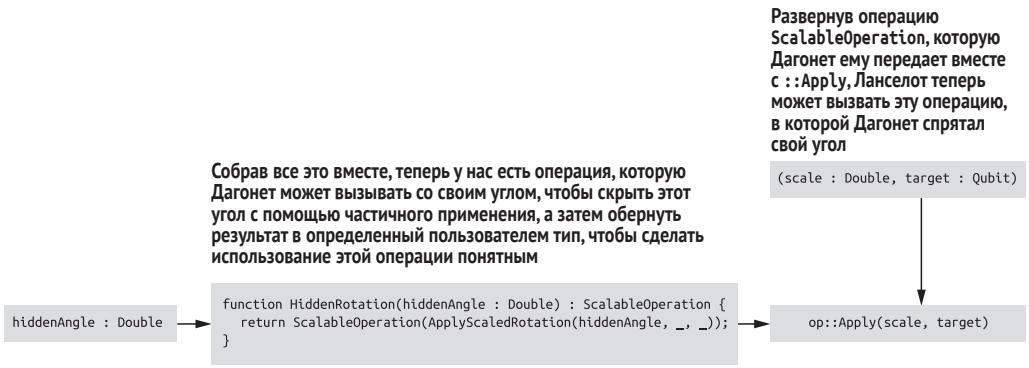


Рис. 9.5 Игра Дагонета на угадывание с частичным применением и пользовательскими типами. Дагонет использует функцию для строительства передаваемого Ланселоту пользовательского типа, представляющего поворот со своим скрытым углом

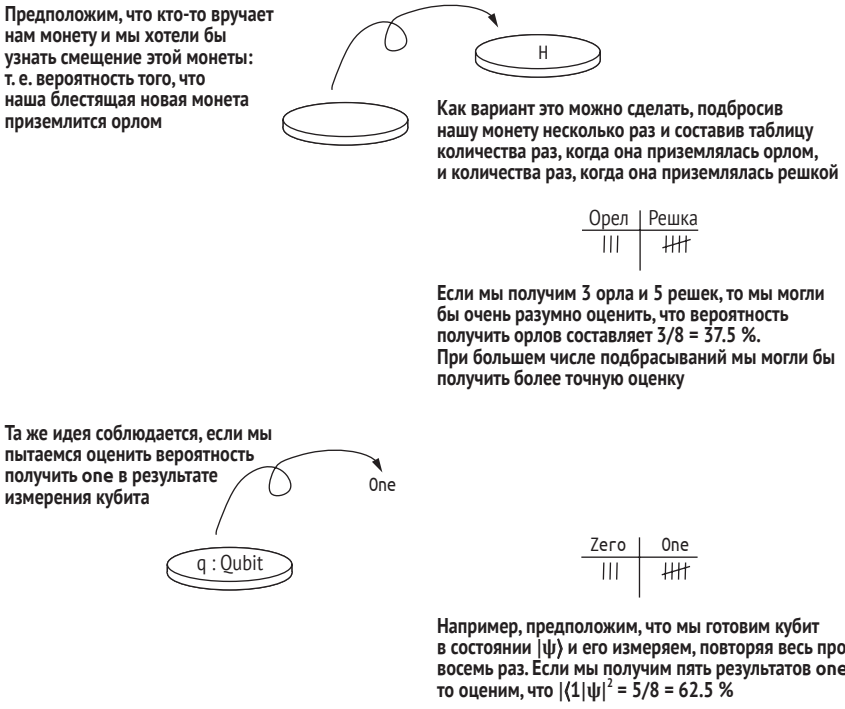


Рис. 9.6 Выполненное Ланселотом оценивание аналогично оцениванию исхода подбрасывания монеты. Он может оценить смещение монеты, подбросив ее много раз и подсчитав количество «орлов». Аналогичным образом Ланселот может подготавливать один и тот же кубит в одном и том же состоянии много раз, при этом каждый раз его измерять и подсчитывать результаты измерений

Давайте посмотрим, как будет выглядеть исходный код для этой процедуры оценивания подбрасываний в игре Дагонета. Обратите внимание, что поскольку Ланселот и Дагонет договорились, что ось *Z* должна

быть осью поворота в их игре, Ланселот может подготовить свой целевой кубит в состоянии |+⟩, чтобы поворот Дагонета что-то делал.

Листинг 9.7 operations.qs: оценивание вероятности измерить |1⟩

```
operation EstimateProbabilityAtScale(
    scale : Double,                                    ❶
    nMeasurements : Int,                               ❷
    op : ScalableOperation)                            ❸
: Double {                                             ❹
    mutable nOnes = 0;                                 ❺
    for idx in 0..nMeasurements - 1 {
        use target = Qubit();                          ❻
        within {                                       ❼
            H(target);                                 ❽
        } apply {
            op::Apply(scale, target);                  ❾
        }
        set nOnes += Meas.MResetZ(target) == One
                     ? 1 | 0;                          ❿
    }
    return Convert.IntAsDouble(nOnes) /                ⓫
           Convert.IntAsDouble(nMeasurements);
}
```

❶ Операция Ланселота должна принимать значение типа Double, представляющее масштаб, который он выбирает для выполнения операции, данной ему Дагонетом.
❷ Ланселот выбирает число повторов измерения своего кубита, чтобы получить свою оценку вероятности.
❸ Последнее входное значение имеет тип ScalableOperation, т. е. UDT, объявленный ранее в этой главе. Это входное значение представляет собой операцию, которую Дагонет предоставляет Ланселоту.
❹ На выходе Ланселот хочет вернуть оценочную вероятность, поэтому мы можем объявить ее как выходное значение типа Double.
❺ В целях отслеживания числа исходов |1⟩, наблюдавшихся до этого момента, мы определяем мутируемую переменную со значением типа Int, равную 0.
❻ Для каждого измерения нам нужно выделить кубит, который является целью для операции Дагонета.
❼ Использует ключевые слова «within» и «apply» для шаблона «ботинки и носки», о котором мы узнали в главе 8.
❽ Имплементирует стратегию Ланселота, используя операцию H для подготовки кубита в состоянии |+⟩.
❾ После подготовки входных данных для операции Дагонета мы ее вызываем, используя ::Apply, чтобы распаковать UDT ScalableOperation.
❿ Тернарный оператор ?| (очень похожий на условное выражение if ... else в языке Python или оператор ?: в C, C++ и C#) обеспечивает удобный способ наращивания значения в nOnes.
⓫ В целях получения окончательной оценки вероятности измерить |1⟩ мы вычисляем отношение числа One к суммарному числу. Функция Convert.IntAsDouble помогает нам возвращать число с плавающей точкой.

В листинге 9.7 блок `within/apply` обеспечивает, чтобы кубит Ланселота вернулся на правильную ось. Мы можем подсчитать число раз, когда

окончательное измерение возвращает результат `One`, добавляя `1` или `0` в `nOnes`. Здесь тернарный оператор `?|` (очень похожий на условное выражение `if … else` в языке Python или оператор `?:` в C, C++ и C#) обеспечивает удобный способ наращивания значений в nOnes.

Операция под любым другим именем

Возможно, вы заметили, что операции, как правило, именуются глаголами, в то время как функции, как правило, именуются существительными. Это помогает запоминать различие, которое вы видели в главе 7: функция – это нечто, в то время как операция что-то делает. Последовательность в использовании имен помогает понимать принцип работы квантовой программы, поэтому подобные соглашения используются во всем языке Q# и его библиотеках.

Теперь мы можем написать операцию, которая выполняет всю игру и возвращает все, что нужно Ланселоту, чтобы угадать скрытый угол Дагонета.

Листинг 9.8 operations.qs: выполнение полной игры

```
operation RunGame(
    hiddenAngle : Double, scales : Double[], nMeasurementsPerScale : Int
) : Double[] {
    let hiddenRotation = HiddenRotation(hiddenAngle);          ❶
    return Arrays.ForEach(                                     ❷
        EstimateProbabilityAtScale(                            ❸
            _,
            nMeasurementsPerScale,
            hiddenRotation
        ),
        scales                                                 ❹
    );
}
```

❶ Создает новое значение ScalableOperation, в котором скрывается угол Дагонета, используя функцию HiddenRotation, которую мы написали ранее.
❷ Операция ForEach в Microsoft.Quantum.Arrays (которой мы дали сокращенное название Arrays) выполняет операцию и применяет ее к каждому элементу массива scales.
❸ В целях получения операции, которую мы передаем в ForEach, мы используем частичное применение, чтобы изолировать число измерений, которые Ланселот делает в каждом отдельном scale, и скрытую операцию, которую ему дал Дагонет.
❹ При передаче scales в качестве второго входного значения в операцию ForEach каждый элемент scales подставляется в слот частичного применения _.

ПРИМЕЧАНИЕ Может показаться забавным, что операция `ForEach` действует как `map` в Python и других языках, когда в Q# также есть `Microsoft.Quantum.Arrays.Mapped`. Критическое различие заключается в том, что `ForEach` выполняет операцию, а `Mapped` – функцию.

Для того чтобы Ланселот действительно понимал все данные, которые он получает из своей программы на Q#, было бы полезно применить несколько старых добрых классических методов науки о данных. Поскольку Python отлично с этим справляется, выполнение нашей новой операции `RunGame` из главной программы Python может стать отличным способом помочь Ланселоту.

9.3 Беги, змейка, беги: выполнение Q# из Python

В предыдущих главах мы выполняли наш исходный код Q# в среде блокнотов Jupyter Notebook с ядром Q#. В этой главе мы хотим обратиться к другому способу выполнения исходного кода Q#: из Python. Вызов Q# из Python бывает полезен в различных сценариях, в особенности если мы хотим предобработать данные перед их использованием на Q# либо если хотим визуализировать выходные данные нашей квантовой программы.

Давайте приступим к написанию файлов для имплементирования игры Дагонета и Ланселота. В целях испытания взаимодействия Q# и Python мы будем использовать главную программу Python для выполнения программы Q#. Это означает, что у нас будет два файла игры: operations.qs и файл host.py, который мы будем использовать непосредственно для выполнения игры. Давайте рассмотрим файл host.py подробнее, чтобы увидеть, каким образом можно взаимодействовать с Q# из Python; см. рис. 9.7.

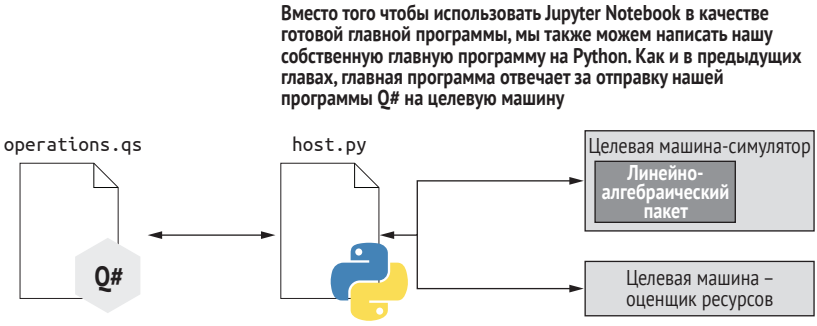


Рис. 9.7 Использование главных программ, написанных на Python, для Q#. Как и хост Jupyter Notebook, программа Python может координировать отправку программы Q# на конкретную целевую машину и сбор результатов

Вся необходимая нам функциональность, которая обеспечивает взаимодействие между Python и Q#, поставляется Python'овским пакетом qsharp.

ДЛЯ СПРАВКИ В приложении A к книге содержатся полные инструкции по установке пакета Python `qsharp`.

Имея пакет `qsharp`, мы сможем его импортировать, как и любой другой пакет Python. Давайте посмотрим на небольшой пример файла Python, где мы можем увидеть это в действии.

Листинг 9.9 qsharp-interop.py: использование кода Q# непосредственно в Python

```
import qsharp                                          ❶

prepare_qubit = qsharp.compile("""                     ❷
    open Microsoft.Quantum.Diagnostics;                ❸

    operation PrepareQubit(): Unit {                   ❹
        using (qubit = Qubit()) {
                DumpMachine();
        }
    }
    """)

if __name__ == "__main__":
    prepare_qubit.simulate()                           ❺
```

❶ Пакет qsharp необходимо импортировать так же, как и любой другой пакет Python.
❷ Мы также хотим использовать вызываемый объект, определенный как prepare_qubit, поэтому будем использовать метод simulate из пакета qsharp, который выполняет ранее скомпилированные фрагменты кода Q#.
❸ Использует Python'овскую функцию qsharp.compile, чтобы взять строку, содержащую код Q#, и скомпилировать ее для использования в файле Python.
❹ Описанная в этой строке кода Q# операция просто подготавливает кубит в состоянии |0⟩ и использует DumpMachine, чтобы показать, что целевая машина знает об этом кубите.
❺ Как и в обычном файле Q#, нам нужно включить инструкции open, чтобы использовать разные части стандартной библиотеки Q#.

Давайте попробуем выполнить скрипт qsharp-interop.py из листинга 9.9.

Листинг 9.10 Выполнение листинга 9.9

```
$ python qsharp-interop.py
Preparing Q# environment...
# wave function for qubits with ids (least to most significant): 0
|0⟩:     1.000000 + 0.000000 i  ==      ******************** [  1.000000  ]
➥     --- [  0.00000 rad  ]
|1⟩:     0.000000 + 0.000000 i  ==                           [  0.000000  ]
```

ДЛЯ СПРАВКИ Если вы выполняете код из среды Q# Jupyter Notebook, то выходные данные фрагмента на языке Q# будут выглядеть по-другому. Пример см. на рис. 9.9 ниже в этой главе.

Из выходных данных в листинге 9.10 мы видим, что он действительно подготавливает состояние |0⟩, поскольку единственным членом в выходных данных, имеющим коэффициент 1,0, является состояние |0⟩.

Пакет `qsharp` также ищет операции или функции Q#, определенные в файлах `*.qs`, в том же каталоге, что и наша программа Python. В данном случае, когда мы перейдем к остальной части этой главы, мы добавим исходный код в файл Q# под названием operations.qs. Это довольно удобный способ начать наш файл host.py для игры. Загруженный пакет `qsharp` затем позволяет нам импортировать операции и функции из пространств имен в файлы Q# в том же каталоге, что и host.py. Мы видели `RunGame` ранее и вскоре увидим `RunGameUsingControlledRotations`.

Листинг 9.11 host.py: начало игры с оцениванием фазы

```
import qsharp                                                    (1)
from PhaseEstimation import RunGame, RunGameUsingControlledRotations  (2)

from typing import Any                                           (3)
import scipy.optimize as optimization
import numpy as np

BIGGEST_ANGLE = 2 * np.pi
```

(1) Импортирует пакет Q# Python.
(2) Импортирует операции RunGame и RunGameUsingControlledRotations из operations.qs для автоматического создания объектов Python, представляющих каждую операцию Q#, которую мы импортируем.
(3) Любая остальная часть импорта помогает с подсказками типов в среде Python, визуализированию результатов нашей симуляции Q# и подгонке данных измерений для получения окончательной догадки Ланселота.

Теперь, когда мы импортировали и настроили наш файл Python, давайте напишем `run_game_at_scales`: функцию, которая вызывает операции Q#.

Листинг 9.12 host.py: функция Python, вызывающая операции Q#

```
def run_game_at_scales(scales: np.ndarray,
                       n_measurements_per_scale: int = 100,
                       control: bool = False
    ) -> Any:                                                    (1)
    hidden_angle = np.random.random() * BIGGEST_ANGLE            (2)
    print(f"Шшшш, скрытый угол равен {hidden_angle}, удачи!")
    return (                                                     (3)
        RunGameUsingControlledRotations
        if control else RunGame
    ).simulate(                                                  (4)
        hiddenAngle=hidden_angle,
        nMeasurementsPerScale=n_measurements_per_scale,
        scales=list(scales)
    )
```

(1) Устанавливает подсказку типа возвращаемого значения равным Any, что говорит Python не беспокоиться о проверке типа значения, возвращаемого из этой функции.
(2) Дагонет выбирает скрытый угол, который он хочет, чтобы Ланселот угадал.

❸ **Возвращаемое значение для run_game_at_scales обусловлено аргументом control, позволяя нам выбирать между двумя симуляциями, которые мы разработаем для этой игры (сейчас мы используем control=False).**

❹ **При импортировании пакетом qsharp этих операций их представления на Python имеют метод под названием «simulate», который принимает необходимые аргументы и передает их в симулятор Q#.**

Этот файл Python должен выполняться как скрипт, поэтому нам также нужно определить `__main__`. Именно здесь можно сделать то, что Ланселот делает в своей голове, используя нашу главную программу на Python, чтобы брать измерения и масштабы и выполнять их подгонку к модели поворота Дагонета. Наилучшая модель того, как угол поворота изменяет результаты измерений, задается следующей ниже формулой, где θ – это скрытый угол Дагонета и `scale` – масштабный фактор Дагонета:

$$\Pr(1) = \sin\left(\frac{\theta * \text{scale}}{1}\right)^2.$$

Упражнение 9.2: обращение к старому

Эту модель можно найти, если использовать правило Борна! Определение из главы 2 показано ниже. Попробуйте, сможете ли вы построить график результирующего значения в зависимости от масштаба Ланселота с помощью Python. Ваш график похож на тригонометрическую функцию?

$$\Pr(\text{измерение}|\text{состояние}) = |\langle \text{измерение}|\text{состояние}\rangle|^2$$

Подсказка: для измерений Ланселота часть правила Борна ⟨измерение| задается как ⟨1|. Непосредственно перед измерением его кубит находится в состоянии HR_1 (θ ∗ масштаб) $H|0\rangle$. Вы можете просимулировать операцию `R1` в пакете QuTiP, используя матричную форму из справочных материалов языка Q# по адресу: https://docs.microsoft.com/qsharp/api/qsharp/microsoft.quantum.intrinsic.r1.

Имея эту модель и данные, мы можем применить функцию `scipy.optimize` из Python'овского пакета SciPy, чтобы подогнать наши данные к модели. Значение, которое он находит для параметра θ, и является скрытым углом Дагонета! В следующем ниже листинге показан пример того, как собрать все это вместе.

Листинг 9.13 host.py: выполнение host.py как скрипта

```
if __name__ == "__main__":
    import matplotlib.pyplot as plt                          ❶
    scales = np.linspace(0, 2, 101)                          ❷
    for control in (False, True):                            ❸
        data = run_game_at_scales(scales, control=control)   ❹

        def rotation_model(scale, angle):                    ❺
```

```
        return np.sin(angle * scale / 2) ** 2
    angle_guess, est_error = optimization.curve_fit(                    ❻
        rotation_model, scales, data, BIGGEST_ANGLE / 2,
        bounds=[0, BIGGEST_ANGLE]
    )
    print(f"Скрытый угол, о котором вы думали, равен {angle_guess}!")

    plt.figure()                                                        ❼
    plt.plot(scales, data, 'o')
    plt.title("Вероятность, что Ланселот измерит One в каждом масштабе")
    plt.xlabel("Входное значение масштаба Ланселота")
    plt.ylabel("Вероятность, что Ланселот измерит One")
    plt.plot(scales, rotation_model(scales, angle_guess))
plt.show()                                                              ❽
```

❶ Этот скрипт строит график данных и результатов, поэтому нам нужно импортировать дружественную библиотеку matplotlib.
❷ Список входных данных Ланселота для игры (т. е. его масштабы) генерируется как равномерно разнесенный последовательный список чисел из np.linspace.
❸ Этот скрипт выполняет обе версии игровой симуляции, чтобы иметь возможность их сравнивать. Пока не беспокойтесь о случае control = True; мы скоро к нему вернемся.
❹ Хранит результат симуляции Q#, выполняемой на Python, в run_game_at_scales.
❺ Представляет операцию на кубите. Ланселот может взять полученные данные, подогнать их к модели и извлечь догадку об угле.
❻ Стандартная функция optimization.curve_fit пакета scipy использует модель функции, входные данные, измеренные данные и первоначальную догадку, чтобы попытаться подогнать все параметры модели.
❼ Валидирование подгонки, найденной с помощью optimization.curve_fit, имеет важность, поэтому мы можем построить график данных и подогнанной модели, чтобы убедиться, что она выглядит правильно.
❽ Показывает графики с данными и вписанными в новое окно.

Теперь, когда у нас есть главная программа, которую мы можем использовать для выполнения всей игры, мы видим, что Ланселот проделывает довольно разумную работу, выясняя, какой угол Дагонет спрятал в своей операции Q#. Проведя различные измерения и воспользовавшись классическими приемами обработки данных, Ланселот может оценить фазу, которую операция Дагонета применяет к его кубитам. Выполнение файла host.py должно сгенерировать два всплывающих окна, которые показывают графики вероятностей измерений в зависимости от масштаба Ланселота для двух стратегий, которые он может использовать (рис. 9.8). Первая – это подход, который мы уже очертили. Вторую мы имплементируем в последнем разделе главы.

ДЛЯ СПРАВКИ Поскольку пакеты экосистемы SciPy для выполнения подгонки не идеальны, иногда отыскиваемый параметр подгонки не является верным. Выполните алгоритм подгонки несколько раз, и, будем надеяться, в следующий раз он сработает лучше. Если у вас есть какие-либо вопросы о пакете построения графиков `matplotlib`, то рекомендуем ознакомиться с вот этими

книжными заголовками от издательства Manning: «Тренировочный центр по науке о данных» Леонарда Апельцина, глава 2, готовится к публикации в 2021 году (*Data Science Bootcamp*, Leonard Apeltsin), и «Наука о данных вместе с Python и Dask» Джесси К. Дэниела, главы 7 и 8 (*Data Science with Python and Dask*, Jesse C. Daniel, 2019).

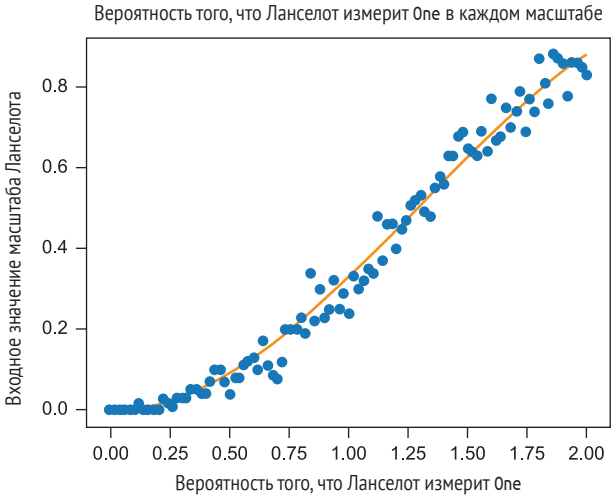


Рис. 9.8 Пример одного из двух графиков, которые должны появиться при выполнении файла host.py

Из этих графиков мы видим, что нам удалось довольно хорошо подогнать данные Ланселота. Это означает, что подогнанное значение, которое мы находим в `angle_guess`, является довольно хорошим приближением скрытого угла Дагонета!

Однако в стратегии Ланселота есть еще одна досаждающая проблема: всякий раз, когда он выполняет измерение, ему нужно подготавливать правильные входные данные для передачи в операцию Дагонета. В данной конкретной игре это может и не быть такой уж большой проблемой, но, учитывая то, что мы будем разведывать более широкие приложения этой игры в следующей главе, всякий раз подготавливать входной реестр в правильном состоянии будет обходиться дорого. К счастью, есть возможность применять *контролируемые операции* для многоразового использования одних и тех же входных данных, как мы увидим в остальной части этой главы.

Вы уже видели примеры контролируемых операций (например, `CNOT`), но оказывается, что многие другие квантовые операции тоже могут применяться условно, что бывает очень полезно. Контролируемые операции, наряду с последней новой концепцией квантовых вычислений, которая нам потребуется (собственные состояния – eigenstates), помогут нам в имплементировании технического приема, который мы встречали в конце главы 8: фазовой отдачи.

ДЛЯ СПРАВКИ В следующих далее нескольких разделах будет много рассуждений о локальной и глобальной фазах. Напомним, что глобальная фаза – это комплексный коэффициент, который может быть выведен (за скобки) из всех членов нашего состояния и не может наблюдаться. Если вам нужно освежить свои знания по фазам, то рекомендуем ознакомиться с главами 4–6.

9.4 Собственные состояния и локальные фазы

К настоящему времени мы уже узнали, что квантовая операция *X* позволяет нам переворачивать биты (|0 ⟩ ↔ |1⟩), а операция *Z* дает возможность переворачивать фазы (|+⟩ ↔ |−⟩). Однако обе эти операции применяют глобальные фазы только к некоторым входным состояниям. Как мы увидели в предыдущих главах, на самом деле мы ничего не можем узнать о глобальных фазах, поэтому понимание того, какие состояния каждая операция оставляет неизменными, имеет важность для понимания того, что можно узнать, применяя эту операцию.

Например, давайте вернемся к операции *Z*. В следующем ниже листинге мы увидим, что происходит, когда пытаемся использовать *Z* не для переворачивания кубита между состояниями |+⟩ и |−⟩, а для входного кубита в состоянии |0⟩.

Листинг 9.14 Применение Z к кубиту в состоянии |0⟩

```
use qubit = Qubit();                    ❶

Z(qubit);                               ❷

DumpRegister((), [qubit]);              ❸

Reset(qubit);                           ❹
```

❶ Как это принято в Q#, мы начинаем с выделения кубита с помощью инструкции use. Она поставляет свежий кубит в состоянии |0⟩.
❷ Применяет операцию Z, вследствие чего состояние кубита преобразовывается в $Z|0\rangle = |0\rangle$.
❸ В целях подтверждения того, что операция Z ничего не сделала, мы используем диагностическую функцию DumpRegister для указания симулятору распечатать полный вектор состояния.
❹ Сбрасывает кубит перед его высвобождением. В этом нет особой необходимости, поскольку мы знаем заранее, что кубит по-прежнему находится в состоянии |0⟩.

В листинге 9.14 мы можем подтвердить, что операция *Z* ничего не сделала, используя функцию `DumpRegister` для указания симулятору распечатать всю диагностическую информацию – в данном случае полный вектор состояния. На рис. 9.9 показано, как выглядит эта диагностическая распечатка.

ДЛЯ СПРАВКИ Если выполнить это на целевой машине, отличной от симулятора, то мы получим не вектор состояния, а любую

другую диагностику, которую машина предлагает (например, идентификаторы оборудования).

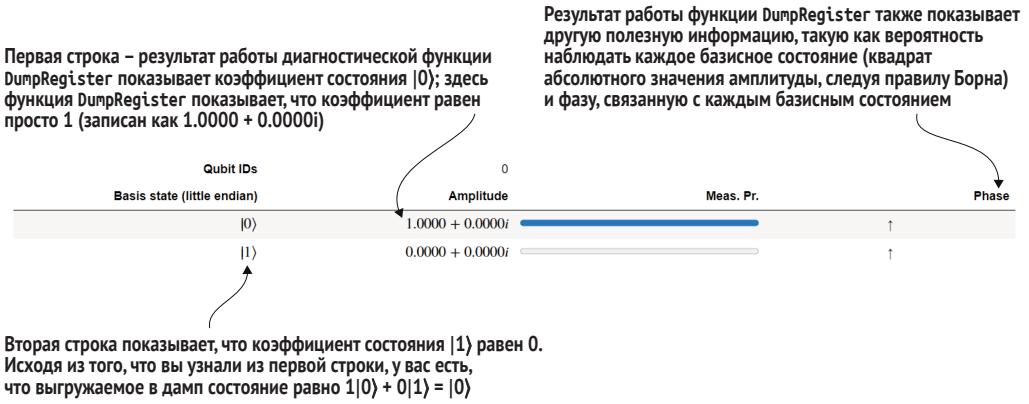


Рис. 9.9 Выходные данные выполняющегося листинга 9.14

Обратите внимание, что в листинге 9.14 `Z` ничего с кубитом не делает, так как $Z|0\rangle = |0\rangle$. Если мы модифицируем листинг, вместо этого подготовив $|1\rangle$, используя `X` перед `Z`, то мы увидим нечто очень похожее.

Листинг 9.15 Применение Z к кубиту в состоянии |1⟩

```
use qubit = Qubit();
X(qubit);                                  ❶

Z(qubit);                                  ❷

DumpRegister("1.txt", [qubit]);            ❹
Reset(qubit);
```

❶ Как и прежде, для подготовки состояния $|1\rangle$ мы можем использовать, что $|1\rangle = X|0\rangle$.
❷ Повторяет наш эксперимент выше, но с другим входным значением.
❸ Как и прежде, мы можем записать состояние кубита в текстовый файл, используя то, что мы выполняем на симуляторе, который сохраняет состояние внутри.

Результат выполнения выглядит следующим образом:

```
# волновая функция для кубитов с идентификаторами (от наименьшего до наиболее
значимого): 0
|0⟩:       0.000000 +  0.000000 i  ==                                  [ 0.000000 ]
|1⟩:      -1.000000 +  0.000000 i  ==     ********************         [ 1.000000 ]
➥ --- [ 3.14159 rad ]                     ❶
```

❶ Этот файл представляет вектор [[0], [−1]] или $-|1\rangle$ в нотации Дирака.

Эффект применения операции `Z` к состоянию $|1\rangle$ состоит в том, чтобы переворачивать знак состояния кубита. Это еще один пример *глобальной фазы*, как мы видели в главах 6 и 8.

Всякий раз, когда два состояния $|\psi\rangle$ и $|\varphi\rangle$ различаются только комплексным числом $e^{i\theta}$, $|\varphi\rangle = e^{i\theta}|\psi\rangle$, мы говорим, что $|\psi\rangle$ и $|\varphi\rangle$ варьируются по *глобальной фазе*. Например, $|0\rangle$ и $-|0\rangle$ отличаются глобальной фазой $-1 = e^{i\pi}$.

Глобальная фаза состояния не влияет на какие-либо вероятности измерения, поэтому мы никогда не сможем обнаружить, в каком состоянии, $|0\rangle$ либо $|1\rangle$, находилось входное значение операции `Z` при ее применении. Мы можем подтвердить это с помощью операции `AssertQubit`, которая проверяет вероятность конкретного результата измерения.

Листинг 9.16 Использование AssertQubit для проверки результата измерения

```
use qubit = Qubit();
AssertQubit(Zero, qubit);        ❶

Z(qubit);                        ❷

AssertQubit(Zero, qubit);        ❸

Message("Тест пройден!");        ❹

Reset(qubit)
```

❶ Проверяет, что измерение кубита возвращает результат Zero, и завершает программу, если это не так.
❷ После этого кубит находится в состоянии $-|0\rangle$, а не в состоянии $|0\rangle$. То есть операция Z применяет глобальную фазу к состоянию кубита.
❸ Снова вызывает AssertQubit, чтобы проверить, что вероятность получить результат Zero по-прежнему равна 1.
❹ Печатает сообщение, чтобы проверить, что квантовая программа успешно прошла оба подтверждения истинности.

Выполнение этого фрагмента кода просто напечатает `Тест пройден!`, поскольку в случае успешного выполнения подтверждения вызовы операции `AssertQubit` ничего не делают. Применение подобных подтверждений позволяет нам писать модульные тесты, в которых используются симуляторы, чтобы подтвердить наше понимание поведения конкретных квантовых программ. На реальном квантовом оборудовании по причине того, что мы не можем выполнять такого рода проверку из-за теоремы о запрете клонирования, подтверждения могут быть безопасно удалены.

ВАЖНО Подтверждения истинности `assert` бывают по-настоящему полезными инструментами для написания модульных тестов и проверки правильности наших квантовых программ. Тем не менее важно помнить, что они будут удалены при выполнении нашей программы на реальном квантовом оборудовании, поэтому мы не используем подтверждения истинности, чтобы обеспечивать правильную работу нашей программы.

Конечно же, использование указанных подтверждений также является просто хорошим практическим приемом программирования; подтверждения в классических языках, таких как Python, нередко бывают отключены по соображениям производительности, вследствие чего мы не можем полагаться на то, что подтверждения всегда будут присутствовать.

Выявление того, каким квантовым состояниям операция `U` назначает глобальные фазы, дает нам ключ к пониманию поведения этой квантовой операции. Мы называем такие состояния *собственными состояниями* операции `U`. Если две операции имеют одинаковые собственные состояния и применяют одни и те же глобальные фазы к каждому из этих собственных состояний, то отличить эти две операции друг от друга невозможно – подобно тому, как если бы две классические функции имели одну и ту же таблицу истинности, мы не можем сказать, какая из них какая, независимо от того, в каком состоянии находятся наши кубиты, когда мы применяем каждую операцию. Это означает, что мы можем понять операции не только по их матричному представлению, но и понимая, каковы их собственные состояния и какую глобальную фазу операция применяет к каждому из них. Как мы уже видели, мы не можем узнать глобальную фазу кубита непосредственно; поэтому в следующем далее разделе мы научимся использовать контролируемые версии операции для превращения этой глобальной фазы в локальную, которую можно измерить. Однако сейчас давайте подведем итог более формальным определением того, что такое собственное состояние.

Если после применения операции `U` состояние реестра кубитов qs модифицируется только глобальной фазой, то мы говорим, что состояние этого реестра является *собственным* состоянием `U`. Например, $|0\rangle$ и $|1\rangle$ являются собственными состояниями операции `Z`. Аналогичным образом $|+\rangle$ и $|-\rangle$ являются собственными состояниями `X`.

Попробуйте следующее ниже упражнение, чтобы попрактиковаться в работе с собственными состояниями.

Упражнение 9.3: диагностическая практика

Попробуйте написать программы Q#, в которых используются `AssertQubit` и `DumpMachine`, чтобы проверить следующее:

- $|+\rangle$ и $|-\rangle$ являются собственными состояниями операции `X`;
- $|0\rangle$ и $|1\rangle$ являются собственными состояниями операции `Rz`, независимо от того, на какой угол вы решаете повернуть.

В качестве дополнения попрактикуйтесь в выяснении собственных состояний операций `Y` и `CNOT` и напишите программу Q#, чтобы проверить свои догадки!

Подсказка: векторная форма собственных состояний унитарной операции может быть найдена с помощью пакета QuTiP. Например, собственные состояния операции `Y` задаются `qt.sigmay().eigenstates()`. Отталкиваясь от этого, вы можете использовать то, что узнали о поворотах в главах 4–6, чтобы определить, какие операции Q# подготавливают эти состояния.

Не забывайте, что вы всегда можете протестировать, что то или иное состояние является собственным состоянием операции, написав быстрый тест на Q#!

Собственные состояния являются очень полезной концепцией и используются в самых различных алгоритмах квантовых вычислений. Мы воспользуемся ими в следующем разделе вместе с контролируемыми операциями для имплементирования технического приема квантовой разработки под названием *фазовая отдача*, с которым мы познакомились в конце главы 7.

Глубокое погружение: это только в порядке вещей

Собственные состояния получили свое название от понятия, используемого во всей линейной алгебре, именуемого *собственными векторами*. Подобно тому, как собственное состояние – это состояние, которое оставляется неизменным квантовой операцией (т. е. самое большее применяется глобальная фаза), собственный вектор – это вектор, который сохраняется вплоть до масштабирующего фактора при умножении на матрицу. То есть если для матрицы A $A\vec{x} = \lambda\vec{x}$ для некоторого числа λ, то $\vec{x}$ является собственным вектором A. Мы говорим, что λ является соответствующим *собственным значением*.

Приставка «eigen», по-немецки означающая «собственный» или «характеристический», указывает на то, что собственные векторы и собственные значения помогают нам понимать свойства или характеристики матриц. В частности, если матрица A коммутирует с ее конъюгатной транспозицией (т. е. если $AA^\dagger = A^\dagger A$), то ее можно *разложить* на проекторы в собственные векторы, каждый из которых масштабируется своими собственными значениями:

$$A = \sum_i \lambda_i \vec{x}_i \vec{x}_i^\dagger.$$

Поскольку это условие всегда соблюдается для унитарных матриц, мы всегда можем понять квантовые операции, разложив их на собственные состояния и фазы, применяемые к каждому собственному состоянию. Например, $Z = |0\rangle\langle 0| - |1\rangle\langle 1|$ и $X = |+\rangle\langle +| - |-\rangle\langle -|$.

Одним из существенных следствий такого разложения матриц является то, что если две матрицы A и B имеют одинаковые собственные векторы и собственные значения, то они являются одной и той же матрицей. Аналогичным образом если две операции могут быть представлены одними и теми же собственными состояниями и собственными фазами, то эти операции неотличимы друг от друга.

Такой образ мышления о состояниях и операциях часто помогает нам разбираться в различных концепциях квантовых вычислений. На игру с оцениванием фазы, над которой вы работаете в этой главе, можно взглянуть и по-другому, как на алгоритм усвоения фаз, связанных с каждым собственным состоянием! В главе 10 вы увидите, что это особенно хорошо связано с определенными приложениями, такими как усвоение свойств химических систем.

9.5 Контролируемое применение: превращение глобальных фаз в локальные фазы

Из того, что мы увидели и можем проверить, *глобальные фазы* состояний ненаблюдаемы, в то время как локальные фазы состояний могут быть измерены. Например, возьмем состояние $1/\sqrt{2}\,(-i|0\rangle - i|1\rangle) = -i/\sqrt{2}\,(|0\rangle + |1\rangle)$. Не существует измерения, которое мы могли бы сделать, чтобы отличить это состояние от $(|0\rangle + |1\rangle)/\sqrt{2}$. Однако мы могли бы отличить любое из этих двух состояний от $(|0\rangle - |1\rangle)/\sqrt{2}$, поскольку оно отличается *локальной фазой*; т. е. одно из состояний имеет + перед $|1\rangle$, а другое –.

> **ДЛЯ СПРАВКИ** Если вы хотите освежить свои знания о фазах и о том, как думать о них как о поворотах, то рекомендуем вернуться к главам 4 и 5. При использовании симулятора в качестве целевой машины результаты работы диагностических функций `DumpMachine` и `DumpRegister` также помогут вам узнать о фазах состояний.

В предыдущем разделе мы немного поиграли с собственными состояниями и увидели, что глобальные фазы собственных состояний могут нести информацию об операции: назовем ее `U`. Если мы хотим узнать эту глобальную фазовую информацию о собственных состояниях, то, похоже, мы застряли. Если бы Ланселот подготавливал только собственные состояния операции Дагонета, то он никогда не смог бы узнать, под каким углом Дагонет спрятался.

Квантовые алгоритмы спешат на помощь! Существует очень полезный трюк, который мы можем применить, чтобы превратить глобальные фазы, применяемые операцией `U`, в локальные фазы, применяемые тесно связанной операцией. В целях охнакомления с тем, как это работает, давайте вернемся к операции `CNOT`. Вспомните из главы 6, что мы можем симулировать `CNOT`, используя унитарную матрицу:

$$U_{\mathrm{CNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Когда мы впервые столкнулись с этой матрицей в главе 6, мы использовали аналогию между унитарными матрицами и классическими таблицами истинности, чтобы выяснить, что операция `CNOT` меняет местами состояния $|10\rangle$ и $|11\rangle$, но оставляет кубиты в состояниях $|00\rangle$ и $|01\rangle$ нетронутыми. То есть `CNOT` переворачивает состояние своего второго кубита, *контролируемого* на состоянии первого кубита. Как показано на рис. 9.10, мы можем прочесть унитарную матрицу для операции `CNOT` как описание своего рода оператора «квантового если»: «*Если* контрольный кубит находится в состоянии $|1\rangle$, *то* применить операцию `X` к целевому кубиту».

Верхняя левая часть этой матрицы сообщает нам о том, что делает операция CNOT, когда контрольный кубит находится в состоянии $|0\rangle$

Начать с операции CNOT, которую мы встречали в главе 6. Мы можем представить ее унитарной матрицей. Поскольку операция CNOT действует на два кубита (контрольный и целевой кубиты), ее унитарная матрица представляет собой матрицу 4×4

$$U_{\text{CNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Нижняя правая часть сообщает нам о том, что происходит, когда контрольный кубит находится в состоянии $|1\rangle$. Обратите внимание, что эта часть матрицы для CNOT совпадает с матрицей для операции X

Мы можем написать унитарные матрицы для других контролируемых операций таким же образом. Например, операция контролируемого-Z (сокращенно CZ) представлена матрицей, в которой матрицы $\mathbb{1}$ и Z находятся на ее диагонали

$$U_{\text{CZ}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Рис. 9.10 Написание унитарных матриц для контролируемых операций

Для использования операции CNOT на языке Q# мы можем попробовать следующий ниже исходный код.

Листинг 9.17 Использование операции CNOT на языке Q#

```
use control = Qubit();
use target = Qubit();
H(control);                    ❶
X(target);                     ❷

CNOT(control, target);         ❸
DumpMachine();

Reset(control);
Reset(target);
```

❶ Подготавливает контрольный кубит в $|+\rangle$.
❷ Подготавливает целевой кубит в $|1\rangle$.
❸ Применяет CNOT и распечатывает состояние симулятора.

Думая о CNOT как о квантовом аналоге условной инструкции, мы можем написать ее унитарную матрицу немного прямолинейнее. В частности, мы можем рассматривать унитарную матрицу для операции CNOT как своего рода «блочную матрицу», которую можем построить, используя тензорное произведение, которое мы видели в главе 4:

$$U_{\text{CNOT}} = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & X \end{pmatrix} = |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes X.$$

Упражнение 9.4: проверка матрицы CNOT

Убедитесь, что $|0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes X$ совпадает с приведенным выше уравнением.

Подсказка: вы можете проверить это вручную, используя функцию `np.kron` пакета NumPy либо функцию `qt.tensor` пакета QuTiP. Если вам нужно освежить свои знания, то рекомендуем ознакомиться с симулированием телепортации в главе 6 или же посмотреть на результат работы алгоритма Дойча–Йожи в главе 8.

Мы можем построить другие операции по этому шаблону, такие как операция `CZ` (контролируемый-`Z`):

$$U_{CZ} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} = |00\rangle\langle 00| + |01\rangle\langle 10| + |10\rangle\langle 10| - |11\rangle\langle 11|.$$

Во многом подобно тому, как операция `CNOT` делает то же самое, что и операция `X` (применяет битовый переворот), но контролируется на состоянии другого кубита, когда его контрольный кубит находится в состоянии $|1\rangle$, операция `CZ` переворачивает фазу, как операция `Z`. На рис. 9.10 показан пример того, как это работает. Давайте посмотрим на то, как контролирование `Z` работает на практике, написав немного исходного кода на Q#, чтобы попробовать `CZ`.

Листинг 9.18 Тестирование операции CZ на языке Q#

```
use control = Qubit();
use target = Qubit();
H(control);                          ❶
X(target);                           ❷

CZ(control, target);                 ❸
DumpRegister("cz-output.txt", [control, target]);

Reset(control);
Reset(target);
```

❶ Подготавливает контрольный кубит в $|+\rangle$.
❷ Подготавливает целевой кубит в $|1\rangle$.
❸ Применяет `CZ` и сохраняет результирующее состояние.

Результат выглядит следующим образом:

```
# волновая функция для кубитов с идентификаторами (от наименьшего до наиболее
значимого): 0;1
|0⟩: 0.000000 + 0.000000 i  ==                                    [  0.000000  ]
|1⟩: 0.000000 + 0.000000 i  ==                                    [  0.000000  ]
```

```
|2⟩: 0.707107 + 0.000000 i  ==     **********                    [  0.500000  ]
➥ ---  [  0.00000 rad  ]
|3⟩: -0.707107 + 0.000000 i ==     **********                    [  0.500000  ]
➥---  [  3.14159 rad  ]
```

Если выполнить этот фрагмент кода, то `cz-output.txt` покажет, что окончательное состояние реестра `[control, target]` будет равно $U_{CZ}|{+}1\rangle = |{-}1\rangle$.

Упражнение 9.5: проверка данных на выходе из CZ

Вручную либо с помощью пакета QuTiP убедитесь, что приведенный выше результат совпадает с $|{-}1\rangle = |{-}\rangle \otimes |1\rangle$.

Если порядок кубитов поменялся местами, но, помимо этого, ответ – правильный, то обратите внимание, что в функции `DumpMachine` используется представление состояний *от младшего к старшему*[1]. В указанном представлении $|2\rangle$ является сокращением для $|01\rangle$, а не для $|10\rangle$. Если это выглядит озадачивающе, то вините архитектуру процессора x86.

То есть в зависимости от состояния *целевого кубита* состояние *контрольного кубита* в результате изменилось, подобно тому, как мы видели в главе 8 с алгоритмом Дойча–Йожи! Это обусловлено тем, что фаза, применяемая `Z` в случае, когда контрольный кубит находится в состоянии $|0\rangle$, не совпадает с той, когда контрольный кубит находится в состоянии $|1\rangle$. Указанный эффект называется *фазовой отдачей*. В главе 8 мы использовали фазовую отдачу с парой кубитов в состоянии $|{+}{-}\rangle$, чтобы определить факт применения операции `CNOT`. Здесь мы увидели, что можем использовать операцию `CZ`, чтобы узнать о глобальной фазе, применяемой операцией `Z`.

ВАЖНО Даже если $|1\rangle$ является собственным состоянием операции `Z`, $|{+}1\rangle$ *не является* собственным состоянием операции `CZ`. Это означает, что вызов `CZ` в реестре в состоянии $|{+}1\rangle$ имеет наблюдаемый эффект!

Фазовая отдача – это распространенный технический прием квантового программирования, поскольку он позволяет нам превращать то, что в противном случае было бы глобальными фазами, в фазу между ветвями $|0\rangle$ и $|1\rangle$ контрольного кубита. В примере `CZ` как входное состояние $|{+}\rangle|1\rangle$, так и выходное состояние $|{-}\rangle|1\rangle$ являются произведенными состояниями, что позволяет нам измерять контрольный кубит, не влияя на целевой кубит.

---

[1] Представление битов от младшего к старшему (или справа налево, little-endian representation) – это формат хранения и передачи двоичных данных, при котором сначала передается младший (наименее значимый) бит. – *Прим. перев.*

Думай глобально, учи фазы локально

Обратите внимание, что глобально-фазовая разница между $|1\rangle$ и $Z|1\rangle = -|1\rangle$ стала локально-фазовой разницей между $|1\rangle$ и $U_{CZ}|+1\rangle = |-1\rangle$. То есть, контролируя инструкцию Z на кубите в состоянии $|+\rangle$, мы смогли узнать, какой была бы глобальная фаза без контроля.

Используя операцию CZ, мы можем имплементировать технический прием фазовой отдачи, чтобы превратить глобальную фазу в локальную фазу, которую затем можно измерить.

Листинг 9.19 Использование CZ для имплементирования фазовой отдачи

```
use control = Qubit();
use target = Qubit();
H(control);                                          ❶
X(target);

CZ(control, target);                                 ❷
if (M(control) == One) { X(control); }               ❸

DumpRegister("cz-target-only.txt", [target]);        ❹

Reset(target);                                       ❺
```

❶ Подготавливает контрольный кубит в $|+\rangle$ и целевой кубит в $|1\rangle$.
❷ Применяет CZ и сохраняет полученное состояние. Однако, прежде чем мы выполним дамп состояния целевого кубита, давайте измерим и сбросим контрольный кубит.
❸ Забавный факт: на самом деле именно так имплементируется операция Reset на языке Q#.
❹ Теперь давайте выполним дамп состояния только цели.
❺ Мы уже сбросили управление, поэтому нам не нужно здесь сбрасывать его снова.

Вот результат:

```
# волновая функция для кубитов с идентификаторами (от наименее до наиболее
значимых): 1
|0⟩:      0.000000 +  0.000000 i  ==                                  [ 0.000000 ]
|1⟩:     -1.000000 +  0.000000 i  ==     ********************         [ 1.000000 ]
➥---       [  3.14159 rad ]          ❶
```

❶ Как и ожидалось, мы получаем, что целевой кубит остается в состоянии $|1\rangle$, готовый к подаче в другую операцию CZ.

9.5.1 Управление любой операцией

Возвращаясь к игре Ланселота и Дагонета, было бы очень полезно, если бы мы могли помочь Ланселоту повторно использовать кубит, который он передает в операцию Дагонета, чтобы ему не приходилось каждый раз заново его готовить. К счастью, использование контролируемых операций для имплементирования фазовой отдачи дает подсказку о том, как это можно сделать. В частности, когда мы использовали фа-

зовую отдачу в главах 7 и 8 для имплементирования алгоритма Дойча–Йожи, целевой кубит находился в состоянии |−⟩ как в начале, так и в конце алгоритма. Это означает, что Ланселот мог бы использовать один и тот же кубит повторно для каждого раунда своей игры и не нуждаться в повторной подготовке каждый раз. Это не имело значения для алгоритма Дойча–Йожи, так как мы провели только один раунд игры Нимуэ и Мерлина. Но это совершенно правильный трюк с точки зрения Ланселота, чтобы выиграть свою игру с Дагонетом, поэтому давайте посмотрим, каким образом можно ему помочь использовать фазовую отдачу.

Проблема в том, что хотя фазовая отдача является полезным инструментом для нашего инструментария как разработчика квантовых систем, до сих пор мы видели, как его использовать только с операциями `X` и `Z`. Мы знаем, что в нашей игре Дагонет сказал Ланселоту, что он будет использовать операцию `R1`; есть ли способ, которым мы можем использовать фазовую отдачу, чтобы здесь помочь? Шаблон, который мы использовали для имплементирования фазовой отдачи в предыдущем разделе, требовал от нас только управления операцией, так что нам нужен способ управления операцией `op::Apply`, которую Дагонет предоставляет Ланселоту. На языке Q# благодаря функтору `Controlled` это так же просто, как написать `Controlled op::Apply` вместо `op::Apply`. Подобно функтору `Adjoint` в главе 6, `Controlled` является ключевым словом языка Q#, которое изменяет поведение операции: в данном случае, чтобы превратить ее в контролируемую версию.

> **ДЛЯ СПРАВКИ** Так же, как `is Adj` указывает на то, что операция может быть использована с функтором `Adjoint`, `is Ctl` в типе операции указывает на то, что она может быть использована с функтором `Controlled`. Для обозначения поддержки операцией и того, и другого мы можем написать `is Adj + Ctl`. Например, тип операции `X (Qubit => Unit is Adj + Ctl)` дает нам знать, что *X* может быть одновременно адъюнктабельной и контролируемой.

Таким образом, чтобы помочь Ланселоту, мы можем поменять строку `op::Apply(scale, target)` на `Controlled op::Apply([control], (scale, target))`, и у нас получится контролируемая версия `R1`.

Хотя это действительно решает проблему Ланселота, бывает полезно еще распаковать немного то, что происходит под капотом. Любая унитарная операция (т. е. квантовая операция, которая не выделяет, не высвобождает и не измеряет кубиты) может контролироваться так же, как мы контролировали операцию `Z`, чтобы получить `CZ`, и как мы контролировали `X`, чтобы получить `CNOT`. Например, мы можем определить операцию контролируемого-контролируемого-`NOT` (`CCNOT`, также именуемую Тоффоли) как операцию, которая берет два контрольных кубита и переворачивает свою цель, если *оба* контрольных кубита находятся в состоянии |1⟩. В математическом плане мы пишем, что операция `CCNOT` превращает входное состояние |x⟩|y⟩|z⟩ в выходное |x⟩|y⟩|z

XOR (y AND z)⟩. Мы также можем написать матрицу, которая позволяет просимулировать операцию `CCNOT`:

$$U_{\text{CCNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Схожим образом операция контролируемого-SWAP (также именуемая операцией Фредкина) преобразовывает входное состояние из $|1\rangle|y\rangle|z\rangle$ в $|1\rangle|z\rangle|y\rangle$ и оставляет свое входное значение без изменений, когда первый кубит находится в состоянии $|0\rangle$.

ДЛЯ СПРАВКИ Мы можем сделать контролируемый-SWAP из трех операций `CCNOT`: `CCNOT(a, b, c); CCNOT(a, c, b); CCNOT(a, b, c);` является эквивалентным `Controlled SWAP([a], (b, c));`. Для того чтобы в этом убедиться, обратите внимание на то, что мы также можем сделать неконтролируемую операцию `SWAP` из трех операций `CNOT` по той же причине, по которой мы можем поменять местами два классических реестра, используя последовательность из трех классических операций XOR.

Мы можем обобщить этот шаблон для любой унитарной операции `U` (т. е. любой операции, которая не выделяет, не высвобождает или не измеряет свои кубиты). На языке Q# преобразование, выполненное с использованием функтора `Controlled`, добавляет новое входное значение в операцию, представляющую то, какие кубиты должны использоваться в качестве контрольных.

ДЛЯ СПРАВКИ Вот где очень пригодится тот факт, что Q# является языком с поддержкой механизма «кортеж вошел, кортеж вышел». Поскольку каждая операция принимает ровно одно входное значение, для любой операции `U` `Controlled U` берет изначальное входное значение в `U` в качестве своего второго входного значения.

Операции `CNOT` и `CZ` – это просто укороченная нотация для надлежащих вызовов `Controlled`. В табл. 9.1 приведены дополнительные примеры этого шаблона.

ДЛЯ СПРАВКИ Точно так же, как функтор `Adjoint` работает с любой операцией, имеющей в своем типе `is Adj` (как мы видели в главе 8), функтор `Controlled` работает с любой операцией, имеющей в своем типе `Ctl`.

Таблица 9.1 Несколько примеров контролируемых операций на языке Q#

Описание	Укороченная нотация	Определение
Контролируемый-NOT	`CNOT(control, target)`	`Controlled X([control], target)`
Контролируемый-контролируемый-NOT (Тоффоли)	`CCNOT(control0, control1, target)`	`Controlled X([control0, control1], target)`
Контролируемый-SWAP (Фредкин)	*n/a*	`Controlled SWAP([control], (target1, target2))`
Контролируемый-Y	`CY(control, target)`	`Controlled Y([control], target)`
Контролируемый-PHASE	`CZ(control, target)`	`Controlled Z([control], target)`

Как мы видели на примере `CZ`, управление операциями в таком ключе позволяет нам превращать глобальные фазы, такие как те, которые применяются к собственным состояниям, в относительные фазы, которые мы можем усваивать посредством измерений.

Больше того, используя контролируемые повороты для осуществления отдачи фазы в контрольный реестр, мы также можем использовать один и тот же целевой кубит многократно, снова и снова. Когда мы применили `CZ` к целевому реестру в собственном состоянии `Z`, этот целевой реестр оставался в том же состоянии, даже несмотря на изменение контрольного реестра. В остальной части данной главы мы увидим, как использовать этот факт, чтобы закончить стратегию Ланселота для его маленькой игры с Дагонетом.

9.6 Имплементирование лучшей стратегии Ланселота для игры с оцениванием фазы

Теперь у нас есть все необходимое для того, чтобы написать немного другую стратегию для Ланселота, которая позволит ему использовать контролируемые операции для многоразового использования одних и тех же кубитов. Как отмечалось ранее, это, возможно, не окажет большого влияния на игру Дагонета, но это имеет значение для других приложений квантовых вычислений.

Например, в главе 10 мы увидим, как задачи квантовой химии могут решаться с помощью игры, очень похожей на ту, в которую играют Дагонет и Ланселот. Там, однако, подготовка правильного входного состояния может потребовать вызова большого числа разных квантовых операций, вследствие чего, если мы сможем сохранять целевой кубит для последующего использования, мы сможем добиться довольно большого выигрыша в производительности.

Давайте кратко рассмотрим этапы игры.

1. Дагонет подбирает секретный угол для однокубитовой операции поворота.

2. Дагонет готовит операцию для использования Ланселотом, которая скрывает секретный угол и дает Ланселоту ввести одно дополнительное значение в форме числа (мы назовем его масштабом), которое будет умножено на секретный угол, чтобы получить общий угол операции поворота.
3. Наилучшая стратегия Ланселота в игре состоит в том, чтобы выбирать много масштабных значений и оценивать вероятность измерить `One` для каждого значения. Для этого ему нужно выполнить следующие ниже шаги много раз для каждого значения масштаба из множества:
   a. подготовить состояние $|+\rangle$ и ввести значение масштаба в поворот Дагонета. Он использует состояние $|+\rangle$, потому что знает, что Дагонет поворачивает вокруг оси Z; и для этого состояния его повороты приведут к изменению локальной фазы, которую он может измерить;
   b. после подготовки каждого состояния $|+\rangle$ Ланселот может поворачивать его с помощью секретной операции, измерять кубит и записывать измерение.
4. Теперь у Ланселота есть данные, касающиеся его масштабного фактора и вероятности того, что он измерил `One` для этого масштабного фактора. Он может выполнить подгонку этих данных в своей голове и получить угол Дагонета на основе подогнанных параметров (он и в самом деле – величайший рыцарь в стране). А мы задействуем Python, который поможет нам сделать то же самое!

Шаг, который необходимо изменить, чтобы применить наши новообретенные навыки с *контролируемыми* поворотами, является шагом 3. Для шага 3a *выделение* кубитов изменится. Вместо того чтобы выделять, подготавливать и измерять по одному кубиту на операцию измерения, Ланселот может выделить один целевой кубит для поворота с помощью черного ящика Дагонета и вместо этого выделять и измерять контрольные кубиты. Он по-прежнему может повторять измерения, но ему не придется всякий раз измерять или заново готовить цель.

Эти изменения можно подытожить, переписав шаг 3 следующим образом:

3. Лучшая стратегия Ланселота для игры состоит в том, чтобы выбрать много значений масштаба и оценивать вероятность измерить `One` для каждого значения. Для этого ему нужно выполнить следующие ниже шаги много раз для каждого значения масштаба из множества. Он готовит один кубит в состоянии $|1\rangle$ для использования в качестве цели для всех своих измерений, поскольку оно является собственным состоянием скрытого поворота:
   a. подготовить второй контрольный кубит в состоянии $|+\rangle$;
   b. применить новую контролируемую версию секретного поворота со значением масштаба Ланселота, откатить подготовку контрольного кубита и его измерить, а затем зарегистрировать измерение.

В нашем коде эти изменения могут быть выполнены путем модифицирования предыдущей операции `EstimateProbabilityAtScale`. Поскольку ось поворота может быть любой, которую выберет Дагонет (для удобства здесь это ось *Z*), Ланселот должен уметь контролировать произвольный поворот. Мы можем сделать это с помощью функтора `Controlled` перед вызовом операции `ScalableOperation`, переданной от Дагонета. Функтор `Controlled` очень похож на функтор `Adjoint` в том, что он принимает операцию и возвращает новую операцию. `Controlled U(control, target)` представляет собой пример синтаксиса, который позволяет нам применять `U` к целевому кубиту, контролируемому на одном или нескольких контрольных кубитах. В следующем ниже листинге показано, каким образом можно модифицировать `EstimateProbabilityAtScale` для использования функтора `Controlled`.

Листинг 9.20 operations.qs: новая стратегия Ланселота

```
operation EstimateProbabilityAtScaleUsingControlledRotations(
    target : Qubit,
    scale : Double,
    nMeasurements : Int,
    op : ScalableOperation)
: Double {
    mutable nOnes = 0;
    for idx in 0..nMeasurements - 1 {
        use control = Qubit();
        within {
            H(control);                          ❶
        } apply {
            Controlled op::Apply(                ❷
                [control],
                (scale, target)
            );
        }
            set nOnes += Meas.MResetZ(control) == One
                         ? 1 | 0;
    }
    return Convert.IntAsDouble(nOnes) /
           Convert.IntAsDouble(nMeasurements);
}
```

❶ **Операция угадывания теперь принимает на входе целевой реестр и использует его многократно. Отсюда нам нужно каждый раз выделять и подготавливать только контрольный реестр.**

❷ **Единственно, нам нужно сделать еще одно изменение – вызвать Controlled op::Apply вместо op::Apply, передавая новый контрольный кубит вместе с изначальными входными данными.**

Другая модификация, которую мы должны сделать (шаг 5), – это операция, которая выполняет игру. Поскольку использование контролируемой операции позволяет Ланселоту использовать целевой кубит многократно, его нужно выделить только один раз в начале игры. Взгляните на следующий ниже листинг с тем, как это можно имплементировать.

Листинг 9.21 operations.qs: имплементирование RunGameUsingControlledRotations

```
operation RunGameUsingControlledRotations(
    hiddenAngle : Double,
    scales : Double[],
    nMeasurementsPerScale : Int)
: Double[] {
    let hiddenRotation = HiddenRotation(hiddenAngle);
    use target = Qubit();                                    (1)
    X(target);                                               (2)
    let measurements = Arrays.ForEach(
        EstimateProbabilityAtScaleUsingControlledRotations(
            target, _, nMeasurementsPerScale, hiddenRotation
        ),
        scales
    );
    X(target);
    return measurements;
}
```

(1) Используя RunGameUsingControlledRotations, мы можем выделять целевой кубит один раз, так как мы используем его снова и снова в каждом угадывании.

(2) Используя операцию X, мы можем подготавливать цель в состоянии |1⟩, собственном состоянии (неконтролируемой) операции R1, в которой Дагонет скрыл свой угол.

Используя операцию `X`, как в листинге 9.21, мы можем подготавливать цель в состоянии |1⟩, собственном состоянии (неконтролируемой) операции `R1`, в которой Дагонет скрыл свой угол. Поскольку каждое измерение использует фазовую отдачу для воздействия только на контрольный реестр, эту подготовку можно выполнить один раз перед началом игры.

Резюме

- Фазовое оценивание – это квантовый алгоритм, который позволяет усваивать фазу, применяемую к реестру кубитов заданной операцией.
- На языке Q# можно объявлять новые пользовательские типы, обозначая то, как данный тип должен использоваться в квантовой программе, либо предоставляя укороченную нотацию для длинных типов.
- Квантовые программы на языке Q# могут выполняться самостоятельно или из главной программы, написанной на Python; это позволяет использовать программы на Q# наряду с инструментами обработки данных, такими как SciPy.
- Когда операция оставляет входные данные в заданном состоянии неизменными, помимо применения глобальной фазы, мы говорим, что это входное состояние является *собственным состоянием*, а соответствующая фаза – *собственной фазой*.

- Используя функтор `Controlled` и фазовую отдачу вместе, мы можем превращать глобальные собственные фазы в локальные фазы, которые можно наблюдать и оценивать.
- Собрав все вместе, мы можем использовать классические приемы подгонки данных для усвоения собственных фаз из измерений, возвращаемых в результате выполнения программы Q#, которая выполняет оценивание фазы.

Часть II: заключение

В этой части книги мы получили массу удовольствия, используя язык Q# и квантовые вычисления, чтобы помочь различным обитателям Камелота. Применяя квантовый генератор случайных чисел, написанный на Q#, мы смогли помочь Моргане одурачить бедного Ланселота. В то же время мы помогали Мерлину и Нимуэ играть свои роли в решении судеб королей, все это время изучая алгоритм Дойча–Йожи и фазовую отдачу. Когда на земле воцарился мир, а костры в замке Камелот горели всю ночь, мы увидели, как использовать все, чему мы научились, чтобы помочь Ланселоту сыграть в еще одну игру, выиграв на этот раз благодаря угадыванию скрытой Дагонетом квантовой операции.

Во время наших эскапад в Камелоте вы узнали довольно много новых трюков, которые помогут вам на вашем пути в качестве квантового разработчика:

- что такое квантовый алгоритм и как его имплементировать с помощью Комплекта инструментов для квантовой разработки и языка Q#;
- как использовать язык Q# из Python и среды блокнотов Jupyter Notebook;
- как создавать оракулы для представления классических функций в квантовых программах;
- пользовательские типы;
- контролируемые операции;
- фазовую отдачу.

В будущем пришло время вернуть то, чему вы научились в Камелоте, домой и применить эти новые технические приемы к чему-то более практичному. В следующей далее главе вы увидите, как квантовые вычисления помогают в понимании задач химии. Не волнуйтесь, если вы не помните периодическую таблицу; вы будете работать с несколькими коллегами, которые знают химическую сторону вещей и ищут вашей помощи в использовании всего того, что вы узнали в этой части книги, с целью модернизирования своего рабочего потока с помощью квантовых технологий.

Часть III

Прикладные квантовые вычисления

К этому месту в книге мы создали великолепный инструментарий квантово-алгоритмических приемов – и в этой части мы узнаем, как применять эти технические приемы к разным практическим задачам. В частности, мы имплементируем и выполним небольшие примеры трех разных квантовых программ, каждая из которых обращается к другой области, в которой могут применяться квантовые вычисления. Эти примеры столь малы, что мы можем их просимулировать с помощью классических компьютеров, но они демонстрируют то, как квантовые устройства могут обеспечивать вычислительные преимущества в задачах, представляющих практический интерес.

В главе 10 мы будем использовать наши навыки квантового программирования для имплементирования квантового алгоритма, который помогает решать сложные задачи химии. Мы будем опираться на это в главе 11, чтобы имплементировать алгоритм поиска по неструктурированным данным; мы научимся применять функциональные возможности, встроенные в язык Q# и QDK, для оценивания ресурсов, необходимых для выполнения квантового приложения в масштабе. Наконец, в главе 12 мы имплементируем алгоритм Шора для факторизации целых чисел, возможно, один из самых известных квантовых алгоритмов благодаря его применению в классической криптографии.

10 Решение химических задач с помощью квантовых компьютеров

Эта глава охватывает следующие ниже темы:

- решение химических симуляций с помощью квантовых компьютеров;
- имплементирование операции Exp и метода Троттера–Сузуки;
- создание программ для оценивания фазы, декомпозиции и т. д.

В главе 9 мы использовали ряд новых функциональных средств языка Q#, таких как определяемые пользователем типы (UDT) и выполнение программ из хостов Python для оказания помощи в написании квантовой программы, которая может оценивать фазы. Как мы увидим в этой главе, оценивание фазы обычно используется в квантовых алгоритмах для строительства более крупных и сложных программ. В этой главе мы рассмотрим нашу первую область практического применения: химию.

10.1 Реальные химические приложения для квантовых вычислений

До этого места в книге мы научились использовать квантовые устройства для всего, начиная с общения с нашей подругой Евой и заканчивая

помощью в решении судеб королей. Однако в данной главе у нас будет возможность сделать что-то более *практичное*.

ПРИМЕЧАНИЕ Теперь, когда у нас есть все необходимое для решения более сложных задач с помощью квантовых компьютеров, сценарий этой главы будет немного сложнее, чем большинство наших предыдущих игр и сценариев. Не волнуйтесь, если что-то будет не совсем понятно с места в карьер. Не торопитесь и читайте медленнее; мы обещаем, что это будет стоить вашего времени!

Как оказалось, наша подруга – квантовый химик Мария достигла предела того, что может сделать ее классический компьютер, чтобы помогать ей симулировать разные химические системы. Задачи, которые Мария решает с помощью технических приемов вычислительной химии, помогают бороться с изменением климата, понимать новые материалы и совершенствовать использование энергии в различных отраслях промышленности; если мы сможем помочь ей с использованием языка Q#, то это может иметь довольно много практических применений. К счастью, используя то, что мы узнали об оценивании фазы в главе 9, именно это мы можем и сделать, так что давайте приступим!

Более качественное тестирование посредством химии

Любой производитель конфет может рассказать вам о важности температуры: сварите сахар до «вязкой» стадии, и получите ириски; но если добавить немного больше энергии, то можно сделать любое число других восхитительных конфет, начиная от ирисок и заканчивая карамелью. Все в сахаре – его вкус, внешний вид и его привлекательность – меняется в зависимости от энергии, которую мы вливаем в него через кастрюлю. В немалой степени, если понять, как меняется форма молекул сахара при добавлении энергии в сладкий плавильный котел, то мы поймем сам сахар.

Мы видим этот эффект не только с конфетами, но и на протяжении всей нашей жизни. Вода, пар и лед различаются, если понимать, какие формы может принимать H_2O – в какие формы он может перестраиваться – в зависимости от энергии. Во многих случаях мы хотим понять, как молекула перестраивается в зависимости от энергии, основываясь не на экспериментах, а на симуляциях. В этой главе мы будем опираться на технические приемы из предыдущих нескольких глав, показывая, каким образом можно симулировать энергию химических систем, чтобы понимать их так же остро, как производитель конфет понимает свое ремесло, и использовать эти химические системы, чтобы делать нашу жизнь лучше – возможно, даже немного слаще.

В целях понимания того, как это работает, и тут мы согласны с Марией, мы начнем с рассмотрения *молекулярного водорода*, или H_2, поскольку это достаточно простая химическая система, мы можем сравнивать то, что усваиваем из нашей квантовой программы, с тем, что могут симулировать классические инструменты моделирования. Благодаря этому, по-

скольку мы применяем одни и те же технические приемы для изучения молекул, которые крупнее, чем можно просимулировать классически, у нас есть отличный тестовый случай, к которому мы можем прибегнуть, чтобы убедиться, что все правильно.

Симуляция внутри симуляции

В этой главе наша работа с Марией включает в себя два разных вида симуляции: использование классического компьютера для симулирования квантового компьютера и использование квантового компьютера для симулирования квантовой системы другого типа. Мы часто хотим делать и то, и другое, поскольку при строительстве приложений квантовой химии полезно использовать классический компьютер для симулирования того, как квантовый компьютер симулирует квантово-химическую систему. Благодаря этому при выполнении нашей квантовой симуляции на реальном квантовом оборудовании мы можем быть уверены в том, что она работает правильно.

Как показано на рис. 10.1, Мария начнет с того, что, используя свой опыт в квантовой химии, опишет задачу, которую она заинтересована решить с помощью квантового компьютера: в данном случае понимание структуры H_2. По большей части эти задачи состоят в усвоении свойств особого вида матрицы, именуемой *гамильтонианой*. Получив от Марии гамильтониану, мы сможем написать квантовую операцию, очень похожую на ту, что показана в листинге 10.1, чтобы ее просимулировать и узнать о ней то, что Мария может использовать, чтобы понять поведение разных химических веществ. На протяжении остальной части этой главы мы будем развивать концепции и понимание того, что нам необходимо выполнить шаги, показанные на рис. 10.1.

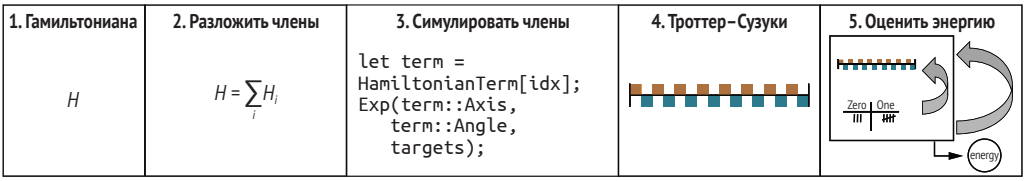


Рис. 10.1 Обзор шагов, которые мы разработаем в этой главе, чтобы помочь Марии усвоить энергетический образ основного состояния ее молекулы

В этой главе мы выполним имплементацию следующих ниже шагов для нашего алгоритма гамильтонианової симуляции:

1. сотрудничать с Марией, чтобы выяснить, какая гамильтониана описывает уровни энергии в интересующей ее системе и аппроксимацию основного (или самого низкого энергетического) состояния;
2. подготовить эту аппроксимацию основного состояния и использовать операцию `Exp` языка Q# с целью имплементирования эволюции квантовой системы для каждого члена гамильтонианы;

3. используя разложение Троттера–Сузуки, имплементированное в функции Q# `DecomposedIntoTimeStepsCA`, просимулировать эволюцию нашей системы под действием всех членов гамильтонианы сразу путем разбивки эволюции на небольшие шаги;
4. после симулирования эволюции системы в условиях гамильтонианы использовать фазовое оценивание, чтобы узнать об изменении в фазе нашего квантового устройства;
5. внести окончательную поправку в фазу, которую мы оцениваем для системы, после чего у нас будет энергия основного состояния для H_2.

В следующем ниже листинге показано, как эти шаги транслируются в исходный код.

Листинг 10.1 Исходный код Q#, который оценивает энергию основного состояния H_2

```
operation EstimateH2Energy(idxBondLength : Int) : Double {
    let nQubits = 2;
    let trotterStepSize = 1.0;
    let trotterStep = EvolveUnderHamiltonian(idxBondLength,
        trotterStepSize, _);
    let estPhase = EstimateEnergy(nQubits,
        PrepareInitalState,
        trotterStep,
        RobustPhaseEstimation(6, _, _));
    return estPhase / trotterStepSize + H2IdentityCoeff(idxBondLength);
}
```

Без дальнейших церемоний давайте погрузимся в первую квантовую концепцию, которая нам нужна, чтобы помочь Марии: энергия.

10.2 Много путей ведут к квантовой механике

До сих пор мы изучали квантовую механику, используя язык вычислений: биты, кубиты, инструкции, устройства, функции и операции. Однако образ мышления Марии о квантовой механике сильно отличается (рис. 10.2). Для нее квантовая механика – это физическая теория, которая говорит ей о поведении субатомных частиц, таких как электроны. Рассуждая в терминах физики и химии, квантовая механика – это теория о *веществе*, из которого состоит все вокруг нас.

Эти два образа мышления встречаются, когда приходит время симулировать поведение физических систем, таких как молекулы. Мы можем использовать квантовые компьютеры для симулирования того, как другие квантовые системы эволюционируют и изменяются с течением времени. То есть квантовые вычисления касаются не только физики или химии; они также помогают нам понимать научные задачи, подобные тем, с которыми сталкивается Мария.

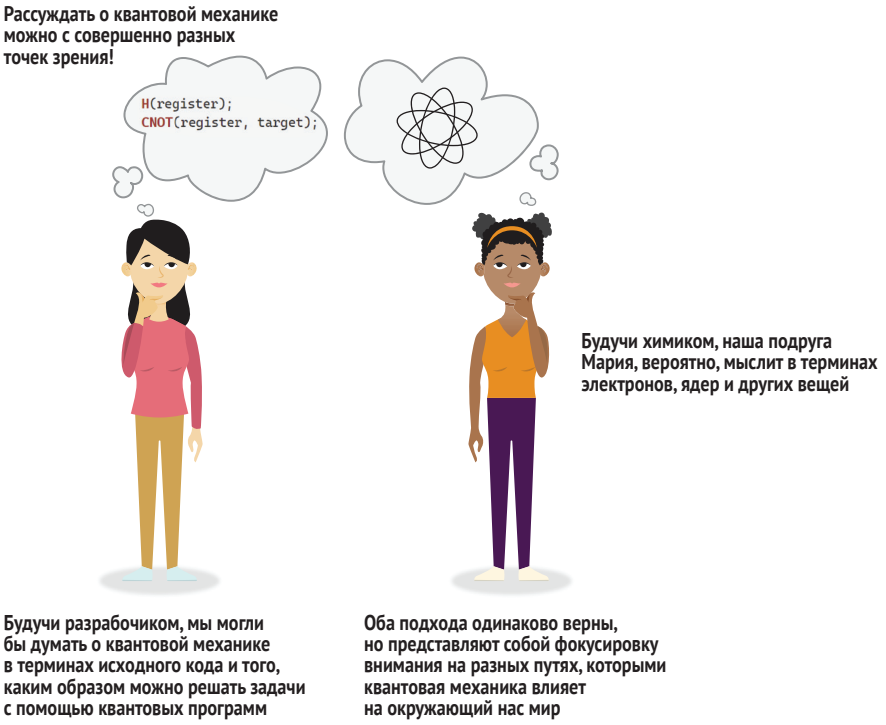


Рис. 10.2 Размышление о квантовой механике на основе двух совершенно разных подходов

В сердцевине того, как мы рассуждаем о квантовых вычислениях, лежит информация, но с точки зрения физики и химии квантовая механика в значительной степени опирается на концепцию *энергии*. Энергия говорит нам о том, как физические системы, столь разнообразные, как мячи и компассы, подвержены влиянию окружающего мира, давая нам согласованный способ понимания каждой из этих разных систем. На рис. 10.3 мы видим, как состояние мяча на холме и состояние компаса можно описать одинаково, используя концепцию энергии.

Как оказалось, энергия применима не только к классическим системам, таким как мячи и компасы. И действительно, мы можем понять, как *квантовые системы*, такие как электроны и ядра, себя ведут, понимая энергию разных конфигураций. В квантовой механике энергия описывается особым видом матрицы, именуемой гамильтонианой. Любая матрица, являющаяся своим собственным адъюнктом, может быть использована в качестве гамильтонианы, и гамильтонианы сами по себе *не являются* операциями.

Вспомните из глав 8 и 9, что *адъюнкт* матрицы A является ее конъюгатной транспозицией, $A^\dagger$. Эта концепция тесно связана с ключевым словом `Adjoint` в языке Q#: если операция op может быть просимулирована унитарной матрицей U, то операция `Adjoint op` может быть просимулирована $U^\dagger$.

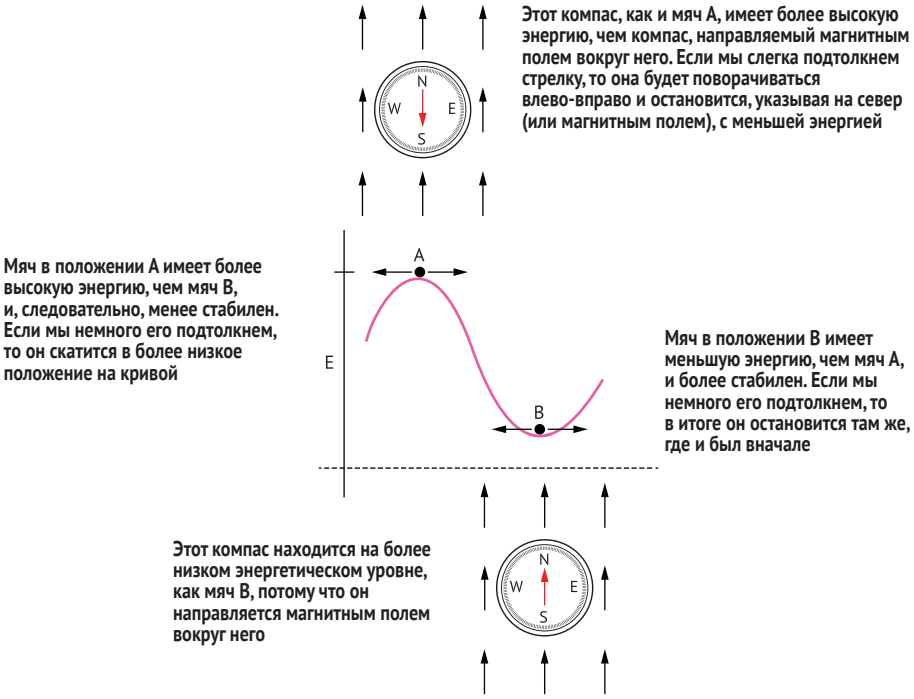


Рис. 10.3 Использование энергии для понимания влияния окружающей среды на разные физические системы. Мяч на вершине холма и компас, указывающий на юг, являются примерами систем с более высокой энергией, чем мяч в долине или компас, указывающий на север

В этой главе мы изучим все инструменты и технические приемы для определения энергий квантовых систем, для которых у нас есть гамильтониана. Часто процесс получения гамильтониан для систем выливается в сотрудничество, но после того, как мы получим их и еще несколько фрагментов информации, мы сможем оценить энергию этой системы. Указанный процесс называется *гамильтониановой симуляцией* и имеет решающее значение для массы разных применений квантовых вычислений, включая химию.

ДЛЯ СПРАВКИ Мы встречали несколько примеров гамильтониан в предыдущих главах: все матрицы Паули (X, Y и Z) являются примерами гамильтониан, помимо того что являются унитарными матрицами. Однако не все унитарные матрицы можно использовать в качестве гамильтониан! Большинство примеров в этой главе требуют дополнительной работы, прежде чем мы сможем применить их в качестве квантовых операций.

Мария заинтересована в понимании энергии связей в ее химических веществах. Поэтому имеет смысл придумать гамильтониану, описывающую ее молекулу; тогда мы сможем помочь оценить интересующую

ее энергию. В химии эта энергия часто именуется *энергией основного состояния*, а соответствующее состояние известно как *основное состояние* (или состояние минимальной энергии).

Получив гамильтониану, следующий шаг будет состоять в том, чтобы выяснить, как построить операции, которые будут симулировать характер изменения квантовой системы во времени, как описывается гамильтонианой. В следующем далее разделе мы научимся описывать эволюцию квантовой системы в условиях гамильтонианы.

Затем, имея в руках операторы, представляющие гамильтониану, следующая задача состоит в том, чтобы выяснить, как симулировать гамильтонианы на нашем квантовом устройстве. Вероятно, в физическое устройство не будет встроена только одна операция, которая будет делать именно то, что нам нужно, поэтому мы должны найти способ разложить наши операции для нашей гамильтонианы с точки зрения того, что наше устройство может обеспечить. В разделе 10.4 мы узнаем, каким образом можно принимать любые операции и выражать их в терминах операций Паули, которые обычно доступны в виде аппаратных инструкций.

Выразив нашу гамильтониану в виде суммы матриц Паули, каким образом мы будем симулировать их все в нашей системе? Вероятно, у нас будет несколько членов, которые в сумме представляют действие гамильтонианы, и они не обязательно будут коммутировать. В разделе 10.6 мы научимся использовать метод Троттера–Сузуки, применяя понемногу каждого члена в операции, чтобы просимулировать эволюцию в условиях всего этого одним махом. И тогда наша квантовая система пройдет эволюционный процесс способом, который представляет гамильтониану нашей коллеги, Марии!

Наконец, в целях выявления энергии системы, описываемой найденной нами гамильтонианой, мы сможем применить фазовое оценивание, чтобы помочь Марии. В разделе 10.7 мы будем использовать алгоритм, с которым познакомились в главе 9, для усвоения фазы, применяемой к нашим кубитам, путем симулирования гамильтонианы. Давайте приступим к делу!

10.3 Использование гамильтониан для описания эволюции квантовых систем во времени

На рис. 10.4 показан трекер шагов по симулированию еще одной квантовой системы с помощью нашего квантового компьютера. В целях использования гамильтонианы для описания энергии физической или химической системы нам нужно посмотреть на ее собственные состояния и их собственные значения.

Вспомните из главы 9, что если состояние $|\psi\rangle$ является собственным состоянием операции op, то применение op к реестру в состоянии $|\psi\rangle$ в лучшем случае применяет глобальную фазу к $|\psi\rangle$. Эта фаза называется *собственным значением*, или *собственной фазой*, соответствующей

этому собственному состоянию. Как и все другие глобальные фазы, эта собственная фаза не может наблюдаться непосредственно, но мы можем использовать функтор `Controlled`, о котором узнали в главе 9, чтобы превратить эту фазу в локальную фазу.

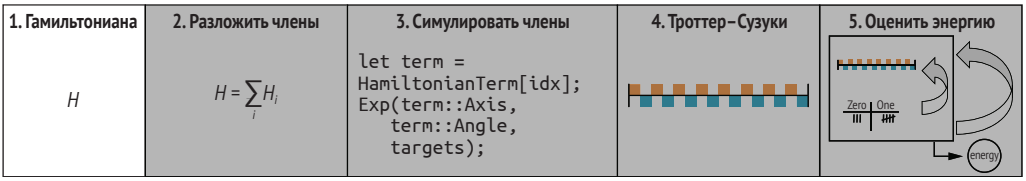


Рис. 10.4 Мы начинаем здесь с изучения молекулы H_2 Марии и того, какая гамильтониана описывает ее эволюцию

Каждое собственное состояние гамильтонианы является состоянием постоянной энергии; точно так же, как квантовые операции ничего не делают с собственными состояниями, система, находящаяся в собственном состоянии своей гамильтонианы, будет оставаться в этой энергии с течением времени. Другое свойство собственных состояний, которое мы увидели в главе 9, по-прежнему соблюдается и здесь: фаза каждого собственного состояния эволюционирует во времени.

Наблюдение, что фазы собственных состояний эволюционируют во времени, является содержанием уравнения Шредингера, одного из самых важных уравнений во всей квантовой физике. Уравнение Шредингера говорит нам о том, что по мере эволюционирования квантовой системы каждое собственное состояние гамильтонианы накапливает фазу, пропорциональную ее энергии. Используя математику, мы можем записать уравнение Шредингера, как показано на рис. 10.5.

Настоящие эксперты – это друзья, которых мы завели на этом пути

Как могло случиться, что это уже 10-я глава, а мы впервые видим самое важное уравнение в квантовой физике? Разработка квантовых приложений бывает тесно связана с квантовой физикой, но это не одно и то же, и нам не нужно быть экспертом в физике, чтобы писать квантовые приложения, – мы можем быть, если нам интересно, но нам это не обязательно. Уравнение Шредингера появляется здесь только потому, что оно нам нужно, чтобы понять, как квантовые компьютеры могут использоваться для практического воздействия.

Точно так же, как наша подруга Мария является экспертом в квантовой химии, а не в квантовых вычислениях, нам не нужно знать все, чтобы сделать что-то потрясающее. Вот для чего нужны друзья!

ДЛЯ СПРАВКИ Уравнение Шредингера связывает то, как фазы разных состояний эволюционируют во времени, с энергией этих состояний. Поскольку глобальные фазы ненаблюдаемы, а соб-

ственные состояния гамильтониан приобретают глобальные фазы только по мере их эволюционирования, уравнение Шредингера говорит нам о том, что собственные состояния гамильтониан не эволюционируют во времени.

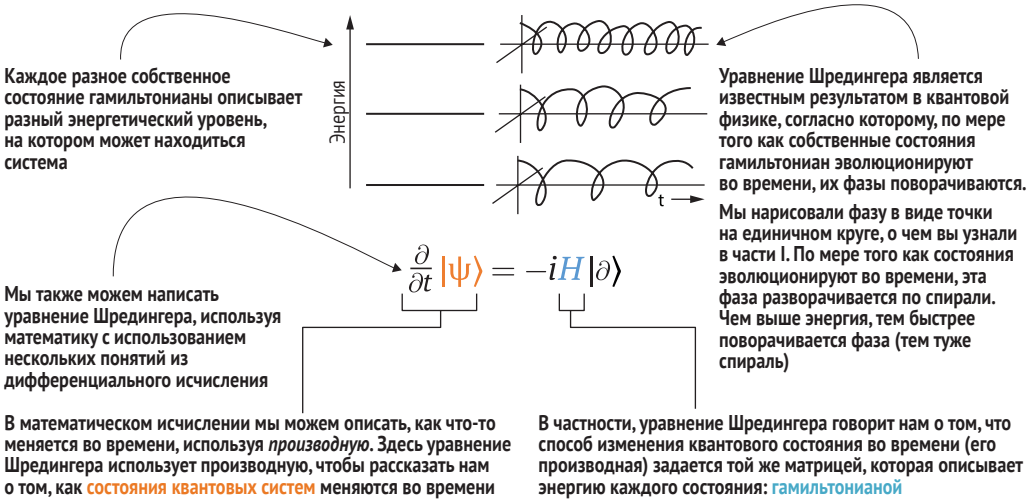


Рис. 10.5 Уравнение Шредингера, записанное в математической нотации

Уравнение Шредингера имеет для нас наибольшую важность в этой главе, потому что оно связывает энергию системы с фазой. Это очень полезная связь, учитывая, что в главе 9 мы узнали, как проводить фазовое оценивание! Существуют и другие способы использования уравнения Шредингера, один из которых представляет собой еще один способ взглянуть на имплементирование операций на квантовых системах.

Один из вариантов имплементировать повороты, которые мы встречали в книге до этого, состоит в том, чтобы установить правильную гамильтониану, а затем подождать. Производная по времени ($\delta/\delta t$) в уравнении Шредингера говорит о том, что способ поворота наших кубитов полностью описывается энергией, связанной с каждым состоянием. Например, уравнение Шредингера говорит о том, что если наша гамильтониана равна $H = \omega Z$ для некоторого числа ω, то если мы хотим повернуть на угол θ вокруг оси Z, то можем позволить нашему кубиту эволюционировать за время $t = \theta/\omega$.

Упражнение 10.1: повороты вокруг

Попробуйте написать другие повороты, показанные ранее в книге (например, Rx и Ry), как гамильтонианы.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

В листинге 10.2 показана простая операция языка Q#, которая симулирует эволюционирование в условиях гамильтонианы $H = \omega Z$.

ПРИМЕЧАНИЕ На практике бóльшая часть проблемы создания квантового компьютера заключается в обеспечении того, чтобы кубиты эволюционировали только в соответствии с инструкциями квантовой программы. Было бы не очень полезно, если бы оставление нашего квантового устройства на мгновение означало бы, что все наши кубиты окажутся в совершенно разных состояниях, когда мы вернемся. Это одна из причин, почему, будучи разработчиками квантовых систем, мы склонны думать на уровне посылаемых устройству инструкций, – т. е. квантовых операций, – а не непосредственно в терминах гамильтониан.

Переключая на мгновение передачу и временно мысля в терминах гамильтониан, мы получаем немного того, что нам нужно, чтобы начать продвигаться вперед в решении задачи, с которой Мария попросила нас помочь. В конце концов, задачи, с которыми работает Мария, гораздо легче описывать на этом языке. Например, в главе 9 мы узнали, каким образом можно усваивать фазу, применяемую поворотами, подобными тому, который Дагонет скрыл от Ланселота. Однако мы также можем выразить игру Дагонета и Ланселота в терминах гамильтониан. Предположим, что Дагонет скрыл угол поворота 2.1π; тогда, поскольку его поворот был вокруг оси Z, мы могли бы описать этот скрытый поворот и как скрытую гамильтониану $H = -2.1\pi Z$.

ПРИМЕЧАНИЕ Нам нужен знак (из-за знака) в уравнении Шредингера. Неправильное понимание этого вопроса встречается в квантовом программировании так же часто, как и ошибки в других языках, поэтому не переживайте, если вы раз или два забудете или если вы почти каждый раз получаете этот надоедливый знак минус. Вы по-прежнему отлично справляетесь.

Используя такое описание, масштаб Ланселота соответствует продолжительности, с которой он позволяет своим кубитам эволюционировать в условиях скрытой гамильтонианы Дагонета. В то время как мышление в терминах игры облегчает написание квантовых программ для усвоения скрытого поворота Дагонета, мышление в терминах гамильтонианы облегчает соотнесение с физическими понятиями, с которыми имеет дело

Мария, такими как напряженность поля и время. Единственная сложность заключается в том, что нам нужно умножать угол для Rz на 2.0, так как Rz умножает свой угол на –1/2 по соглашению в Q#. Поскольку уравнение Шредингера говорит нам о том, что угол должен иметь знак минус, 2.0 дает нам угол, который нам нужен, чтобы соответствовать рис. 10.5.

Листинг 10.2 Эволюционирование в условиях гамильтонианы H = ωZ

```
operation EvolveUnderZ(                          ❶
    strength : Double,                           ❷
    time : Double,                               ❸
    target : Qubit
) : Unit is Adj + Ctl {
    Rz(2.0 * strength * time, target);           ❹
}
```

❶ **Операция, симулирующая эволюцию, в условиях *H* = ω*Z* с использованием Q#.**
❷ **ω говорит о величине описываемых гамильтонианой энергий. Это играет роль скрытого угла Дагонета из главы 9!**
❸ **Какова продолжительность, с которой мы хотим симулировать гамильтониану. Это аналогично масштабу Ланселота из главы 9.**
❹ **Фактическая симуляция выражается всего одной строкой кода, так как повороты вокруг оси *Z* встроены в Q#.**

Поскольку уравнение Шредингера говорит нам о том, что эволюционирующие гамильтонианы поворачивают квантовые системы в соответствии с их энергией, если мы сможем *просимулировать* гамильтониану, которую предоставляет нам Мария, то сможем сыграть точно в такую же игру с оцениванием фазы, что и в главе 9, чтобы узнать энергетические уровни этой гамильтонианы.

Глубокое погружение: гамильтонианы – это штука, которую я могу контролировать

Когда мы впервые вводили квантовые операции, такие как H, X и Z, вы, возможно, задавались вопросом, каким образом мы будем имплементировать их на реальном квантовом устройстве. Используя концепцию гамильтониан, мы можем вернуться к этому вопросу и разведать, каким образом внутренние квантовые операции работают на аппаратном обеспечении.

При приложении магнитного поля к физической системе с магнитным диполем (например, спином электрона) гамильтониана для этой системы включает член, описывающий характер взаимодействия системы с магнитным полем. Обычно мы записываем этот член как $H = \gamma BZ$, где B – это сила указанного магнитного поля, а γ – число, которое описывает силу, с которой эта система реагирует на магнитные поля. Таким образом, чтобы применить поворот Rz в квантовом оборудовании, в котором для имплементирования кубитов используются электронные спины, мы можем активировать магнитное поле и просто подождать нужный промежуток времени. Аналогичные эффекты могут использоваться для имплементирования других гамильтонианових членов или управления гамильтонианой для других квантовых устройств.

Тот же принцип используется и в других квантовых технологиях, таких как ядерно-магнитный резонанс (ЯМР), где были разработаны хорошие классические алгоритмы для построения эффективных гамильтониан путем *пульсирования* магнитных полей с нужной частотой или создания импульсов сложной формы для применения квантовой операции. Традиционно в ЯМР и в квантовых вычислениях в более общем случае алгоритмам дизайна импульсов (pulse-design) даются причудливые аббревиатуры в виде таких названий, как GRAPE, CRAB, D-MORPH и даже ACRONYM. Однако независимо от прихоти эти алгоритмы позволяют нам использовать классические компьютеры для дизайна квантовых операций при наличии контрольных гамильтониан, таких как $H = \gamma BZ$. Если вы заинтересованы в том, чтобы узнать больше, то в оригинальной статье по GRAPE изложено много сведений о теории оптимального управления, которая используется с тех пор[1].

На практике, конечно же, это не будет полной историей. Мало того, что дизайн контрольных импульсов требует гораздо большего, но и для отказоустойчивых квантовых компьютеров внутренние операции, с которыми мы работаем как квантовые разработчики, не соотносятся с физическими операциями напрямую так же, как на ближнесрочном оборудовании. Вернее, эти низкоуровневые аппаратные операции используются для построения исправляющих ошибки кодов, таких что одна внутренняя операция может раскладываться на много разных импульсов, применяемых на все наше устройство.

Предположим, что вместо $H = \omega Z$ Мария спрашивает о том, сможем ли мы просимулировать $H = \omega X$. К счастью, язык Q# также обеспечивает поворот вокруг оси X, поэтому мы можем модифицировать вызов `Rz` в листинге 10.2 вызовом `Rx`. К сожалению, не каждая гамильтониана, которой интересуется Мария, так проста, как $H = \omega Z$ или $H = \omega X$, поэтому давайте посмотрим, какие технические приемы квантовой разработки мы можем использовать для симулирования гамильтониан, которые чуть-чуть посложнее.

Это не те гамильтонианы, которые мы ищем

Вполне вероятно, что когда мы начнем разговаривать с Марией, она также будет работать над описанием гамильтониан для своей системы в своей симуляции и программном обеспечении для моделирования. Однако, вероятно, это будут *фермионные гамильтонианы*, которые отличаются от тех, которые мы используем здесь для описания характера изменений квантовых устройств во времени. В рамках рабочего потока нашего сотрудничества

[1] *Навин Канейджа и соавт.* Оптимальное управление сопряженной динамикой спина: дизайн последовательностей импульсов ЯМР посредством алгоритмов градиентного подъема (Navin Khaneja et al. *Optimal Control of Coupled Spin Dynamics: Design of NMR Pulse Sequences by Gradient Ascent Algorithms*, Journal of Magnetic Resonance 172, no. 2 (2005): 296) // https://www.sciencedirect.com/science/article/abs/pii/S1090780704003696.

с Марией нам, вероятно, потребуется использовать несколько инструментов, таких как NWChem (https://nwchemgit.github.io), для конвертирования между гамильтонианой, которая описывает характер изменения химического вещества во времени, и характером изменения кубитов во времени. В этой книге нет возможности рассматривать данные методы подробно, но есть отличные программные средства, которые в этом помогут. Если вы заинтересованы, то рекомендуем ознакомиться с документацией Комплекта инструментов для квантовой разработки, чтобы получить подробную информацию: https://docs.microsoft.com/azure/quantum/user-guide/libraries/chemistry/. На данный момент это не имеет большого значения, просто удобный совет на случай, когда вы будете разговаривать со своими сотрудниками!

10.4 Поворачивание вокруг произвольных осей с помощью операций Паули

Двигаясь вверх по сложности, возможно, Мария интересуется чем-то, для описания чего требуется больше, чем однокубитовая гамильтониана (рис. 10.6). Если она дает нам гамильтониану типа $H = \omega X \otimes X$, то что можно сделать, чтобы ее просимулировать? К счастью, то, что мы узнали о поворотах в части I этой книги, по-прежнему полезно для такого рода двухкубитовой гамильтонианы, поскольку мы можем это рассматривать как описание еще одного вида поворота.

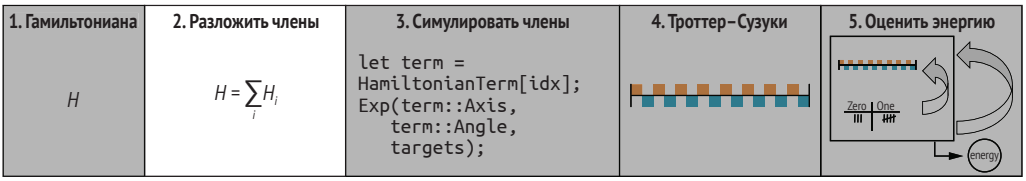


Рис. 10.6 На шаге 2 мы рассмотрим разложение нашей гамильтонианы на обобщенные повороты, которые легче симулировать

ПРИМЕЧАНИЕ В предыдущем разделе мы увидели, что повороты, такие как `Rx`, `Ry` и `Rz`, соответствуют гамильтонианам, таким как X, Y и Z, в указанном порядке. Двухкубитовые гамильтонианы, такие как $X \otimes X$, можно рассматривать как указывающие ось почти таким же образом. Оказывается, в отличие от 3 размерностей, которые мы получаем для однокубитового реестра, существует 15 возможных ортогональных осей поворота для двухкубитового реестра. Поэтому, чтобы нарисовать что-либо в виде картинки, нам понадобится на 13 размерностей больше, чем обычно имеет бумага, что несколько затрудняет иллюстрирование!

Этот поворот не похож ни на одну из встроенных (т. е. внутренних) инструкций, которые мы встречали до сих пор, поэтому может показаться, что мы застряли. Однако, как оказалось, мы все же можем просимулировать эту гамильтониану, используя *однокубитовые* повороты, такие как `Rx`, при условии что мы используем какие-то двухкубитовые операции с обеих сторон. В данном разделе мы увидим, как это работает и как Q# позволяет легко автоматизировать применение многокубитовых поворотов.

В порядке начала работы давайте рассмотрим несколько способов, которыми мы можем изменить то, что делают квантовые операции, окружив их другими операциями. Мы всегда можем использовать математику для рассуждений о решаемой задаче, как мы видели в главе 9, но, к счастью, Q# также предоставляет несколько хороших функций и операций тестирования, которые нам помогут. Например, в главе 9 мы узнали, что окружение операции `CNOT` операциями `H` дает нам `CNOT`, идущую в другом направлении. Давайте посмотрим, каким образом это можно проверить с помощью Q#!

> **ДЛЯ СПРАВКИ** Напомним, что блок `within/apply` в листинге 10.3 применяет принцип «ботинок и носков», о котором мы впервые узнали в главе 9. В большинстве примеров исходного кода в этом разделе используются блоки `within/apply`, помогая отслеживать наше мышление в стиле ботинок и носков.

Листинг 10.3 Изменение управления и цели операции CNOT

```
open Microsoft.Quantum.Diagnostics;                          (1)

operation ApplyCNOT(register : Qubit[])
: Unit is Adj + Ctl {                                        (2)
    CNOT(register[0], register[1]);
}

operation ApplyCNOTTheOtherWay(register : Qubit[])
: Unit is Adj + Ctl {                                        (3)
    within {
        ApplyToEachCA(H, register);
    } apply {
        CNOT(register[1], register[0]);
    }
}

operation CheckThatThisWorks() : Unit {
    AssertOperationsEqualReferenced(2,
        ApplyCNOT, ApplyCNOTTheOtherWay);                    (4)
    Message("У-у-у!");                                       (5)
}
```

(1) **Операции и функции в пространстве имен Microsoft.Quantum.Diagnostics помогают в тестировании и отладке квантовых программ и бывают очень полезны в обеспечении надлежащей работы программ.**

❷ В целях сравнения двух способов записи операции CNOT нам нужно, чтобы каждый из них был вызываем как операция, которая принимает массив кубитов, представляющих квантовый реестр.

❸ В целях проверки эквивалентности, которую мы впервые увидели в главе 7, мы можем написать вторую операцию, которая изменяет управление и цель операции CNOT.

❹ Первое входное значение задает величину реестра, на который воздействует каждая операция, а второе и третье входные значения представляют сравниваемые операции. Если операции делают что-то другое, то подтверждение не выполняется, и квантовая программа завершается.

❺ Если мы видим сообщение «У-у-у!», то мы можем с уверенностью заключить, что, посмотрев на то, что две операции делают с состояниями квантовых реестров, отличить их друг от друга невозможно.

ПРИМЕЧАНИЕ Такие подтверждения, как `AssertOperationsEqualReferenced`, имеют смысл только при выполнении на симуляторе, так как их выполнение требует нарушения теоремы о запрете клонирования. На фактическом оборудовании подобные подтверждения были бы удалены, подобно тому, как запуск Python с аргументом `-O` в командной строке отключает ключевое слово `assert`. Это означает, что подтверждения Q# дают нам возможность *безопасно* жульничать, поскольку квантовые программы, использующие инструкции подтверждения, делают то же самое независимо от того, жульничаем мы или нет.

Упражнение 10.2: проверка идентичностей операции CNOT

Примените пакет QuTiP, чтобы убедиться, что две операции `ApplyCNOT` и `ApplyCNOTTheOtherWay` могут быть просимулированы одной и той же унитарной матрицей и, следовательно, делать одно и то же.

Упражнение 10.3: три операции CNOT дают операцию SWAP

Подобно тому, как мы используем три классические инструкции XOR для имплементирования классического обмена (свопа) прямо на месте, мы можем применить три операции `CNOT`, чтобы сделать то же самое, что и одна операция `SWAP`. Следующий ниже фрагмент кода Q# делает то же самое, что и `SWAP(left, right)`:

```
CNOT(left, right);
CNOT(right, left);
CNOT(left, right);
```

Перепроверьте, что это то же самое, что и `SWAP(left, right)`, используя `AssertOperationsEqualReferenced` и пакет QuTiP.

Дополнительный кредит: `SWAP(left, right)` – это то же самое, что и `SWAP(right, left)`, поэтому приведенный выше фрагмент кода должен работать, даже если мы начнем с `CNOT(right, left)`. Перепроверьте!

Глубокое погружение: изоморфизм Чоя–Джамилковского

Операция `AssertOperationsEqualReferenced` в листинге 10.3 работает с использованием изящного математического элемента, именуемого *изоморфизмом Чоя–Джамилковского*, который говорит о том, что любая операция, которую можно просимулировать с помощью унитарной матрицы, идеально эквивалентна частичному состоянию, именуемому ее состоянием Чоя. Это означает, что симулятор может эффективно найти всю таблицу истинности для любой адъюнктабельной операции (т. е. любой операции, которая имеет в своей сигнатуре `is Adj`), найдя ее состояние Чоя. В операции `AssertOperationsEqualReferenced` эта концепция используется для подготовки реестра кубитов в состоянии Чоя для каждой операции, передаваемой на вход. На симуляторе легко сжульничать и проверить, что два состояния являются одинаковыми, хотя теорема о запрете клонирования говорит нам о том, что сделать это на реальном устройстве невозможно.

При написании модульных тестов и других проверок правильности квантовых программ он служит мощным техническим приемом при использовании классических симуляторов, в то же время предотвращая обман на реальном оборудовании.

При выполнении `CheckThatThisWorks` в среде блокнотов Jupyter Notebook (как мы видели в главе 7) либо в командной строке мы должны увидеть сообщение «У-у-у!», говорящее нам о том, что наша программа Q# успешно прошла вызов операции `AssertOperationsEqualReferenced`. Поскольку это подтверждение истинности проходит только в том случае, если две операции, которые мы ей передаем, делают одно и то же для всех возможных входных значений, то мы знаем, что эквивалентность, о которой мы узнали в главе 7, работает.

Мы можем использовать ту же логику, чтобы проверить то, как двухкубитовые операции, такие как `CNOT`, преобразовывают другие операции. Например, преобразование вызова `X` в вызовы `CNOT` делает то же самое, что и многократный вызов `X`, как показано в следующем ниже листинге.

Листинг 10.4 Применение X к каждому кубиту в реестре

```
open Microsoft.Quantum.Diagnostics;
open Microsoft.Quantum.Arrays;

operation ApplyXUsingCNOTs(register : Qubit[])
: Unit is Adj + Ctl {                                  ❶
    within {
        ApplyToEachCA(                                 ❷

            CNOT(register[0], _),                      ❸

            Rest(register)                             ❹
        );
    } apply {
        X(register[0]);                                ❺
```

```
    }
}

operation CheckThatThisWorks() : Unit {
    AssertOperationsEqualReferenced(2,
        ApplyXUsingCNOTs,
        ApplyToEachCA(X, _)                          ❻
    );
    Message("У-у-у!");
}
```

❶ Операция, представляющая собой одиночный вызов X с операциями CNOT, используя блок within/apply.
❷ Для части блока within/apply, касающейся «носков», мы можем записать нужные нам вызовы CNOT, используя ApplyToEachCA вместе с техническим приемом частичного применения, о котором мы узнали в главе 7.
❸ Эта часть нашего вызова ApplyToEachCA говорит о применении операции CNOT, контролируемой на первом кубите реестра к каждому элементу массива кубитов.
❹ Использует Rest, чтобы выбрать все, кроме первого (т. е. 0-го) элемента реестрового массива.
❺ Часть блока within/apply, касающаяся «ботинок», немного проще: просто операция X на том же кубите, который мы использовали в качестве контрольного для нашей последовательности вызовов CNOT.
❻ На этот раз вместо написания нашей собственной операции, с которой нужно проводить сравнение, мы сравниваем операцию X с каждым кубитом в реестре с помощью частичного применения.

Упражнение 10.4: унитарная эквивалентность

Используя пакет QuTiP, убедитесь, что при выполнении на двухкубитовых реестрах две программы в листинге 10.4 могут быть просимулированы одной и той же унитарной матрицей и, таким образом, делать то же самое со своими входными реестрами.

Упражнение 10.5: эквивалентность программ

Попробуйте изменить листинг 10.4, чтобы проверить эквивалентность обеих программ при применении к более чем двум кубитам.

Примечание: использование `AssertOperationsEqualReferenced` для более чем нескольких кубитов может обходиться довольно дорого.

Используя концепцию `within/apply`, можно строить и другие интересные виды операций. В частности, преобразование поворота с помощью операций `CNOT` таким же образом, как в листинге 10.4, позволяет нам имплементировать виды многокубитовых поворотов, которые Мария просила в начале этого раздела. Используя диагностические функции `DumpMachine` и `DumpRegister`, о которых мы узнали в главе 9, мы можем увидеть, что подобно тому, как `Rx` применяет X-осевой поворот между $|0\rangle$ и $|1\rangle$, мы можем имплементировать X-осевой поворот ($X \otimes X$) между $|00\rangle$ и $|11\rangle$.

Листинг 10.5 Создание многокубитовой операции Rx

```
open Microsoft.Quantum.Diagnostics;
open Microsoft.Quantum.Math;

operation ApplyRotationAboutXX(
    angle : Double, register : Qubit[]
) : Unit is Adj + Ctl {
    within {
        CNOT(register[0], register[1]);          ❶
    } apply {
        Rx(angle, register[0]);                  ❷
    }
}

operation DumpXXRotation() : Unit {
    let angle = PI() / 2.0;
    use register = Qubit[2];                     ❸

    ApplyRotationAboutXX(angle, register);       ❹

    DumpMachine();                               ❺

    ResetAll(register);                          ❻
}
```

❶ **Для простоты в этом листинге мы конкретизировали до двухкубитового случая, но мы можем использовать вызов ApplyToEachCA таким же образом для работы с реестрами из более двух кубитов.**

❷ **Вместо применения операции X к контрольному кубиту мы хотим применить поворот X вокруг произвольного угла к контрольному кубиту.**

❸ **В целях проверки того, что делает наша новая операция ApplyRotationAboutXX, мы начинаем с запроса у нашей целевой машины двухкубитового реестра с помощью инструкции «using».**

❹ **Затем мы применяем новый поворот вокруг оси ($X \otimes X$) к новому реестру, чтобы увидеть, что он делает.**

❺ **При выполнении на симуляторе диагностическая функция DumpMachine выдает полное состояние симулятора, позволяющее нам проверить, как новая операция поворота изменила состояние реестра.**

❻ **Как обычно, прежде чем выпустить реестр обратно на целевую машину, нам нужно сбросить все кубиты обратно в состояние $|0\rangle$.**

Упражнение 10.6: предсказывание операции ApplyRotationAboutXX

Попробуйте подготовить реестр в состояниях, отличных от $|00\rangle$, перед вызовом операции `ApplyRotationAboutXX`. Делает ли указанная операция то, что вы ожидали?

Подсказка: вспомните из части I книги, что мы можем подготавливать копию состояния $|1\rangle$, применяя операцию `X`, и что мы можем подготавливать $|+\rangle$, применяя операцию `H`.

Результат выполнения листинга 10.5 может немного отличаться от рис. 10.6, поскольку ядро IQ# для среды блокнотов Jupyter Notebook поддерживает несколько разных способов обозначения кубитовых состояний.

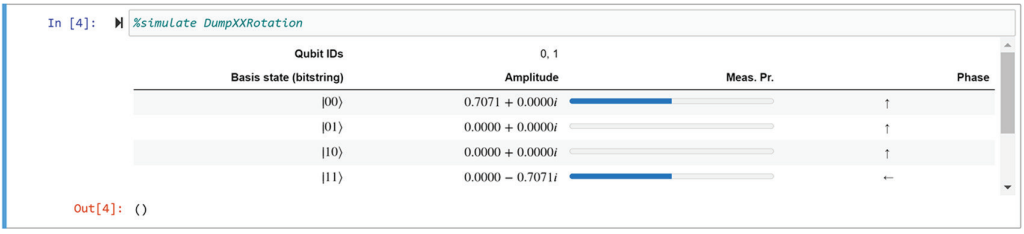


Рис. 10.7 Результат выполнения листинга 10.5 в среде блокнотов Jupyter Notebook. Из результата работы диагностической функции DumpMachine в DumpXXRotation мы видим, что наше результирующее состояние равно суперпозиции между |00⟩ и |11⟩

ДЛЯ СПРАВКИ По умолчанию IQ# использует представление от младшего к старшему (little-endian), полезное для арифметических задач, подобных тем, которые мы увидим в главе 12. Для обозначения кубитовых состояний с использованием битовых строк, подобных тем, которые мы видели в книге до этого, выполните `%config dump.basisStateLabelingConvention = "Bitstring"` из новой ячейки блокнота Jupyter Notebook.

Упражнение 10.7: поворот Rx v. вокруг $X \otimes X$

Попробуйте применить `DumpMachine`, чтобы разведать действие операции `Rx` на одном кубите, и сравните с поворотом двух кубитов вокруг оси ($X \otimes X$), который мы имплементировали в листинге 10.5. Чем похожи эти две операции поворота? и чем они отличаются? Сравните поворачивание вокруг оси ($X \otimes X$) с применением операции `Rx` к каждому кубиту в двухкубитовом реестре.

В общем случае *любой* поворот вокруг оси, заданный тензорным произведением матриц Паули (например, $X \otimes X$, $Y \otimes Z$ или $Z \otimes Z \otimes Z$), может быть имплементирован путем применения однокубитового поворота, преобразованного последовательностью операций, таких как `CNOT` и `H`. Однако поиск правильного преобразования бывает немного надоедливым, поэтому Q# предлагает в помощь хорошую встроенную операцию под названием `Exp`.

Листинг 10.6 Использование Exp для определения способа преобразования состояния

```
open Microsoft.Quantum.Diagnostics;
open Microsoft.Quantum.Math;

operation ApplyRotationAboutXX(
```

```
    angle : Double, register : Qubit[]
) : Unit is Adj + Ctl {
    within {
        CNOT(register[0], register[1]);
    } apply {
        Rx(angle, register[0]);
    }
}

operation CheckThatThisWorks() : Unit {
    let angle = PI() / 3.0;
    AssertOperationsEqualReferenced(2,
        ApplyRotationAboutXX(angle, _),
        Exp([PauliX, PauliX], -angle / 2.0, _)
    );
    Message("У-у-у!");
}
```

ПРЕДУПРЕЖДЕНИЕ Соглашения, используемые `Exp` и `Rx` для обозначения углов, отличаются в $-1/2$ раза. При использовании операции `Exp` и операций однокубитового поворота в одной и той же программе обязательно перепроверьте все ваши углы!

Используя `Exp`, легко просимулировать гамильтониану $H = \omega X \otimes X$ или любую другую гамильтониану, состоящую из тензорных произведений матриц Паули (рис. 10.8). Как показано в следующем ниже листинге, в языке Q# можно задавать $(X \otimes X)$ значением Q# `[PauliX, PauliX]`.

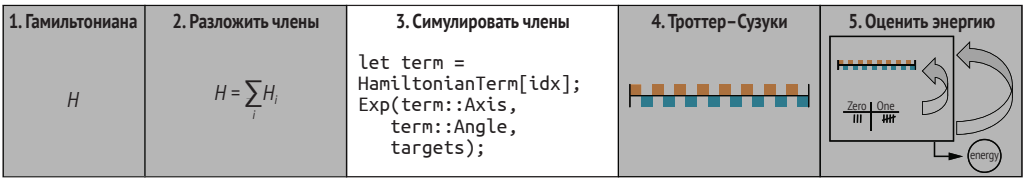


Рис. 10.8 На шаге 3 мы рассмотрим способ использования операции `Exp` для программирования обобщенных поворотов, представляющих гамильтониану, которую мы пытаемся просимулировать

Листинг 10.7 Использование `Exp` для симулирования эволюции при $X \otimes X$

```
operation EvolveUnderXX(                                    (1)
    strength : Double,                                      (2)
    time : Double,                                          (3)
    target : Qubit
) : Unit is Adj + Ctl {
    Exp([PauliX, PauliX], strength * time, target);         (4)
}
```

(1) Используя то, что мы узнали до этого, мы можем написать операцию для симулирования эволюции в условиях гамильтонианы, пропорциональной $(X \otimes X)$, подобно тому, как операция, которую

мы написали в листинге 10.2, симулировала эволюцию в условиях гамильтонианы, пропорциональной *Z*.

❷ Параметр, представляющий силу гамильтонианы: величина энергий, описываемых нашей гамильтонианой.

❸ Параметр, описывающий длительность симулирования эволюции (аналогично параметру масштаба Ланселота из главы 9).

❹ Запрашивает поворот вокруг оси ($X \otimes X$) с использованием операции Exp, предоставляемой пространством имен Microsoft.Quantum.Intrinsic.

$Z \otimes Z$ – это не просто два поворота Z

Может возникнуть соблазн подумать, что мы можем имплементировать двухкубитовый поворот вокруг $Z \otimes Z$, повернув первый кубит вокруг Z, а затем повернув второй кубит вокруг Z. Однако эти операции оказываются совершенно разными:

$$R_z(\theta) \otimes R_z(\theta) = \begin{pmatrix} e^{-i\theta} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{pmatrix}; \quad R_{zz}(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 & 0 & 0 \\ 0 & e^{i\theta/2} & 0 & 0 \\ 0 & 0 & e^{i\theta/2} & 0 \\ 0 & 0 & 0 & e^{-i\theta/2} \end{pmatrix}.$$

Как вариант на это можно взглянуть как на то, что поворот вокруг $Z \otimes Z$ чувствителен только к *паритету* каждого вычислительно-базисного состояния, поэтому $|00\rangle$ и $|11\rangle$ поворачиваются на одинаковую фазу.

Теперь, когда в нашем распоряжении есть операция Exp, ее довольно легко использовать для написания операции, симулирующей каждый член гамильтонианы, которую нам предоставила Мария.

Листинг 10.8 operations.qs: симулирование эволюции одного члена

```
operation EvolveUnderHamiltonianTerm(
    idxBondLength : Int,                          ❶
    idxTerm : Int,                                ❷
    stepSize : Double,                            ❸
qubits : Qubit[])
: Unit is Adj + Ctl {
    let (pauliString, idxQubits) =
        H2Terms(idxTerm);                         ❹
    let coeff =
        (H2Coeff(idxBondLength))[idxTerm];        ❺
    let op = Exp(pauliString,
        stepSize * coeff, _);                     ❻
    (RestrictedToSubregisterCA(op, idxQubits))
        (qubits);                                 ❼
}
```

❶ Индекс для обращения к гамильтониане, которую нам предоставила Мария. Каждая из них соответствует разной длине связи.

❷ **Член этой гамильтонианы, в условиях которой мы хотим симулировать эволюцию.**
❸ **Продолжительность симулирования эволюции: т. е. сколько времени займет шаг симуляции.**
❹ **Получает член из гамильтонианы, используя idxTerm вместе с функцией H2Terms, предлагаемой в репозитории исходного кода этой книги.**
❺ **Получает коэффициент этого члена, используя функцию H2Coeff, также предлагаемую в репозитории образцов, прилагаемых к этой книге.**
❻ **Симулирует эволюцию в условиях этого члена, используя Exp для выполнения поворота, масштабируемого на размер шага эволюции, точно так же, как это делала операция EvolveUnderXX из листинга 10.7.**
❼ **Поскольку не все члены влияют на все кубиты, мы можем использовать операцию RestrictedToSubregisterCA, предлагаемую языком Q#, чтобы применить вызов Exp только к подмножеству входных значений.**

В следующем далее разделе мы научимся использовать это для симулирования эволюции в условиях полной гамильтонианы Марии.

10.5 Внесение изменений, которые мы хотим видеть в системе

Теперь, когда мы научились описывать характер изменения квантового устройства во времени, используя концепцию гамильтонианы, возникает вполне естественный вопрос о том, как имплементировать конкретную гамильтониану, которую мы хотим просимулировать. Большинство квантовых устройств имеют операции, которые им легко выполнять. Например, в предыдущем разделе мы увидели, что эволюция легко симулируется в условиях любой гамильтонианы, заданной тензорным произведением матриц Паули. Тем не менее гамильтониана, которая нас (и Марию) интересует, скорее всего, не является встроенной операцией. Пожалуй, она является чем-то, что не доступно непосредственно на нашем квантовом компьютере.

> **ДЛЯ СПРАВКИ** Обычно в устройствах легко имплементируются несколько операторов Паули и, возможно, несколько других операций. Затем игра превращается в выяснение того, каким образом преобразовывать нужную нам операцию в операции, которые устройство легко может выполнять.

Если нет простой кнопки для симулирования эволюции в условиях нашей гамильтонианы, то каким образом можно имплементировать симуляцию конкретной гамильтонианы, которую можно применять к кубитам в нашем устройстве?

Давайте разберемся детально. Буквально еще в главе 2 мы узнали, что вектор можно описывать как линейную комбинацию *базисных векторов* или направлений. Оказывается, мы можем делать то же самое с матрицами, и действительно удобным базисом для этого являются операторы Паули.

Освежение памяти в отношении матриц Паули

Если вам нужно освежить свою память по вопросу, что такое матрицы Паули, то будьте спокойны, мы вас прикроем:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}; \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Подобно тому, как мы можем описать любое направление на карте севером и западом, мы можем описать любую матрицу как линейную комбинацию матриц Паули. Например:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \frac{1}{2}1 + \frac{1}{2}Z.$$

Схожим образом

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} = \frac{1}{2}(7\,1 + 7X - iY - 3Z).$$

То же самое соблюдается и для матриц, воздействующих на несколько кубитов:

$$U_{\mathrm{SWAP}} = \frac{1}{2}(1 \otimes \mathbb{1} + X \otimes X + Y \otimes Y + Z \otimes Z).$$

Упражнение 10.8: проверка идентичности

Примените пакет QuTiP для проверки приведенных выше уравнений.

Подсказка: вы можете применить `qt.qeye(2)`, чтобы получить копию 1, `qt.sigmax()`, чтобы получить копию X, и т. д. Для вычисления тензорных произведений, таких как $X \otimes X$, можно применить `qt.tensor`.

Это хорошая новость, потому что тогда мы можем написать гамильтониану, которую хотим просимулировать как линейную комбинацию матриц Паули. В предыдущем разделе мы узнали, что мы можем использовать Exp для легкого симулирования гамильтониан, состоящих только из тензорных произведений матриц Паули. Это делает базис Паули очень удобным, так как, скорее всего, рабочий поток из химических инструментов Марии уже будет выдавать гамильтонианы для нашего квантового устройства в базисе Паули.

Давайте посмотрим на представление гамильтонианы, которую Мария хочет, чтобы мы просимулировали, используя базис Паули для ее разложения. Задействуя свои навыки химического моделирования, Мария может любезно нам подсказать, что гамильтониана, которую нам нужно просимулировать с помощью кубитов, задается приведенным

ниже уравнением, где каждое из a, b_0, ... и b_4 – это действительное число, которое зависит от длины связи, при которой она хочет симулировать H_2:

$$H = a\mathbb{1} \otimes \mathbb{1} + b_0 Z \otimes \mathbb{1} + b_1 \mathbb{1} \otimes Z + b_2 Z \otimes Z + b_3 Y \otimes Y + b_4 X \otimes X.$$

ДЛЯ СПРАВКИ Все используемые Марией члены и коэффициенты взяты из статьи «Масштабируемая квантовая симуляция молекулярных энергий»[1]. Точные коэффициенты зависят от длины связи между атомами водорода, но все эти константы любезно набраны за вас и находятся в репозитории исходного кода данной книги по адресу: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

Имея на руках это представление гамильтонианы Марии, самое время выяснить, как на самом деле ее использовать. В этой гамильтониане есть шесть членов, и какой же член мы должны применить в первую очередь? Имеет ли значение порядок? К сожалению, порядок, в котором члены используются, часто имеет значение при симулировании эволюции системы в условиях гамильтонианы. В следующем далее разделе мы узнаем о методе, который позволяет нам разбивать эволюцию системы на небольшие шаги, чтобы симулировать эволюцию сразу во всех членах.

10.6 Претерпевая (очень малые) изменения

В этом месте полезно сделать шаг назад и оценить, на каком шаге мы помогаем Марии. Мы узнали, как разбивать произвольные гамильтонианы на суммы матриц Паули и как использовать операцию `Exp` для симулирования эволюции в условиях каждого члена в этой сумме. В целях симулирования произвольных гамильтониан остается лишь скомбинировать эти симуляции для симулирования всей гамильтонианы в целом (рис. 10.9). Для этого мы можем применить еще один квантово-вычислительный трюк, именуемый *разложением Троттера–Сузуки*.

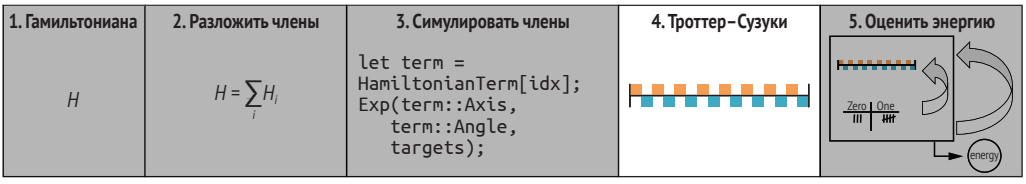


Рис. 10.9 В этом разделе мы займемся разведкой использования разложения Троттера–Сузуки с целью симулирования действия полной гамильтонианы путем ее разбивки на гораздо меньшие эволюции каждого члена из шага 3

[1] *П. Дж. Дж. О'Молли и соавт.* Масштабируемая квантовая симуляция молекулярных энергий (P. J. J. O'Malley et al. *Scalable Quantum Simulation of Molecular Energies*, 2015) // https://arxiv.org/abs/1512.06860.

Однако, прежде чем вдаваться в детали декомпозиции Троттера–Сузуки, давайте вернемся к аналогии с картой, которую мы использовали на протяжении всей книги, чтобы разбить понятия линейной алгебры (обсуждаемые в приложении С к книге).

Предположим, что мы проводим разведку центра Финикса и решаем посмотреть, каково это – идти на северо-восток по всему городу. Если мы начнем с того, что пройдем несколько кварталов на север, а затем несколько кварталов на восток, то маршрут, который мы проследим на карте, не будет сильно похож на диагональную линию. С другой стороны, если в каждом квартале мы будем переключаться между севером и востоком, то мы проследим что-то внешне гораздо более похожее на путь на карте Миннеаполиса, которая приводится в приложении С к книге. То есть мы можем просимулировать путь, которым мы могли бы пройти через Миннеаполис, даже если мы застряли в Финиксе, быстро переключая направления нашего движения; см. рис. 10.10.

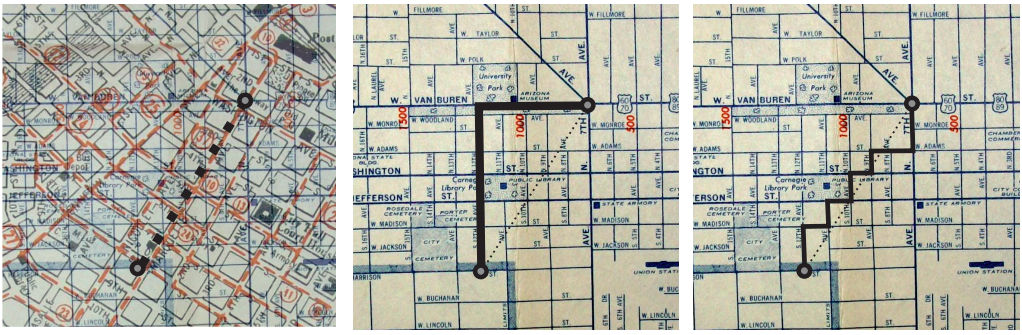


Рис. 10.10 Если мы находимся в центре Финикса, то мы все равно можем просимулировать наш путь по центру Миннеаполиса, быстро меняя направления. В идеале мы бы шли к месту назначения просто по диагонали, но, учитывая планировку улиц, мы можем приблизиться к диагонали, делая короткие зигзаги. Автор фото: davecito

В предыдущем разделе мы увидели, что так же, как и состояния, разные члены гамильтонианы можно рассматривать как направления на многомерной карте местности. Тензорные произведения матриц Паули, такие как $Z \otimes$ и $X \otimes Z$, играют роль, аналогичную сторонам света или осям карты. Однако при попытке просимулировать гамильтониану Марии она указывает не вдоль одной оси, а вдоль своего рода диагонали в этом многомерном пространстве. Вот тут-то и вступает в дело разложение Троттера–Сузуки.

Подобно тому, как наш путь выглядит более диагональным, когда мы быстро меняем направление движения, мы можем быстро переключаться между симулированием разных гамильтониановых членов. Как показано на рис. 10.11, разложение Троттера–Сузуки говорит нам о том, что при быстром переключении в таком ключе мы приближенно эволюционируем в условиях суммы разных членов, которые мы симулируем.

Рис. 10.11 Использование декомпозиции Троттера–Сузуки для аппроксимирования эволюции сразу в двух гамильтониановых членах. Как и в предыдущей аналогии с картой, если мы хотим как можно быстрее применить эффект двух гамильтониан, мы должны понемногу чередовать эволюционирование в условиях каждого из них до тех пор, пока не достигнем полной эволюции

В принципе, мы могли бы записать это на Q# как цикл `for`. В псевдокоде у нас может быть что-то вроде следующего.

Листинг 10.9 Симулирование гамильтонианы с использованием Троттера–Сузуки

```
operation EvolveUnderHamiltonian(time, hamiltonian, register) {          ❶
    for idx in 0..nTimeSteps - 1 {                                       ❷
        for term in hamiltonian {                                        ❸
            evolve under term for time / nTimeSteps
        }
    }
}
```

❶ Поскольку это псевдокод, давайте на мгновение не будем беспокоиться о типах. Эта операция не будет компилироваться без типов, но пока это нормально.

❷ Для каждого шага, на которые мы хотим разделить нашу симуляцию (подумайте о городских кварталах в Финиксе или пикселах на экране), нам нужно выполнить немного каждого гамильтонианова члена.

❸ Внутри каждого временного шага мы можем перебирать каждый симулируемый член и симулировать каждый из них в течение одного шага.

К счастью, Q# предоставляет стандартно-библиотечную функцию, которая делает для нас именно это: `DecomposedIntoTimeStepsCA`. В листинге 10.10 мы покажем, как вызов `DecomposedIntoTimeStepsCA` позволяет легко использовать разложение Троттера–Сузуки для симулирования эволюции в условиях гамильтонианы Марии. Функция `DecomposedIntoTimeStepsCA` поддерживает более высокопорядковые разложения Троттера–Сузуки, чем первопорядковая аппроксимация, которую мы разведали в данной главе до этого (представленная порядком `trotterOrder`, равным 1). В некоторых случаях эта функция бывает полезна для повы-

шения точности симуляции, но для наших целей отлично подходит порядок (`trotterOrder`) 1.

Листинг 10.10 operations.qs: использование DecomposedIntoTimeStepsCA

```
operation EvolveUnderHamiltonian(
    idxBondLength : Int,                                    ❶
    trotterStepSize : Double,                               ❷
    qubits : Qubit[])
: Unit is Adj + Ctl {
    let trotterOrder = 1;                                   ❸
    let op = EvolveUnderHamiltonianTerm(
            idxBondLength, _, _, _);                        ❹
    (DecomposedIntoTimeStepsCA ((5, op), trotterOrder))
        (trotterStepSize, qubits);                          ❺
}
```

❶ EvolveUnderHamiltonian применяет надлежащую гамильтониану, основываясь на коэффициентах для желаемой длины связи молекулы H_2, с которой Мария попросила нас помочь.

❷ Размер шага, который представляет продолжительность, с которой мы хотим симулировать эволюцию гамильтонианы.

❸ В некоторых случаях trotterOrder > 1 бывает полезен для повышения точности симуляции, но trotterOrder, равный 1, для наших целей отлично подходит.

❹ Частичное применение может зафиксировать входное значение idxBondLength в EvolveUnderHamiltonianTerm, оставляя аргументы idxTerm, stepSize и qubits пустыми.

❺ Эта функция выдает операцию, которую можно использовать для автоматического симулирования эволюции в условиях всей гамильтонианы, используя операцию, которая симулирует каждый член по одному, чтобы мы имели возможность продолжать и применить его.

10.7 Окончательная сборка

Теперь, когда у нас есть более точное понимание того, что такое гамильтонианы и каким образом можно симулировать эволюцию с ними, чтобы понять, как квантовые системы меняются во времени, мы готовы составить программу, которая поможет Марии решить ее вопрос (рис. 10.12). Напомним, что Мария является химиком, который изучает энергии основного состояния (т. е. минимально возможные энергии) разных химических веществ. Она попросила нас помочь выяснить энергии основного состояния молекулы H_2 с помощью нашего квантового устройства. Поскольку атомы водорода, из которых состоят молекулы H_2, также являются квантовыми системами, гораздо проще симулировать поведение H_2 с помощью кубитов, чем с помощью классического компьютера.

ДЛЯ СПРАВКИ Квантовые компьютеры настолько хорошо подходят для симулирования поведения других квантовых систем, что, возможно, это приложение было первым, которое когда-либо было предложено для квантовых вычислений!

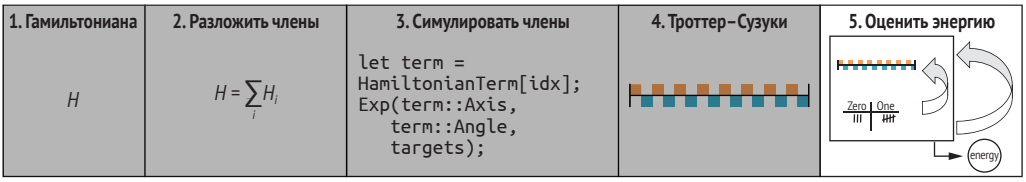


Рис. 10.12 Последний шаг, который поможет Марии просимулировать ее молекулу H_2, заключается в использовании фазового оценивания для прочтения энергии основного состояния

Рисунок 10.13 напоминает обо всех шагах и технических приемах, которые мы усвоили в этой главе, чтобы просимулировать эволюцию молекулы H_2 Марии в нашем квантовом устройстве.

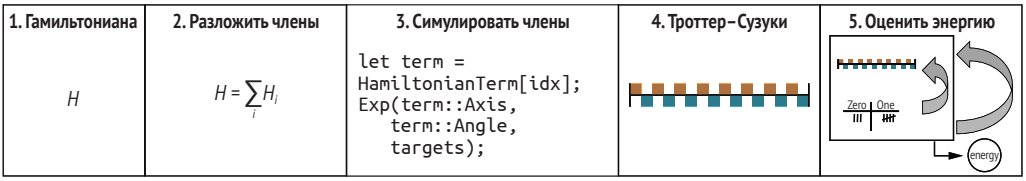


Рис. 10.13 Обзор шагов, разработанных в этой главе, чтобы помочь Марии узнать энергию основного состояния ее молекулы

Таким образом, будучи квантовыми разработчиками, мы можем сотрудничать с Марией, чтобы симулировать эволюцию молекулы H_2 во времени и рассчитывать энергию основного состояния благодаря уравнению Шредингера. Ключевым моментом, о котором следует помнить, является то, что возможные энергетические уровни молекулы H_2 соответствуют разным *собственным состояниям* гамильтонианы.

Предположим, что наши кубиты находятся в собственном состоянии гамильтонианы. Тогда симулирование эволюции в условиях этой гамильтонианы не изменит состояние нашего реестра кубитов, *за исключением* применения глобальной фазы, пропорциональной энергии этого состояния. Указанная энергия точно говорит нам о том, что конкретно нам нужно для решения задачи Марии, но глобальные фазы ненаблюдаемы. К счастью, в главе 9 мы узнали из игры Ланселота и Дагонета, как превращать глобальные фазы в нечто, что мы можем усваивать с помощью фазового оценивания, – вот отличное место для применения этого технического приема! Подводя итог, шаги по сотрудничеству с Марией и решению ее задачи заключаются в следующем.

1. Подготовить начальное состояние, которое Мария нам предоставила. В этом случае она любезно говорит нам подготовить $|10\rangle$.
2. Разбить гамильтониану, которая представляет систему, на небольшие шаги и затем последовательно их просимулировать, что и будет представлять всю операцию.
3. Применить каждый шаг, представляющий гамильтониану, к нашему первоначальному состоянию.

4 Использовать алгоритм фазового оценивания, чтобы узнать о накопленной глобальной фазе в нашем квантовом состоянии, которая будет пропорциональна энергии.

У нас есть навыки и исходный код из предыдущих разделов этой главы, чтобы собрать все это воедино, поэтому давайте попробуем.

Начиная с файла Q# (здесь он называется operations.qs в соответствии с тем, что мы видели в предыдущих главах) мы откроем несколько пространств имен для использования готовых функций и операций.

Листинг 10.11 operations.qs: пространства имен, необходимые для QDK

```
namespace HamiltonianSimulation {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Canon;                    (1)
    open Microsoft.Quantum.Simulation;               (2)
    open Microsoft.Quantum.Characterization;         (3)
```

(1) Мы встречали Microsoft.Quantum.Intrinsic и Microsoft.Quantum.Canon раньше; в этих пространствах имен есть базовые утилиты / вспомогательные функции и операции, которые нам нужны.
(2) Microsoft.Quantum.Simulation – это пространство имен для QDK, в котором, как и следовало ожидать, есть утилиты для симулирования систем.
(3) Microsoft.Quantum.Characterization имеет простые в использовании имплементации алгоритмов фазового оценивания, которые мы разработали в главе 9.

Далее нам нужно добавить данные, которые есть у Марии о ее молекуле. Все это набрано за вас в демонстрационном файле в репозитории книги на GitHub: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp/blob/master/ch10/operations.qs. Функции `H2BondLengths`, `H2Coeff` и `H2IdentityCoeff` – это то, что нам нужно (их довольно долго воспроизводить здесь в тексте).

Имея все данные о коэффициентах от Марии в нашем файле, нам понадобятся фактические члены / структура гамильтонианы, которую мы будем использовать с только что добавленными нами коэффициентами. В следующем ниже листинге показана схема функции, которая возвращает члены гамильтонианы Марии, выраженные в операторах Паули, а также операция, которая будет подготавливать двухкубитовый реестр в правильное состояние для этого алгоритма.

Листинг 10.12 operations.qs: функция, возвращающая члены из гамильтонианы

```
function H2Terms(idxHamiltonian : Int)                (1)
: (Pauli[], Int[]) {
    return [
        ([PauliZ], [0]),                              (2)
        ([PauliZ], [1]),
        ([PauliZ, PauliZ], [0, 1]),                   (3)
        ([PauliY, PauliY], [0, 1]),
        ([PauliX, PauliX], [0, 1])
```

```
    ][idxHamiltonian];
}

operation PrepareInitalState(q : Qubit[])          ❹
: Unit {
    X(q[0]);
}
```

❶ Функция H2Terms позволяет легко конструировать члены гамильтонианы Марии.
❷ Эта функция на самом деле представляет собой просто жестко закодированный список кортежей, описывающих члены гамильтонианы. Этот первый кортеж говорит о том, что первым членом гамильтонианы является операция PauliZ на нулевом кубите.
❸ Применяет PauliZ как к нулевому, так и к первому кубиту.
❹ Нам также нужен способ подготовить кубиты для алгоритма. Следуя совету Марии, мы помещаем первый кубит в состояние |1⟩, оставляя остальные входные кубиты в состоянии |0⟩.

С целью выполнения шагов 2 и 3 нашего квантового алгоритма нам нужны операции, которые мы определили ранее: `EvolveUnderHamiltonianTerm` и `EvolveUnderHamiltonian`.

Глобальные фазы и функтор Controlled

Подобно тому, что мы видели в главе 9, применение `EvolveUnderHamiltonian` ничего не делает с кубитами, подготовленными в собственном состоянии гамильтонианы Марии – и действительно, в этом весь смысл! В главе 9 мы смогли решить это, используя функтор `Controlled`, чтобы превратить глобальную фазу, полученную в результате применения операции Дагонета на кубите, подготовленном в собственном состоянии, в локальную фазу, которую можно наблюдать, а затем с помощью фазовой отдачи применить эту фазу к контрольному кубиту. Операция `EstimateEnergy`, предоставляемая Комплектом инструментов для квантовой разработки, использует точно такой же трюк, чтобы узнать то, что в противном случае было бы глобальной фазой нашей операции `EvolveUnderHamiltonian`. Это означает, что очень важно, чтобы наша операция поддерживала функтор `Controlled`, добавив `Ctl` к сигнатуре для каждой операции, которую мы передаем в `EstimateEnergy` в рамках помощи Марии.

Наконец, мы можем применить операцию `EstimateEnergy`, поставляемую с Комплектом инструментов для квантовой разработки, с целью автоматизирования применения шагов Троттера–Сузуки и шага оценивания фазы. Мы также можем использовать встроенную операцию оценивания фазы, в которой имплементирована усовершенствованная версия алгоритма фазового оценивания, о котором мы узнали в главе 9. Например, в библиотеке `Microsoft.Quantum.Simulation` есть операция под названием `EstimateEnergy`, в которой используется фазовое оценивание для оценивания энергии собственного состояния. Она принимает спецификацию числа кубитов (`nQubits`), операцию подготовки желаемого начального состояния (`PrepareInitialState`), способ применения га-

мильтонианы (`trotterStep`) и алгоритм, который мы хотим использовать для оценивания фазы, полученной в результате применения гамильтонианы. Давайте проверим это в действии.

Листинг 10.13 operations.qs: операция Q#, которая оценивает энергию основного состояния H_2

```
operation EstimateH2Energy(idxBondLength : Int)
: Double {                                          ❶
    let nQubits = 2;                                ❷
    let trotterStepSize = 1.0;                      ❸
    let trotterStep = EvolveUnderHamiltonian(
        idxBondLength, trotterStepSize, _);         ❹
    let estPhase = EstimateEnergy(nQubits,          ❺
        PrepareInitalState, trotterStep,
        RobustPhaseEstimation(6, _, _));
    return estPhase / trotterStepSize
        + H2IdentityCoeff(idxBondLength);           ❻
}
```

❶ Вот оно! Операция EstimateH2Energy принимает индекс длины связи молекулы и возвращает энергию ее основного состояния, или низшее энергетическое состояние.
❷ Определяет, что нам нужно два кубита для симулирования этой системы.
❸ Задает параметр масштаба для шагов Троттера–Сузуки, которые применяют члены гамильтонианы к кубитам.
❹ Строится на основе ApplyHamiltonian и дает удобное название для операции, которая применяет члены нашей гамильтонианы с параметрами.
❺ Мы имеем встроенную в Microsoft.Quantum.Simulation операцию, которая оценивает фазу, полученную в результате применения нашей гамильтонианы, которая, как мы знаем, представляет энергию системы.
❻ В целях обеспечения того, чтобы единицы измерения соответствовали возвращаемой энергии, мы должны разделить на размер шага trotterStepSize и добавить энергию из члена тождества в гамильтониане.

Теперь давайте наконец выполним алгоритм! Поскольку энергия основного состояния является функцией от длины связи молекулы, мы можем использовать главную программу Python для выполнения алгоритма Q#, а затем построить график результатов как функцию от длины связи.

Листинг 10.14 host.py: настройка симулирования на Python

```
import qsharp                                       ❶
import HamiltonianSimulation as H2Simulation

bond_lengths = H2Simulation.H2BondLengths.simulate()    ❷

def estimate_energy(bond_index: float,                  ❸
                    n_measurements_per_scale: int = 3
    ) -> float:
    print(f"Оценивание энергии для длины связи {bond_lengths[bond_index]} Å.")
    return min([H2Simulation.EstimateH2Energy.simulate(idxBondLength=bond_index)
                    for _ in range(n_measurements_per_scale)])
```

❶ Импортирует пакет Python для Q#, а затем импортирует пространство имен Q# (HamiltonianSimulation) из нашего файла operations.qs. Python'овский пакет qsharp делает пространства имен Q# доступными в виде обычных инструкций импортирования.
❷ В целях упрощения задачи извлекает список длин связей, которые мы можем просимулировать для H_2, из функции H2BondLengths на языке Q#.
❸ Функция estimate_energy является Python'овской оберткой для операции языка Q#, EstimateH2Energy, но выполняет ее несколько раз с целью обеспечения минимизирования оценки энергии.

Зачем нам нужно выполнять EstimateH2Energy несколько раз?

Состояние |01⟩, которое предоставила нам Мария, на самом деле не является собственным состоянием какой-либо гамильтонианы H_2, а является чем-то, что она вычислила, используя квантово-химическую аппроксимацию, именуемую *теорией Хартри–Фока*. Поскольку квантовая химия является ее областью знаний, она способна помочь, предоставив такие аппроксимации.

На практике это означает, что при выполнении фазового оценивания с помощью инструментов, предлагаемых пространством имен `Microsoft.Quantum.Characterization`, мы не усваиваем энергию конкретного собственного состояния; вместо этого мы случайным образом проецируем на собственное состояние и усваиваем его энергию. Поскольку наше начальное состояние является довольно хорошей аппроксимацией, в большинстве случаев мы будем проецировать на состояние с наименьшей энергией гамильтонианы Марии (т. е. основное состояние), но нам может не повезти, и мы правильно узнаем энергию неправильного собственного состояния. Поскольку мы ищем наименьшую энергию, многократное выполнение и взятие минимума делает гораздо более вероятным то, что мы узнаем нужную нам энергию.

После того как в главной программе Python все будет настроено, нам останется лишь написать и выполнить главную функцию.

Листинг 10.15 host.py: главная программа для нашей симуляции

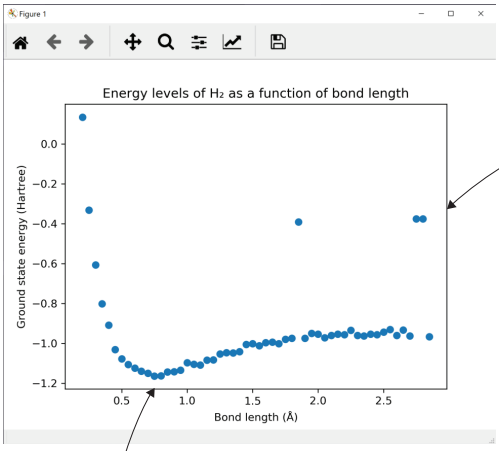
```
if __name__ == "__main__":
    import matplotlib.pyplot as plt                                          ❶

    print(f"Число длин связей: {len(bond_lengths)}.\n")
    energies = [estimate_energy(i) for i in range(len(bond_lengths))]        ❷
    plt.figure()
    plt.plot(bond_lengths, energies, энергетические уровни H2
             ➥ как функция длины связи')
    plt.xlabel('Длина связи (Å)')
    plt.ylabel('Энергия основного состояния (Хартри)')
    plt.show()                                                               ❸
```

Настраивает данные и стиль для графика.

❶ Выполнив host.py в качестве скрипта, мы построим график расчетных энергий основного состояния из квантового алгоритма на языке Q#.
❷ Непосредственно генерирует список расчетных энергий для каждой длины связи молекулы H_2.
❸ Вызов plt.show() должен показать всплывающее окно либо вернуть изображение графика!

На рис. 10.14 показан пример того, что должно вывести выполнение команды `python host.py`. На этом графике показаны результаты нашей симуляции для различных гамильтониан для разных длин связей молекулы H_2. Мы видим, что энергия самого низкого состояния намного выше, когда длина связи является короткой, и как бы выравнивается по мере того, как связи становятся длиннее. Наименьшая возможная энергия должна возникать при длине связи, приближенно равной 0.75 Å. Если присмотреться, то стабильная (т. е. равновесная) длина связи для водорода составляет 0.74 Å! Так что да, оказывается, молекула Марии довольно хорошо известна, но мы видим, как мы могли бы проследить этот процесс не только для других химических веществ, но и для симулирования других квантовых систем.

При выполнении нашей программы Q# с помощью Python мы получаем график Matplotlib, показывающий результаты симулирования нашей программы Q# на классическом компьютере



Каждая точка представляет оценку энергии основного состояния H_2 при разной длине связи

Для описания энергии разных молекул химики, как правило, используют единицу, именуемую Хартри

Для каждой длины связи, измеренной в ангстремах (пишется Å, равна десятым долям нанометра), Мария предоставляет нам другую гамильтониану, описывающую энергетические уровни H_2 в этой длине

Иногда используемая нами аппроксимация в качестве начального состояния не так хороша, и мы усваиваем энергию состояния, отличающегося от основного состояния.

Повторяя чаще и минимизируя, мы можем устранить этот вид ошибки. Попробуйте!

Оценки, полученные из нашей программы Q#, можно использовать, чтобы найти длину связи, которая дает нам наименьшую энергию основного состояния.

Здесь наименьшая энергия основного состояния составляет около 0.75 Å (0.075 нанометра). Фактическое число составляет около 0.74 Å, так что мы довольно близко к нему подобрались с помощью нашей квантовой программы!

Рис. 10.14 Образец графика, который должен быть произведен в результате выполнения host.py. Точные данные, вероятно, будут отличаться, но в целом мы должны увидеть, что минимальная энергия основного состояния происходит примерно на уровне 0.75 Å на горизонтальной оси, являясь длиной связи с наименьшей энергией.

Поздравляем: мы имплементировали наше первое практическое приложение для квантового компьютера! Разумеется, фактическое химическое вещество, которое мы использовали здесь, является довольно простым, но этот процесс соблюдается для большинства других квантовых систем, которые мы, возможно, захотим просимулировать. В следующих двух главах мы рассмотрим два других применения квантовых компью-

теров: неструктурированный поиск с помощью алгоритма Гровера и факторизацию чисел с помощью алгоритма Шора.

Резюме

- Одно из самых захватывающих применений квантовых вычислений состоит в том, чтобы помогать нам понимать свойства квантово-механических систем, таких как химические взаимодействия.
- Мы можем рассуждать о квантовой механике по-разному. Используя Python и Q#, мы рассуждали о квантовой механике как о своего рода вычислении, но химики и физики могут рассуждать о квантовой механике как о наборе правил, описывающих варианты взаимодействий и поведения физических систем.
- Физики и химики используют особый вид матрицы, именуемой *гамильтонианой*, чтобы предсказывать характер изменения квантово-механических систем во времени. Если мы сможем просимулировать гамильтониану на квантовом компьютере, то сможем симулировать физические системы, описываемые этими матрицами.
- Важным частным случаем гамильтонианы является тензорное произведение матриц Паули. Эти гамильтонианы описывают своего рода обобщение поворотов, которые мы встречали на протяжении всей книги, и могут быть просимулированы с помощью операции `Exp` языка Q#.
- Более сложные гамильтонианы могут быть разбиты на суммы более простых гамильтониан, что позволяет нам симулировать по одной части гамильтонианы за раз.
- Если быстро чередовать части симулируемой гамильтонианы, то можно получить более точную аппроксимацию симулирования полной гамильтонианы. Это похоже на то, как быстрота чередования ходьбы на север и на запад выглядит немного похожей на ходьбу по диагонали на северо-запад при уменьшении масштаба.
- Используя химические модели (такие, какие мы можем получить от друзей-химиков), мы можем писать и симулировать гамильтонианы для квантовой химии. Сочетание этой симуляции с фазовым оцениванием позволяет нам усваивать энергетическую структуру разных химических веществ, помогая нам предсказывать их поведение.

11 Поиск с помощью квантовых компьютеров

Эта глава охватывает следующие ниже темы:

- поиск по неструктурированным данным с помощью квантового алгоритма;
- использование оценщика ресурсов QDK для понимания стоимости выполнения алгоритмов;
- отражение квантовых реестров вокруг состояний.

В главе 10 мы ушли с головой в наше первое применение квантовых вычислений, работая с нашей коллегой Марией, чтобы помочь ей вычислить энергию основного состояния молекулы водорода. Для этого мы имплементировали алгоритм гамильтонианової симуляции, в котором использовалось несколько технических приемов фазового оценивания, разработанных в главе 9.

В этой главе мы рассмотрим еще одно применение квантовых вычислений: поиск данных. Эта область приложений всегда является горячей темой в сферах высокопроизводительных вычислений и демонстрирует еще один способ использования технических приемов, которые мы узнали ранее, для строительства еще одной квантовой программы, в данном случае основанной на фазовой отдаче. Мы также рассмотрим *оценщика ресурсов*, встроенного в Комплект инструментов для квантовой разработки (QDK), чтобы разобраться в том, как он помогает нам понимать масштабирование наших квантовых программ, даже если они становятся слишком большими для локального выполнения.

11.1 Поиск по неструктурированным данным

Предположим, что мы хотим просмотреть некоторые данные, чтобы найти номер телефона контактного лица. Если список контактов отсортирован по имени, то довольно легко найти номер телефона, связанный с тем или иным именем, с помощью *двоичного поиска*:

Алгоритм 11.1: псевдокод двоичного поиска

1. Подобрать пару имя / номер телефона в середине нашего списка. Назовем эту пару опорным элементом.
2. Если имя опорного элемента является именем, которое мы ищем, то вернуть номер телефона опорного элемента.
3. Если имя, которое мы ищем, стоит перед именем опорного элемента, то повторить поиск в первой половине списка.
4. В противном случае если имя, которое мы ищем, следует за именем опорного элемента, то повторить поиск во второй половине списка.

Не просто персонаж из «Звездного пути»

В этой главе мы будем много говорить о поиске по данным. Эти данные могут поступать в самых разных формах:

- номера телефонов;
- имена собак;
- результаты измерения погоды;
- типы дверных звонков.

Все эти данные объединяет то, что мы можем представить их на классических компьютерах в виде цепочек битов – битовых строк, используя множество различных соглашений о том, как это представление должно работать.

Поиск таким путем может осуществляться довольно быстро и является ключом к тому, каким образом можно проводить поиск по базам данных, полным информации. Проблема в том, что в алгоритме 11.1 мы критически зависим от сортировки списка имен и телефонных номеров. Если он не отсортирован, то двоичный поиск просто не сработает.

ПРИМЕЧАНИЕ Теперь, когда у нас есть все необходимое для решения более сложных задач с помощью квантовых компьютеров, сценарий этой главы будет немного сложнее, чем большинство наших предыдущих игр и сценариев. Не волнуйтесь, если что-то будет не вполне понятно с места в карьер: не торопитесь и читайте медленнее. Мы обещаем, что это будет стоить вашего времени!

Говоря по-другому, в целях осуществления быстрого поиска по нашим данным нам нужно применить к ним какую-то структуру: отсор-

тировать данные либо принять какое-то другое допущение, которое позволит нам избегать необходимости просматривать каждый элемент в отдельности. Если никакой структуры у нас нет, то лучшее, что можно сделать, – это случайным образом просматривать данные до тех пор, пока мы не найдем то, что нам нужно. Шаги, перечисленные в алгоритме 11.2, показывают псевдокод того, как мы будем осуществлять поиск по бесструктурному списку. Нам может повезти, но в среднем случайный поиск будет только в два раза быстрее, чем просмотр каждого отдельного элемента:

Алгоритм 11.2: псевдокод поиска по неструктурированному списку

1. Выбрать из списка случайный элемент.
2. Если этот элемент правильный, то вернуть его. В противном случае выбрать новый элемент и повторить.

Тот факт, что поиск по неструктурированным спискам затруднен, также является основой большей части криптографии. В этом случае, вместо того чтобы писать список явно, наша задача при попытке взломать алгоритм шифрования состоит в том, чтобы пробовать разные ключи до тех пор, пока один из них не сработает. Функция расшифровки может рассматриваться как *неявное* определение списка, в котором есть один специальный «маркированный элемент», соответствующий правильному секретному ключу.

Псевдокод в алгоритме 11.3 может представлять эту задачу дешифровки. Выбираемое нами случайное входное значение является ключом, который мы используем с «функцией» или алгоритмом расшифровки, чтобы увидеть, что он действительно расшифровывает сообщение:

Алгоритм 11.3: псевдокод поиска в неструктурированных данных, поступающих на вход в функцию

1. Выбрать случайное входное значение.
2. Вызвать нашу функцию с этим входным значением. Если оно сработало, то вернуть входное значение.
3. В противном случае выбрать новое случайное входное значение и повторить.

Если бы мы смогли быстрее производить поиск по неструктурированным спискам, то это позволило бы нам сортировать базы данных, решать математические задачи или – да – даже взламывать некоторые виды классического шифрования.

Пожалуй, это вызовет удивление, но если существует возможность написать функцию, определяющую наш список, как квантовую операцию (используя то, что мы узнали об оракулах в главе 8), то мы сможем применять квантовый алгоритм, известный под названием *алгоритма Гровера*, чтобы отыскивать входное значение намного быстрее, чем с помощью алгоритма 11.3.

ДЛЯ СПРАВКИ Мы приближаемся к концу книги, а это значит, что у нас есть возможность собрать воедино все то, что мы узнали на протяжении всей книги. В частности, в этой главе мы будем использовать то, что мы узнали об *оракулах* из игры Нимуэ и Мерлин в главе 8, для представления входных данных алгоритма Гровера. Если вам нужно освежить свою память в отношении того, что такое оракулы, то не беспокойтесь; глава 8 вам поможет.

При выполнении алгоритма Гровера мы ищем одно или несколько конкретных значений функции по всем возможным значениям, поступающим на вход функции. Если мы хотим выполнить поиск в неструктурированном списке данных, то мы можем рассмотреть возможность определения функции, которая отвечает за поиск конкретной записи в списке. Затем мы можем выполнить поиск по данным, поступающим на вход в эту функцию, чтобы отыскать те или иные выходные значения функции, которые мы хотим.

Возьмем сценарий, в котором нам нужно расшифровать сообщение за 1 минуту. Нам нужно попробовать 2.5 миллиона разных ключей, но только один сработает для расшифровки сообщения, и попытка использовать по одному ключу за раз займет слишком много времени. Мы можем применить алгоритм Гровера и функцию, представляющую задачу, такую как «расшифровывает ли этот криптографический ключ конкретное сообщение?», чтобы отыскать правильный ключ гораздо быстрее и без необходимости проверять каждый ключ по отдельности! Это очень похоже на пример с замком, показанный на рис. 11.1, где мы рассматриваем разные возможные ключи как входные значения.

Рис. 11.1 Структурированный и неструктурированный поиски. Если мы просматриваем адресную книгу в алфавитном порядке, то в данных есть некоторая структура, которую мы можем использовать для более быстрого отыскания данных. В случае с коробкой случайных ключей мы просто должны продолжать пробовать случайные ключи до тех пор, пока замок не откроется

Мы сделаем функцию, необходимую для алгоритма Гровера, более точной, когда вернемся к оракулам позже в этой главе; но полезно помнить, что при поиске с помощью алгоритма Гровера мы производим поиск в значениях, поступающих на вход в функцию, а не в списке данных. Имея это в виду, ниже приведен псевдокод алгоритма Гровера:

Алгоритм 11.4: псевдокод выполнения неструктурированного поиска (алгоритма Гровера)

1. Выделить реестр кубитов, достаточно большой, чтобы представлять все поступающие на вход в функцию значения, в которых мы выполняем поиск.
2. Подготовить реестр в равномерном суперпозиционном состоянии: т. е. все возможные состояния имеют одинаковую амплитуду. Это связано с тем, что из-за типа задачи у нас нет никакой дополнительной информации о том, какое входное значение является «правильным», поэтому состояние представляет собой равномерное вероятностное распределение (или *априорное распределение*) на данных.
3. Отразить реестр вокруг маркированного состояния или состояния, которое мы ищем. Здесь *отражение* означает подбор конкретного состояния и переворачивание на нем знака; отражение и то, как имплементировать отражения на языке Q#, будут рассмотрены подробнее в следующем разделе.
4. Отразить реестр вокруг первоначального состояния (равномерной суперпозиции).
5. Повторять шаги 3 и 4 до тех пор, пока вероятность измерения искомого элемента не станет достаточно высокой. Затем измерить реестр. Мы можем математически вычислить оптимальное число раз, которое нам нужно это проделать, чтобы максимизировать правильный ответ.

Эти шаги показаны на рис. 11.2.

ПРИМЕЧАНИЕ Когда мы пройдем эту главу, мы увидим, что на алгоритм Гровера, как вариант, можно смотреть как на своего рода поворот между состояниями, представляющими, что мы нашли или не нашли правильный маркированный элемент: ключ дешифровки для нашего сценария. Если мы применим шаги 3 и 4 алгоритма Гровера слишком много раз, то мы повернем мимо искомого состояния, поэтому выбор числа итераций является неотъемлемой частью алгоритма!

На рис. 11.3 показан пример соотнесения стоимости классического поиска в неструктурированном списке с использованием алгоритма Гровера.

ДЛЯ СПРАВКИ Описать то, что показано на рис. 11.3, можно, воспользовавшись концепцией, именуемой *асимптотической сложностью*. В частности, мы говорим, что классический неструктурированный поиск требует $O(N)$ вызовов функции для поиска по N входным значениям, в то время как алгоритм Гровера требует $O(\sqrt{N})$ вызовов. Не волнуйтесь, если это вам незнакомо; но если вы интересуетесь тем, как разобраться в алгоритмах в таком ключе, то рекомендуем ознакомиться с главой 1 книги «Грокаем алгоритмы» Адитьи Й. Бхаргавы (*Grokking Algorithms*, Aditya *Y*. Bhargava, Manning, 2016).

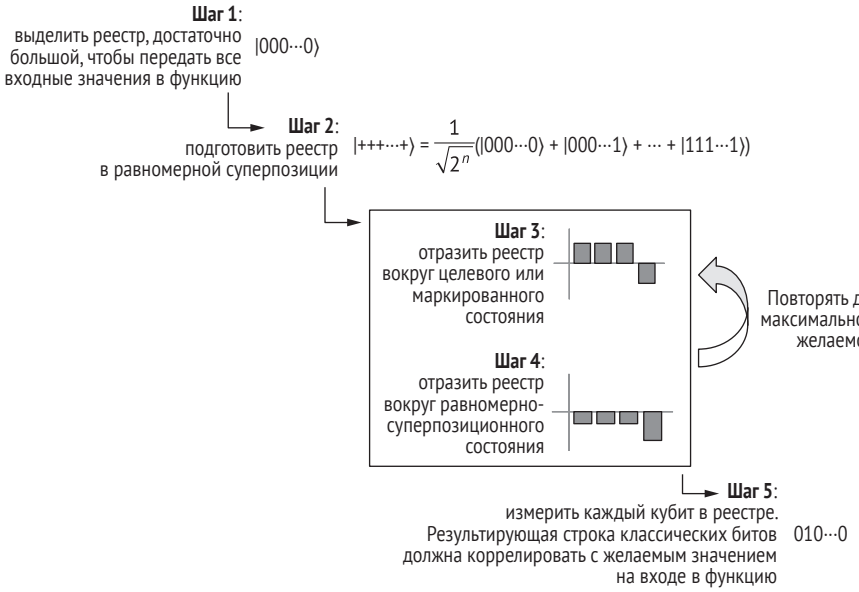


Рис. 11.2 Шаги алгоритма Гровера, который выполняет поиск по значениям, поступающим на вход в функцию, ища конкретное значение на выходе из функции. Мы начинаем с выделения реестра кубитов, достаточно большого, чтобы иметь возможность представить все входные значения, в которых мы хотим выполнять поиск, а затем помещаем их в равномерно-суперпозиционное состояние. Наконец, мы можем отразить состояние реестра нужное число раз, чтобы максимизировать вероятность измерить ответ, который мы ищем

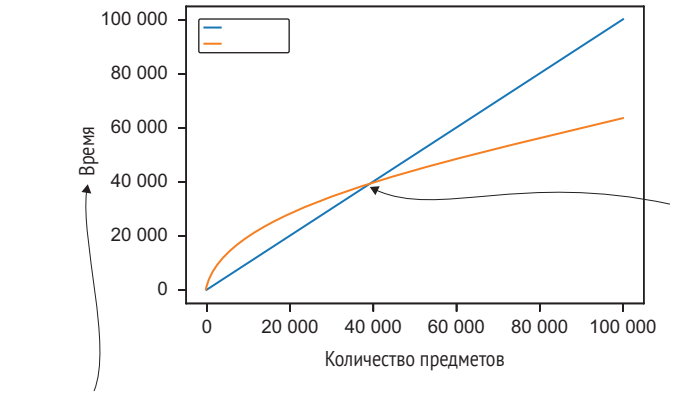


Рис. 11.3 Пример того, как время, затрачиваемое на поиск по неструктурированным спискам, может масштабироваться для классических и квантовых компьютеров. Мы видим, что для меньшего числа элементов квантовый подход занимает больше времени; но по мере увеличения числа элементов в поиске квантовый подход к поиску занимает меньше времени

Как и прежде, давайте приступим и посмотрим, как выглядит этот исходный код. В листинге 11.1 приведен пример операции Q#, в которой используется алгоритм Гровера для поиска маркированного элемента в неструктурированном списке. Здесь вместо 2.5 миллиона ключей мы уменьшим область действия до 8 ключей, при этом маркированный элемент или правильный ключ будет обозначен целым числом в диапазоне от 0 до 7. Да-да, это означает, что мы могли бы решить эту задачу так же быстро на классическом компьютере. Однако в конце главы мы увидим, как при использовании алгоритма Гровера по мере увеличения числа необходимых для поиска ключей число шагов или вычислений, необходимых для поиска правильного ключа, значительно уменьшается. Кроме того, здесь в нашем примере исходного кода функция, представляющая алгоритм дешифровки, на самом деле не выполняет никакого дешифрования; она просто действует так, как будто играет в игру на угадывание, и возвращает булево значение, если ей дан правильный ключ. Имплементирование конкретного алгоритма дешифровки для этой главы выходит за рамки книги и, вероятно, потребует некоторого исследования. Здесь цель состоит в том, чтобы показать, как использование алгоритма Гровера для поиска в данных, поступающих на вход в функцию, может ускорить решение конкретных задач.

Листинг 11.1 operations.qs: исходный код Q#, который выполняет поисковый алгоритм Гровера

```
operation RunGroverSearch(nItems : Int, idxMarkedItem : Int) : Unit {
    let markItem = ApplyOracle(
        idxMarkedItem, _, _);                                    ❶
    let foundItem = SearchForMarkedItem(
        nItems, markItem);                                       ❷
    Message(
        $"Промаркирован {idxMarkedItem} и найден
    ➥ {foundItem}.");                                           ❸
}
```

❶ Можно задействовать частичное применение для включения индекса маркированного элемента в оракул, который мы передаем поисковому алгоритму.
❷ Выполняет поисковый алгоритм Гровера на реестре из трех кубитов и предоставляет оракула markItem, которого мы определили ранее.
❸ Выдает сообщение для проверки, что он нашел правильный элемент.

Если выполнить образец в листинге 11.1, то мы получим следующий ниже результат:

```
In [1]: %simulate RunGroverSearch nItems=7 idxMarkedItem=6
Out[1]: Промаркирован 6, и найден 6
```

Из выполнения примера алгоритма Гровера мы видим, что искомый ключ дешифровки был проиндексирован или промаркирован числом 6, и алгоритм также нашел ключ под номером 6. Теперь, поскольку опера-

ция `SearchForMarkedItem` действительно является основой этого примера, давайте рассмотрим ее имплементацию.

Листинг 11.2 operations.qs: написание алгоритма Гровера как операции Q#

```
operation SearchForMarkedItem(                                    (1)
    nItems : Int,                                                 (2)
    markItem : ((Qubit[], Qubit) => Unit is Adj)                  (3)
) : Int {                                                         (4)
    use qubits = Qubit[BitSizeI(nItems)];                         (5)
    PrepareInitialState(qubits);                                  (6)

    for idxIteration in                                           (7)
            0..NIterations(BitSizeI(nItems)) - 1 {
        ReflectAboutMarkedState(markItem, qubits);
        ReflectAboutInitialState(PrepareInitialState, qubits);
    }
    return MeasureInteger(LittleEndian(qubits));                  (8)
}
```

(1) Как обычно, мы начинаем с определения новой операции, используя ключевое слово «operation».
(2) Первое входное значение, необходимое для нашей операции, является числом элементов в списке.
(3) Следующее входное значение является представлением нашей поисковой задачи. Мы можем неявно определить нашу поисковую задачу с помощью оракула, который маркирует, когда элемент в нашем списке является правильным.
(4) По завершении поиска у нас будет индекс позиции, где находился маркированный элемент. Определение выходного значения как Int позволяет нам вернуть этот индекс.
(5) Для запуска поиска нам нужно выделить достаточно большой реестр для сохранения индекса в списке.
(6) Поскольку мы начинаем поиск в неструктурированном списке, все элементы одинаково хорошо подходят для поиска. Мы представляем это, подготавливая равномерную суперпозицию по всем индексам в списке.
(7) Суть алгоритма Гровера сводится к многократному отражению вокруг начального состояния и индексу искомого элемента.
(8) После завершения измерение реестра кубитов покажет нам индекс элемента, найденного алгоритмом Гровера.

Стандартная библиотека Q# предоставляет полезную операцию `MeasureInteger`, которая интерпретирует результаты измерений как классическое целое число. В целях использования `MeasureInteger`, как в листинге 11.2, можно промаркировать наш реестр как кодирующий целое число в реестре в представлении от младшего к старшему, используя пользовательский тип `Microsoft.Quantum.Arithmetic.LittleEndian`.

ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ Если вам нужно освежить свои знания о пользовательских типах (UDT) и их использовании, то рекомендуем ознакомиться с главой 9, где мы их применяли для того, чтобы помочь Ланселоту и Дагонету играть в свою игру на угадывание угла.

К этому месту в книге у нас есть почти все квантовые концепции, необходимые для понимания листинга 11.2. В остальной части данной главы мы научимся использовать то, что узнали, для имплементирования примера оракула, который может определять простую поисковую задачу, и имплементирования двух отражений, которые составляют алгоритм Гровера для решения этой задачи.

Дружище, у меня голова вразнос с твоим запросом!

Использование такой операции, как `ReflectAboutMarkedState`, в качестве оракула с целью определения запроса для алгоритма Гровера, возможно, покажется немного надуманным. В конце концов, поскольку индекс маркированного элемента «вживлен» в качестве входного значения для `ReflectAboutMarkedState`, может показаться, что в этом примере мы довольно сильно жульничаем. Тем не менее `SearchForMarkedItem` видит нашего оракула только как непрозрачный ящик и не видит его входных значений, вследствие чего вживление входного значения в таком ключе, как ни крути, не позволяет нам жульничать.

Использование такого рода простого оракула помогает нам сосредоточиться на принципе работы алгоритма Гровера, без необходимости понимать более сложного оракула. Однако на практике мы захотим использовать более сложного оракула, который представляет более сложную поисковую задачу. Например, для поиска в некоторых данных, представленных в виде списка элементов, мы могли бы использовать технологию под названием *квантовой RAM* (qRAM), чтобы превратить наш список в оракула. Подробности qRAM выходят за рамки этой книги, но в интернете есть несколько замечательных ресурсов о qRAM и о том, насколько дорого может стоить ее использование в конкретном приложении. Обратитесь по адресу https://github.com/qsharp-community/qram, где можно получить отличный вводный материал по qRAM и библиотеку Q#, с которой вы можете начать работу.

Еще одна область, в которой алгоритм Гровера широко используется, – это шифрование с симметричным ключом (в отличие от публичного ключа, который мы охватим в главе 12). Например, https://github.com/microsoft/grover-blocks предлагает имплементации оракулов, представляющих ключевые части шифров AES и LowMC в формате, который дает возможность понимать эти шифры, используя алгоритм Гровера.

11.2 Отражение вокруг состояний

В алгоритме 11.4 и листинге 11.2 мы неоднократно использовали две операции в цикле for: `ReflectAboutInitialState` и `ReflectAboutMarkedState`. Давайте разберемся, как эти операции помогают искать во входных значениях функции, представляющие наш сценарий расшифровки.

Каждая из этих операций является примером отражения вокруг конкретного состояния. Это пример нового вида квантовой операции, но

мы по-прежнему можем просимулировать ее с помощью унитарной матрицы, как и раньше. Термин «*отражение вокруг состояния*» (reflection about state) означает, что при наличии у нас реестра кубитов мы выбираем конкретное состояние, в котором он может находиться, и если он находится в этом состоянии, то мы переворачиваем знак состояния (меняем фазу этого состояния). Если вы думаете, что это похоже на некоторые из контролируемых операций, которые мы рассматривали ранее, то вы будете правы; мы будем использовать контролируемые операции для имплементирования этих отражений.

11.2.1 Отражение вокруг состояния «все единицы»

Давайте начнем с рассмотрения особенно полезного примера: отражения вокруг состояния «все единицы», $|11\cdots1\rangle$. Мы можем имплементировать это отражение, используя операцию `CZ` (контролируемого-`Z`), которую мы впервые увидели в главе 9.

> **ДЛЯ СПРАВКИ** Напомним, что функтор `Controlled` скорее может использоваться в суперпозиции, чем просто являться причудливым блоком `if`. Для обзора того, как работает функтор `Controlled`, рекомендуем ознакомиться с главой 9, где мы его использовали, чтобы помочь Ланселоту и Дагонету играть в свою игру.

Листинг 11.3 operations.qs: отражение вокруг состояния $|11\cdots1\rangle$

```
operation ReflectAboutAllOnes(register : Qubit[]) : Unit is Adj + Ctl {
    Controlled Z(Most(register),
        Tail(register));                    ❶
}
```

❶ Функтор Controlled позволяет нам использовать операцию Z в контролируемом стиле.

Как и другие контролируемые операции, `Controlled Z` принимает два входных значения: реестр, который должен использоваться в качестве контрольных кубитов, и кубит, к которому будет применена операция `Z`, если все кубиты в контрольном реестре находятся в состоянии $|1\rangle$. В листинге 11.3 мы можем использовать функцию `Most` из `Microsoft.Quantum.Arrays`, чтобы получить все кубиты, кроме последнего, и `Tail`, чтобы получить только последний кубит.

> **ДЛЯ СПРАВКИ** Используя `CZ` вместе с `Most` и `Tail`, наша имплементация работает независимо от того, сколько кубитов находится в нашем реестре. Это станет полезно позже, так как для представления данных в нашем списке может понадобиться разное число кубитов.

Вспомните из главы 9, что операция `CZ` применяет фазу -1 к состоянию $|11\cdots1\rangle$ и ничего не делает с любым другим вычислительно-базис-

ным состоянием. Возвращаясь к главе 2, где каждое вычислительно-базисное состояние является своего рода направлением, это означает, что отдельное направление переворачивается операцией CZ, в то время как все остальные входные состояния остаются нетронутыми. Именно поэтому мы называем операции, которые ведут себя в таком ключе, *отражениями*, хотя из-за числа размерностей, которые могут быть задействованы, сложно представить это графически.

Матричные представления операции CZ

Увидеть, что операция CZ переворачивает знак одного входного состояния, как описано ранее, можно, написав унитарную матрицу, которая симулирует CZ. Ниже приведен пример с одним контрольным кубитом:

$$U_{\mathrm{CZ}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

А вот пример для двух контрольных кубитов:

$$U_{\mathrm{CCZ}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}.$$

Используя то, что мы узнали об унитарных матрицах на протяжении всей этой книги, указанные матрицы дают понять, что входные состояния, соответственно, $|11\rangle$ и $|111\rangle$ переворачиваются в -1, в то время как все остальные входные состояния остаются нетронутыми (получают фазу +1). Один и тот же шаблон продолжает соблюдаться независимо от того, сколько контрольных кубитов мы используем с CZ.

Упражнение 11.1: диагностика операции CZ

Примените `DumpMachine`, чтобы увидеть, как CZ действует на равномерно-суперпозиционное состояние $|+\cdots\rangle$.

Подсказка: вспомните, что $|+\rangle = H|0\rangle$, и поэтому мы можем использовать программу `ApplyToEachCA(H, register)` для подготовки $|+\cdots+\rangle$ на реестре, который начинается в состоянии $|00\cdots0\rangle$.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

11.2.2 Отражение вокруг произвольного состояния

Как только у нас в кармане будет отражение вокруг $|11\cdots1\rangle$, мы сможем применять его и для отражений вокруг других состояний. Это важно, поскольку мы, вероятно, не сможем настроить нашу оракульную функцию, представляющую алгоритм дешифровки, так чтобы входное значение или ключ, который мы хотим, был представлен всеми входными данными. Также вспомните из нашего примера исходного кода, что оракул имплементирует дешифровку лишь в стиле игры на угадывание, а не реальный алгоритм дешифровки.

Глубокое погружение: отражения – это повороты?

Учитывая наше геометрическое понимание отражений, было бы естественно думать о них как о своего рода повороте на 180°. Это оказывается верным только потому, что квантовые состояния являются векторами комплексных чисел; если бы у нас был доступ только к действительным числам, то мы бы не смогли получить отражение с помощью поворотов! Мы можем убедиться в этом, вспомнив, как мы описывали состояния и повороты в главах 2 и 3: как повороты двумерного круга.

Если мы подберем двумерный объект, перевернем его и положим обратно, то не будет никакого способа преобразовать его обратно в то, каким он был, не подобрав его снова. С другой стороны, трехмерное пространство дает нам достаточно места для комбинирования разных поворотов, чтобы делать отражение. Поскольку комплексные числа дают нам третью ось при описании состояний кубитов (а именно ось Y), то это также позволяет нам делать отражения, которые нам нужны в алгоритме Гровера.

Хитрость отражений вокруг состояний, отличных от состояния «все единицы», заключается в том, чтобы превратить любое состояние, вокруг которого мы хотим выполнить отражение, в состояние «все единицы», вызвать `ReflectAboutAllOnes`, а затем откатить операцию, которую мы использовали для соотнесения нашего отражения в состояние «все единицы». Мы можем описать любое состояние, начав с состояния «все нули», поэтому нам нужен способ перейти от состояния «все нули» к состоянию «все единицы», где мы можем использовать отражение, которое мы только что узнали. В следующем ниже листинге показан пример подготовки реестра в состоянии «все единицы» из состояния «все нули».

Листинг 11.4 operations.qs: подготовка состояния «все единицы»

```
operation PrepareAllOnes(register : Qubit[]) : Unit is Adj + Ctl {
    ApplyToEachCA(X, register);              ❶
}
```

❶ Операция ApplyToEachCA позволяет нам применить первое входное значение (операцию) к каждому кубиту в реестре (второе входное значение).

В Q# все только что выделенные реестры начинаются в состоянии $|00\cdots0\rangle$. Следовательно, в листинге 11.4 при применении `X` к каждому только что выделенному кубиту эта операция подготавливает наш новый реестр в состоянии $|11\cdots1\rangle$.

Для следующего шага нам нужно рассмотреть операцию, подготавливающую состояние, вокруг которого мы хотим выполнить отражение. Если у нас есть адъюнктабельная операция (`is Adj`), которая подготавливает конкретное состояние, вокруг которого мы хотим выполнить отражение, то нам лишь нужно *откатить подготовку* (unprepare) состояния, подготовить состояние «все единицы», отразить вокруг состояния «все единицы» ($|11\cdots1\rangle$), откатить подготовку состояния «все единицы», а затем *заново подготовить* состояние, вокруг которого мы пытаемся выполнить отражение.

Почему мы любим нотацию Дирака

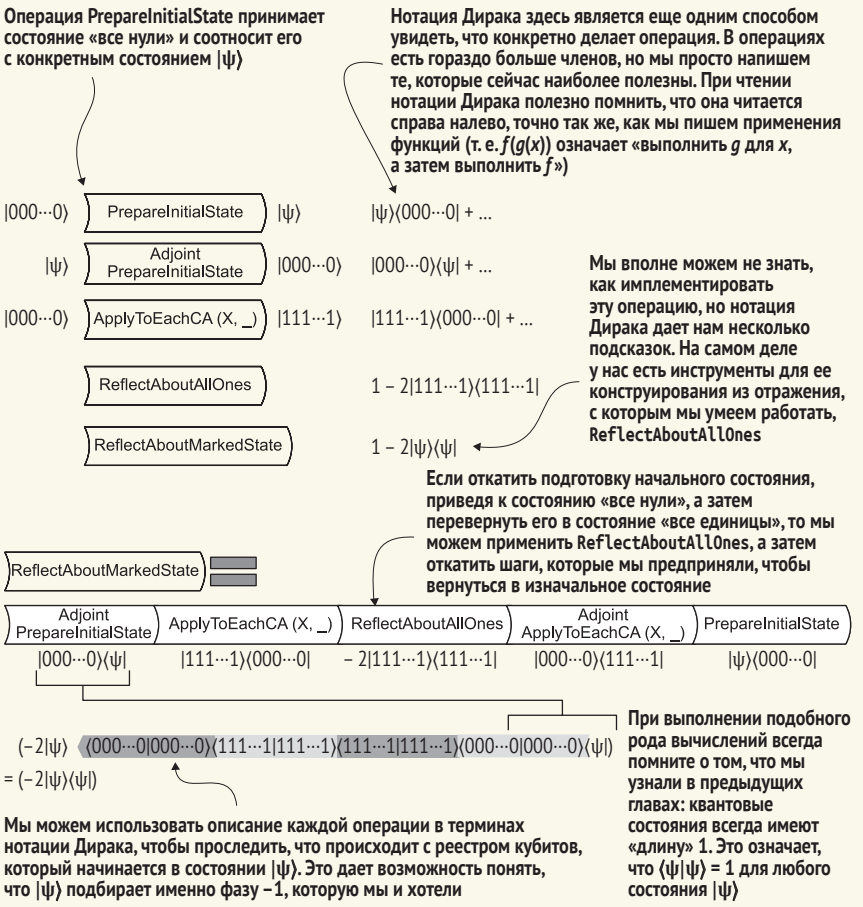
В понимании шагов алгоритма 11.5 полезно подумать о том, что каждая операция в этой последовательности шагов делает со своим входным состоянием. К счастью, нотация Дирака (впервые введенная в главе 2) помогает записывать то, как унитарные матрицы преобразовывают разные состояния, давая возможность понимать и предсказывать то, что соответствующие операции Q# будут делать с нашими кубитами.

Например, возьмем операцию Адамара `H`. Как мы видели на протяжении всей книги, `H` можно просимулировать с помощью следующей ниже унитарной матрицы:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Указанная унитарная матрица действует как своего рода таблица истинности, сообщая нам о том, что H преобразовывает состояние $|0\rangle$ в состояние $1/\sqrt{2}(|0\rangle + |1\rangle)$. Используя нотацию Дирака, мы можем сделать это яснее, написав $H = |+\rangle\langle 0| + |-\rangle\langle 1|$. Думая о кет-компонентах ($|\cdot\rangle$) как обозначающих входные значения и бра-компонентах ($\langle\cdot|$), мы можем прочитать это как «операция H преобразовывает $|0\rangle$ в $|+\rangle$ и $|1\rangle$».

На приведенном ниже рисунке показано то, как нотация Дирака действует как своего рода визуальный язык, который рассказывает нам о входных и выходных значениях для разных унитарных матриц, облегчая понимание принципа совместной работы последовательности операций Q#.



Разбиение отражений вокруг состояний с помощью нотации Дирака (aka бракетной нотации). Каждый шаг `ReflectAboutMarkedState` подразделен, и мы показываем нотацию Дирака для этой операции. Когда все отдельные шаги соединены, результирующее состояние равно $-2|\psi\rangle\langle\psi|$

Алгоритм 11.5: как выполнять отражение вокруг произвольного состояния

- Используя функтор `Adjoint`, «откатить» подготовку (un-prepare) произвольного состояния, соотнеся его с состоянием «все нули» $|00\cdots0\rangle$.
- Подготовить состояние «все единицы» $|11\cdots1\rangle$ из состояния «все нули».
- Использовать `CZ` для отражения вокруг $|11\cdots1\rangle$.
- Откатить подготовку состояния «все единицы», соотнеся его с состоянием «все нули».
- Снова подготовить состояние, соотнеся состояние «все нули» с произвольным состоянием.

ДЛЯ СПРАВКИ В алгоритме 11.5 шаги 1 и 5 делают откат адъюнктов друг друга, как и шаги 2 и 4. Используя то, что мы узнали о мышлении в стиле «ботинок и носков», это делает процедуру алгоритма 11.5 идеальной для имплементирования функциональности `within/apply` языка Q#! Для напоминания о том, как работает эта функциональность, рекомендуем ознакомиться с главой 8, в которой мы использовали блоки `within/apply` для имплементирования алгоритма Дойча–Йожи для Нимуэ и Мерлина.

Поскольку у нас нет никакого предварительного понимания того, что является правильным входным значением для нашего оракула при выполнении алгоритма Гровера, мы хотим начать наш поиск с равномерной суперпозиции $|+\cdots\rangle$, представив, что любое входное значение может быть правильным. Это дает нам возможность использовать то, что мы узнали из алгоритма 11.5, чтобы попрактиковаться в имплементировании отражений на языке Q#! Следуя шагам по отражению первоначального состояния, мы можем имплементировать операцию `ReflectAboutInitialState`, используемую в листинге 11.2. В следующем ниже листинге показано, каким образом можно следовать алгоритму 11.5 в операции Q#.

Листинг 11.5 operations.qs: отражение вокруг произвольного состояния

```
operation PrepareInitialState(register : Qubit[]) : Unit is Adj + Ctl {
    ApplyToEachCA(H, register);                                    ❶
}

operation ReflectAboutInitialState(
    prepareInitialState : (Qubit[] => Unit is Adj),                ❷
    register : Qubit[])                                            ❸
: Unit {
    within {                                                       ❹
        Adjoint prepareInitialState(register);                     ❺
        PrepareAllOnes(register);
    } apply {
        ReflectAboutAllOnes(register);
    }
}
```

❶ Равномерно-суперпозиционное состояние представляет, что у нас нет предварительной информации для нашего поиска (в конце концов, это задача неструктурированного поиска).
❷ Следуя алгоритму 11.5, в целях отражения вокруг первоначального состояния нам нужно предоставить операцию, которая его подготавливает.
❸ Конечно же, нам также нужен реестр кубитов, к которому будет применяться отражение!
❹ Выполняет шаги 1 и 2 из алгоритма 11.5.
❺ Функтор Adjoint указывает на то, что мы хотим выполнить «реверсию», или противоположную операцию, которая подготавливает первоначальное состояние. Другими словами, если бы мы начали с первоначального состояния, то применение операции Adjoint prepareInitialState вернуло бы нас в состояние $|00\cdots0\rangle$.

Теперь у нас есть исходный код для отражения вокруг этого начального состояния, и как проверить, что он делает то, что мы ожидаем? При выполнении целевой машины-симулятора мы можем применить такие команды, как `DumpRegister`, чтобы показать всю информацию, которую она использует для симулирования реестра кубитов. На рис. 11.4 показан результат на выходе из `DumpRegister` после подготовки равномерной суперпозиции.

Подготовка первоначальных состояний

In [6]:
```
open GroverSearch;
open Microsoft.Quantum.Diagnostics;

operation DumpInitialState() : Unit {
    using (register = Qubit[3]) {
        PrepareInitialState(register);
        DumpRegister((), register);
        ResetAll(register);
    }
}
```

Подобно тому, как мы делали в главе 8, мы можем применить диагностическую функцию `DumpRegister`, чтобы увидеть, что конкретно делает операция `PrepareInitialState`

Out[6]: • DumpInitialState

In [7]:
```
%simulate DumpInitialState
```

Qubit IDs	0, 1, 2		
Basis state (little endian)	Amplitude	Meas. Pr.	Phase
|0⟩	0.3536 + 0.0000*i*		↑
|1⟩	0.3536 + 0.0000*i*		↑
|2⟩	0.3536 + 0.0000*i*		↑
|3⟩	0.3536 + 0.0000*i*		↑
|4⟩	0.3536 + 0.0000*i*		↑
|5⟩	0.3536 + 0.0000*i*		↑
|6⟩	0.3536 + 0.0000*i*		↑
|7⟩	0.3536 + 0.0000*i*		↑

Out[7]: ()

При выполнении блокнот IQ# печатает, что наш реестр находится в равной суперпозиции восьми разных вычислительно-базисных состояний.

По умолчанию эти базисные состояния помечены как имеющие представление от младшего к старшему, поэтому дамп показывает нам каждое базисное состояние как целое число, а не как битовую строку.

Например, |6⟩ в представлении бит от младшего к старшему – это |011⟩ при записии в виде битовой строки

Рис. 11.4 Использование DumpRegister для просмотра первоначального состояния, подготовленного операцией PrepareInitialState. Каждое возможное базисное состояние имеет одинаковую амплитуду (и, следовательно, одинаковую вероятность измерения), которая называется равномерной суперпозицией

Для отражения вокруг маркированного состояния другого отражения, которое нам нужно для алгоритма Гровера, мы должны использовать несколько иной подход. В конце концов, мы не знаем, как готовить маркированное состояние – это именно та задача, для решения которой мы используем алгоритм Гровера! К счастью, возвращаясь к главе 8, мы можем применить то, что узнали из игры Нимуэ и Мерлина, чтобы имплементировать отражение вокруг состояния, *даже если мы не знаем, что это за состояние*.

ПРИМЕЧАНИЕ В этом суть алгоритма Гровера: мы можем применить нашего оракула для отражения вокруг маркированного состояния, вызвав его один раз с правильной суперпозицией входных зна-

чений. Как мы увидим в следующем далее разделе, каждое из этих отражений дает нам некоторую информацию о маркированном элементе. В отличие от этого, каждый вызов классической функции может исключать не более одного возможного входного значения.

В целях ознакомления с тем, как это работает в данном случае, давайте сначала сделаем шаг назад и посмотрим на то, каким является наше маркированное состояние. Поскольку наш список определяется оракулом, мы можем написать унитарный оператор O, который позволяет нам симулировать этого оракула. Ниже приведена нотация Дирака:

$$O|x\rangle \otimes |y\rangle = |x\rangle \otimes \begin{cases} |\neg y\rangle & x - \text{это отмеченный элемент} \\ |y\rangle & \text{в противном случае} \end{cases}.$$

В главе 8 мы узнали, что применение операции X к кубиту в $|-\rangle$ применяло фазу –1, поскольку $|-\rangle$ является собственным состоянием операции X. Используя тот же трюк здесь, мы можем написать, что наш оракул делает, когда флаговый кубит (реестр $|y\rangle$) находится в состоянии $|-\rangle$:

$$O|x\rangle \otimes |-\rangle = \begin{cases} -|x\rangle|-\rangle & x - \text{это отмеченный элемент} \\ |x\rangle|-\rangle & \text{в противном случае} \end{cases}.$$

Это именно та операция, которая нам нужна для имплементирования отражения! Таким образом, следуя тому, что мы узнали в главе 8, мы можем имплементировать его точно так же: просто применить наш алгоритм к кубиту, который начинается в состоянии $|-\rangle$. Тогда этот оракул будет представлять алгоритм расшифровки для нашего сценария, здесь упрощенный до функции, которая принимает возможные ключи на входе и просто возвращает булев результат, указывающий на правильность или неправильность ключа. В следующем ниже листинге показан пример операции Q#, в которой используется этот подход; напомним, что в Q# мы можем передавать операцию в качестве входного значения в еще одну функцию или операцию.

Листинг 11.6 операции.qs: отражение вокруг маркированного состояния

```
operation ReflectAboutMarkedState(
    markedItemOracle :                              ❶
        ((Qubit[], Qubit) => Unit is Adj),
    inputQubits : Qubit[])                          ❷
: Unit is Adj {
    use flag = Qubit();                             ❸
    within {
        X(flag);                                    ❹
        H(flag);
    } apply{
        markedItemOracle(inputQubits, flag);        ❺
    }
}
```

❶ Операция для нашего оракула маркировки элементов имеет тип ((Qubit[], Qubit) => Unit is Adj), указывающий на то, что он принимает реестр кубитов плюс один дополнительный кубит и может быть адъюнктным.
❷ Вторым входным значением для отражения вокруг маркированного состояния является реестр, к которому мы хотим применить наше отражение.
❸ Нам нужно выделить один дополнительный кубит (именуемый флаговым) для применения оракула, который соответствует реестру у в приведенных выше уравнениях.
❹ Тем же самым образом, как мы использовали операции H и X в главе 8 для подготовки кубита Нимуэ в состоянии |–⟩, мы используем, что |–⟩ = HX|0⟩, здесь, чтобы подготовить наш флаговый кубит.
❺ Мы применяем нашего оракула для использования трюка Дойча–Йожи и применим фазу –1 к состоянию, маркированному нашим оракулом.

Поскольку в листинге 11.6 эта подготовка находится в блоке `within/apply`, Q# автоматически переводит наш кубит за нас обратно в состояние |0⟩, откатывая операции `X` и `H`. В конце концов, как мы видели в главе 8, применение нашего оракула оставляет его цель в состоянии |–⟩.

Упражнение 11.2: подготовка флага

В листинге 11.6 мы также можем написать `H(flag); Z(flag);`. Использование пакета QuTiP и `AssertOperationsEqualReferenced` либо сразу обоих этих инструментов доказывает, что эти два способа подготовки флагового кубита дают нам одно и то же отражение.

Что удивительно, так это то, что мы использовали трюк Дойча–Йожи, чтобы выполнить отражение вокруг состояния, которое было *неявно* определено нашим оракулом! Нам не нужно было явно знать о том, каким было маркированное состояние, чтобы применить отражение: это идеально подходит для использования в неструктурированном поиске.

В следующем разделе мы рассмотрим способ комбинирования отражений первоначального и маркированного состояний, чтобы свести все вместе, полностью имплементировать алгоритм Гровера и отыскать наш ключ.

11.3 Имплементирование поискового алгоритма Гровера

Теперь, когда мы узнали о поворачивании состояний и провели ревизию оракулов, самое время окончательно собрать все это вместе, чтобы выполнить неструктурированный поиск! Давайте начнем с того, что еще раз посмотрим на все шаги по имплементированию алгоритма Гровера (рис. 11.5).

1 Выделить реестр кубитов, достаточно большой, чтобы индексировать набор данных, в котором мы выполняем поиск.

2. Подготовить реестр в равномерно-суперпозиционном состоянии: т. е. все возможные состояния имеют одинаковую амплитуду. Это связано с тем, что из-за типа задачи у нас нет никакой дополнительной информации о наборе данных, поэтому оно представляет собой равномерное вероятностное распределение (или априорное распределение) на данных.
3. Отразить реестр вокруг маркированного состояния или состояния, которое мы ищем.
4. Отразить реестр вокруг первоначального состояния (равномерной суперпозиции).
5. Повторять шаги 3 и 4 до тех пор, пока вероятность измерить искомый элемент не станет достаточно высокой. Затем измерить реестр. Мы можем математически вычислить оптимальное число раз, которое нам нужно это проделать, чтобы максимизировать вероятность вернуть маркированный элемент.

Как можно ближе, не перегибая палку

Если мы применим слишком много итераций алгоритма Гровера, то амплитуда состояния, которое мы хотим измерить, уменьшается. Это обусловлено тем, что каждая итерация фактически является поворотом; хитрость заключается в том, чтобы остановить этот поворот в нужной точке. В целях разработки тригонометрии для критериев остановки мы записываем состояние реестра, используемого в алгоритме Гровера, как суперпозицию немаркированных и маркированных состояний. Здесь мы не будем подробно останавливаться на демонстрации этого вывода, однако рекомендуем обратиться по адресу https://docs.microsoft.com/quantum/libraries/standard/algorithms или к разделу 6.1.3 книги «Квантовые вычисления и квантовая информация» Майкла А. Нильсена и Исаака Л. Чуанга (*Quantum Computation and Quantum Information*, Michael A. Nielsen and Isaac L. Chuang, Cambridge University Press, 2010), если вам интересно узнать больше о математике, используемой за кулисами. Она имплементирована для вас в репо образцов, прилагаемых к этой книге, но формула приведена ниже, если вы хотите попробовать запрограммировать ее на Q# самостоятельно:

$$N_{\text{iterations}} = \text{round}\left[\frac{\pi}{4\arcsin\left(\frac{1}{\sqrt{2^n}}\right)} - \frac{1}{2}\right].$$

В предыдущем разделе мы разработали несколько операций, необходимых для полной имплементации. Например, мы имплементировали шаг 2 с операцией `PrepareInitialState` и отражения в шагах 3 и 4 соответственно в качестве операций `ReflectAboutMarkedState` и `ReflectAboutInitialState`. Нам все еще нужна функция, которая поможет идентифицировать число раз, которые нужно повторять шаги 3 и 4, а также имплементация оракула, выявляющего элемент, который мы ищем. Да-

вайте начнем с функции, которая помогает определить критерии остановки алгоритма Гровера.

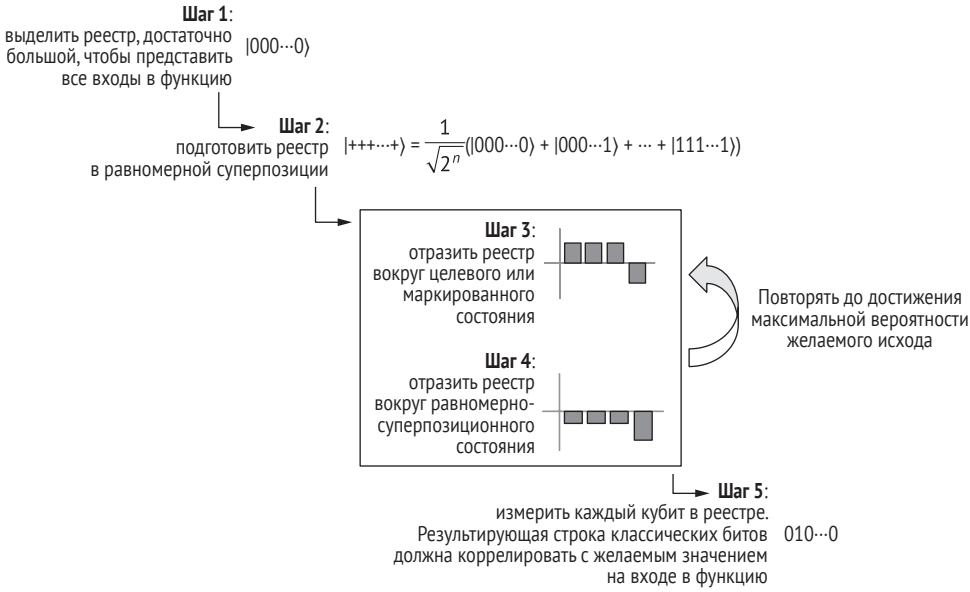


Рис. 11.5 Вспомните шаги алгоритма Гровера, которые выполняют поиск в данных, поступающих на вход функции, в поисках конкретного результата на выходе из функции. Мы начинаем с выделения реестра кубитов, достаточно большого для представления всех входных данных, в которых мы хотим выполнить поиск, а затем помещаем его в равномерно-суперпозиционное состояние. Наконец, мы можем отразить состояние реестра нужное число раз, чтобы максимизировать вероятность измерить ответ, который ищем

Листинг 11.7 operations.qs: критерии остановки для поискового алгоритма Гровера

```
function NIterations(nQubits : Int) : Int {
    let nItems = 1 <<< nQubits;                    ❶
    let angle = ArcSin(1. /
        Sqrt(IntAsDouble(nItems)));                ❷
    let nIterations =
        Round(0.25 * PI() / angle - 0.5);          ❸
    return nIterations;
}
```

❶ **<<< является оператором левого битового сдвига, используемым для вычисления 2nQubits, представляющей максимальное число элементов, которые могут быть проиндексированы квантовым реестром nQubits.**

❷ **Вычисляет эффективный угол поворота, применяемый каждой итерацией алгоритма Гровера.**

❸ **Используя эффективный угол поворота вместе с некоторой тригонометрией, мы можем вычислить число итераций, которые будут максимизировать вероятность того, что мы измерим маркированный элемент.**

Теперь, когда мы можем рассчитать, когда останавливать цикл в нашей имплементации алгоритма Гровера, последнее, что нам нужно, – это оракул, который может – имея искомый элемент и потенциальный элемент из набора данных – перевернуть фазу в части нашего реестра, если потенциальный элемент является искомым элементом. Для целей примера давайте вообразим, что оракул представляет своего рода игру на угадывание. Если кто-то загадывает 4 и просит нас угадать его число, то это пример своего рода классической функции:

$$f(x) = \begin{cases} 1 & x = 4 \\ 0 & \text{в противном случае} \end{cases}.$$

Классически у нас не было бы лучшей стратегии, чем пробовать разные входные данные для f, пока мы не попробуем $x = 4$. Если мы хотим вместо этого попробовать алгоритм Гровера, используя то, что мы узнали в главе 8, то мы знаем, что нам нужна операция, представляющая f:

$$U_f(|x\rangle \otimes |y\rangle) = |x\rangle \otimes \begin{cases} X|y\rangle & x = 4 \\ |y\rangle & \text{в противном случае} \end{cases}.$$

Имплементировать операцию, которую можно просимулировать с помощью U_f, довольно легко, используя функцию Q# `ControlledOnInt`, предлагаемую в рамках стандартных библиотек Q#. Как и функтор `Controlled`, функция `ControlledOnInt` позволяет нам контролировать операцию на состоянии еще одного реестра. Разница в том, что в то время как `Controlled` всегда контролирует состояние «все единицы» $|11\cdots1\rangle$, функция `ControlledOnInt` позволяет нам контролировать другое состояние, заданное целым числом. Например, если `Length(register)` равно 3, то `(ControlledOnInt(4, X))(register, flag)` переворачивает состояние флага всякий раз, когда реестр находится в состоянии $|100\rangle$, поскольку 4 записывается как 100 в представлении от младшего к старшему.

Упражнение 11.3: действие функции ControlledOnInt

Попробуйте написать, что конкретно `(ControlledOnInt(4, X))(register, flag)` делает с состоянием `register + [flag]` с помощью нотации Дирака (ознакомьтесь с главами 2 и 4, если вам нужно освежить свои знания) либо с помощью написания унитарной матрицы, которую можно использовать для симулирования `(Controlledonint(4, X))`, воздействующей на трехкубитовый реестр и флаговый кубит.

Подсказка: поскольку `(ControlledOnInt(4, X))` в данном примере воздействует на четыре кубита (три контрольных кубита и целевой кубит), унитарная матрица должна быть матрицей 16×16.

Попробуйте сделать то же самое, но для `(ControlledOnInt(4, Z))`.

Используя функцию `ControlledOnInt`, мы можем быстро написать оракула, который переворачивает состояние флагового кубита, основыва-

ясь на значении на входе в этот оракул, как показано в следующем ниже листинге. Здесь наш оракул должен перевернуть свой флаговый кубит всякий раз, когда значение на входе в оракул находится в маркированном состоянии.

Листинг 11.8 operations.qs: оракул, маркирующий нужное нам состояние

```
operation ApplyOracle(
    idxMarkedItem : Int,                    ❶
    register : Qubit[],
    flag : Qubit
) : Unit is Adj + Ctl {
    (ControlledOnInt(idxMarkedItem, X))
        (register, flag);                   ❷
}
```

❶ Задает индекс искомого элемента в виде целого числа (в данном примере используются три кубита, поэтому мы можем ввести любое целое число от 0 до $2^3 - 1 = 7$).
❷ Функция ControlledOnInt, о которой мы узнали ранее, может применить X к флагу, контролируемому на входном реестре, находящемся в правильном маркированном элементе.

Добавив эти два фрагмента исходного кода, мы можем вернуться к предыдущему образцу кода.

Листинг 11.9 operations.qs: алгоритм Гровера как операция Q#

```
operation SearchForMarkedItem(                          ❶
    nItems : Int,                                       ❷
    markItem : ((Qubit[], Qubit) => Unit is Adj)        ❸
) : Int {                                               ❹
    use qubits = Qubit[BitSizeI(nItems)];               ❺
    PrepareInitialState(qubits);                        ❻

    for idxIteration in                                 ❼
            0..NIterations(BitSizeI(nItems)) - 1 {
        ReflectAboutMarkedState(markItem, qubits);
        ReflectAboutInitialState(PrepareInitialState, qubits);
    }

    return MeasureInteger(LittleEndian(qubits));        ❽
}
```

❶ Как обычно, мы начинаем с определения новой операции, используя ключевое слово «operation».
❷ Первым входным значением является число элементов в нашем списке.
❸ Подобно примеру с криптографией, мы можем определить список неявно с помощью оракула, который маркирует, что элемент в нашем списке является правильным.
❹ В конце поиска у нас будет индекс позиции, где был маркированный элемент. Определение выходного значения в виде Int позволяет нам вернуть этот индекс.
❺ Для запуска поиска нам нужно выделить достаточно большой реестр для сохранения индекса в нашем списке.
❻ Поскольку мы выполняем поиск в неструктурированном списке, когда мы только начинаем поиск, все элементы одинаково хорошо подходят для поиска.

❼ Суть алгоритма Гровера сводится к многократному отражению вокруг начального состояния и индекса для искомого элемента.
❽ После завершения измерение реестра кубитов покажет нам индекс элемента, найденного алгоритмом Гровера.

Стандартная библиотека Q# предоставляет полезную операцию `MeasureInteger`, которая интерпретирует результаты измерений как классическое целое число. В целях использования MeasureInteger, как в листинге 11.10, можно промаркировать наш реестр как кодирующий целое число в реестре с кодировкой от младшего к старшему, используя пользовательский тип `Microsoft.Quantum.Arithmetic.LittleEndian`.

У нас есть весь нужный нам исходный код, поэтому давайте приведем пример.

Листинг 11.10 operations.qs: конкретный пример работы поискового алгоритма Гровера

```
operation RunGroverSearch(nItems : Int, idxMarkedItem : Int) : Unit {
    let markItem = ApplyOracle(
        idxMarkedItem, _, _);                          ❶
    let foundItem = SearchForMarkedItem(
            nItems, markItem);                         ❷
    Message(
        $"Промаркирован {idxMarkedItem}, и найден
    ➥ {foundItem}.");                                 ❸
}
```

❶ Мы можем использовать частичное применение, чтобы включить индекс маркированного элемента в оракула, который мы передаем в поисковый алгоритм.
❷ Выполняет алгоритм Гровера на реестре из трех кубитов и предоставляет оракулу элемент markItem, который мы определили ранее.
❸ Выдает сообщение для проверки, был им найден правильный элемент или нет.

Если выполнить этот пример, то мы получим следующий ниже результат:

```
In [1]: %simulate RunGroverSearch nItems=7 idxMarkedItem=2
Out[1]: Промаркирован 2, и найден 2.
```

Упражнение 11.4: изменение оракулов

Попробуйте изменить определение оракула, чтобы контролировать на другом целом числе. Изменит ли это выходные данные при выполнении алгоритма Гровера?

Поздравляем: теперь мы можем использовать квантовую программу для неструктурированного поиска! Но что на самом деле происходит? Ключевая идея из геометрии, которая заставляет алгоритм Гровера работать, заключается в том, что при отражении вокруг двух разных осей

мы получаем поворот. На рис. 11.6 показан пример того, как это работает для карт местности.

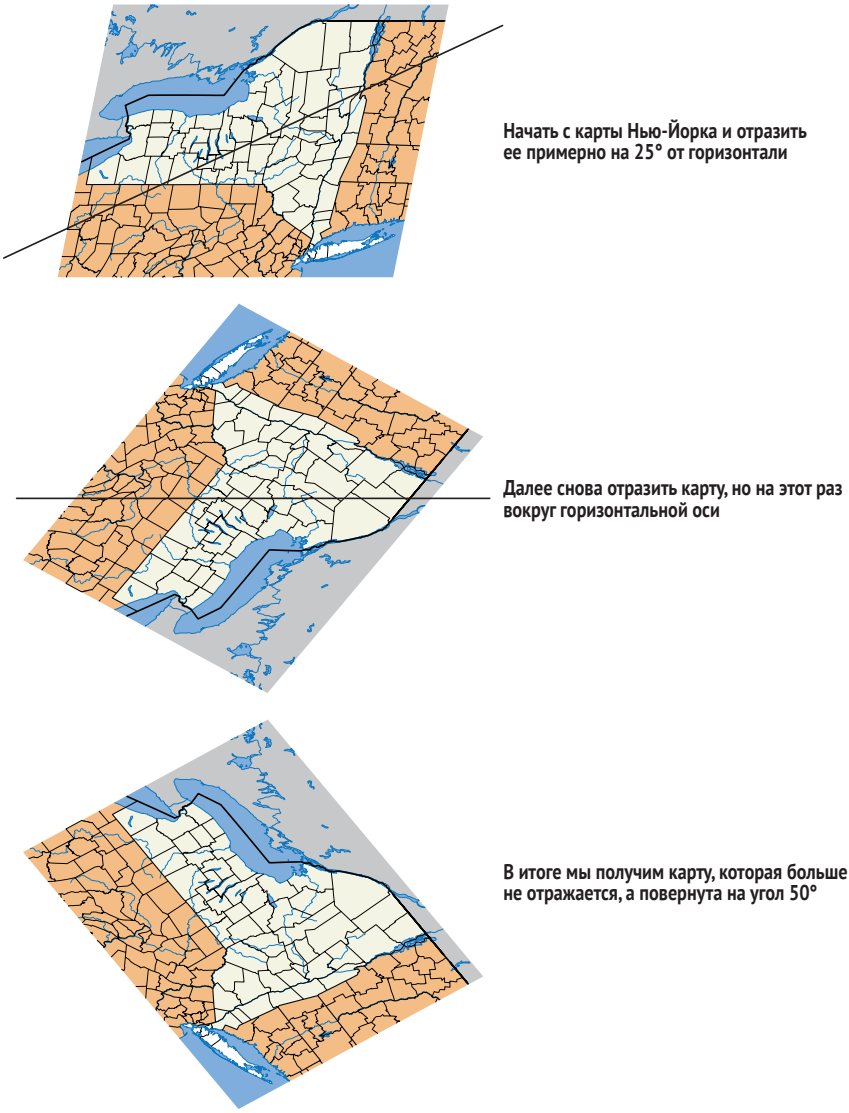


Рис. 11.6 Как пары отражений могут совершать поворот. Если мы отразим карту по линии на 25° от горизонтали, а затем отразим ее вокруг горизонтальной оси, то это будет то же самое, что и поворот карты на 50° вниз от горизонтали

Та же идея работает и для квантовых состояний. В алгоритме Гровера отражения первоначального и маркированного состояний комбинируются в один поворот из немаркированных состояний в маркированное состояние. В целях понимания принципа его работы мы можем использовать технические приемы, которые узнали на протяжении всей книги,

чтобы посмотреть, что происходит с амплитудами каждого состояния реестра по мере прохождения шагов алгоритма.

По рис. 11.7 видно, что каждый раунд отражений, по-видимому, *усиливает* амплитуду состояния, соответствующего искомому индексу. Поворачивая между немаркированными и маркированным состояниями, мы можем привести состояние наших кубитов в соответствие с маркированным состоянием, которое мы хотим найти.

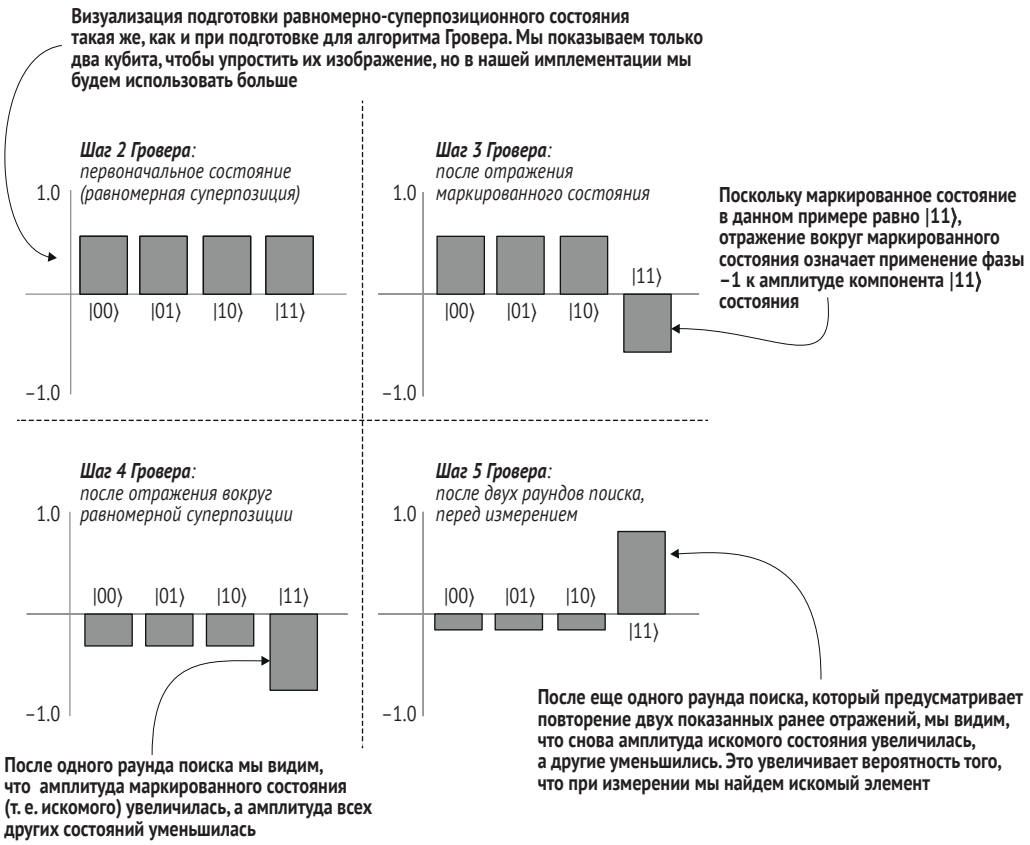


Рис. 11.7 Схема, показывающая, как изменяются амплитуды состояния нашего реестра кубитов по мере прохождения алгоритма Гровера. По мере того как мы продолжаем делать отражения, некоторые амплитуды усиливаются, а остальные уменьшаются

Как оказалось, мы можем использовать те же идеи и в других приложениях. Алгоритм Гровера является примером более широкого класса квантовых алгоритмов, которые выполняют то, что называется *амплитудным усилением*. Это означает, что мы значительно увеличили вероятность того, что при измерении нашего реестра кубитов результаты измерений в виде строки классических битов будут тем элементом, который мы ищем.

Прежде чем мы закроем эту главу, будет полезно кратко обсудить, как выполнение поиска, подобного тому, который мы только что имплементировали на квантовом оборудовании, масштабируется по сравнению с использованием классического оборудования.

Детерминированные и вероятностные квантовые алгоритмы

Алгоритм Гровера работает, увеличивая с каждой итерацией вероятность того, что мы получим правильный ответ. Однако в общем случае алгоритм Гровера, возможно, не сможет увеличить эту вероятность до 100 %. Отсюда алгоритм Гровера является примером вероятностного квантового алгоритма, что означает, что мы не получим гарантированного ответа, который ищем, всякий раз, когда мы его выполняем. На практике это не проблема, так как мы всегда можем выполнить его небольшое число раз, чтобы получить еще более высокую вероятность успеха.

Может показаться заманчивым заключить, что все квантовые алгоритмы в этом же смысле являются вероятностными, но это не так. Как мы видели в главе 8, алгоритм Дойча–Йожи является примером *детерминированного* квантового алгоритма, который дает одни и те же результаты всякий раз, когда мы его выполняем.

Упражнение 11.5: разведывательный анализ алгоритма Гровера с помощью функции DumpMachine

До этого места в книге мы много узнали о поворотах, что помогает понять поворот, применяемый каждой итерацией алгоритма Гровера. Попробуйте изменить имплементацию алгоритма Гровера, применив вдвое больше итераций, и используйте функцию `DumpMachine`, чтобы посмотреть на результирующее состояние. Похоже ли оно на то, которого вы ожидаете от двукратного применения поворота?

Более общие примеры амплитудного усиления

Наряду с фазовым оцениванием амплитудное усиление является одним из наиболее фундаментальных технических приемов, используемых в квантовых алгоритмах. За 25 лет, прошедших с тех пор, как алгоритм Гровера впервые ввел концепцию амплитудного усиления, было разработано огромное число вариантов, охватывающих широкий спектр разных задач, например когда имеется несколько маркированных элементов, когда мы хотим оптимизировать функцию, а не отыскать маркированный элемент, или даже когда мы только иногда можем правильно готовить первоначальное состояние. Многие из этих методов доступны в рамках пространства имен `Microsoft.Quantum.Amplitude.Amplification` в стандартных библиотеках Q#. Обязательно посмотрите!

11.4 Оценивание ресурсов

Ранее, когда мы упростили наш сценарий с 2.5 миллиона ключей до 8, мы упоминали, что использование алгоритма Гровера будет иметь преимущество по мере роста числа ключей, необходимых для поиска. Тогда сколько же времени требуется, чтобы выполнить алгоритм Гровера на практике? Это оказывается довольно сложным вопросом – мы могли бы написать об этом несколько книг. Отчасти данный вопрос обязан своей сложностью тому, что оценивание потребностей в ресурсах неизбежно зависит от множества различных частей нашего стека квантовых вычислений.

Например, ошибки встречаются в квантовых устройствах довольно часто, поэтому нам нужно использовать исправление ошибок для защиты наших вычислений во время их выполнения. То, какой метод исправления ошибок используется для защиты наших вычислений, оказывает огромное влияние на то, что требуется для выполнения нашей программы. Именно по этой причине целые конференции посвящены поиску более совершенных кодов исправления ошибок.

К счастью, язык Q# и Комплект инструментов для квантовой обработки предоставляют несколько инструментов, необходимых для того, чтобы начать разбираться в том, что конкретно требуется для выполнения разных квантовых программ. Вместо того чтобы выполнять нашу программу на симуляторе, представляющем работу реального квантового компьютера, мы можем выполнять ее на *оценщике ресурсов*, который сообщает нам о числе внутренних операций каждого типа, которые нам нужно вызвать, числе кубитов, которые требуются нашей программе, и числе квантовых операций, которые в нашей программе можно вызвать параллельно. Давайте рассмотрим небольшой пример, используя то, что мы узнали об алгоритме Дойча–Йожи в главе 8.

Листинг 11.11 Повторное определение алгоритма Дойча–Йожи

```
In [1]: operation ApplyNotOracle(control : Qubit, target : Qubit)
        : Unit {                                                    ❶
            within {
                X(control);
            } apply {
                CNOT(control, target);
            }
        }
Out[1]: - ApplyNotOracle
In [2]: open Microsoft.Quantum.Measurement;                         ❷

        operation CheckIfOracleIsBalanced(                          ❸
                oracle : ((Qubit, Qubit) => Unit)
        ) : Bool {
            use control = Qubit();
            use target = Qubit();
                H(control);
```

```
            within {
                X(target);
                H(target);
            } apply {
                oracle(control, target);
            }
            return MResetX(control) == One;
        }
Out[2]: - CheckIfOracleIsBalanced
In [3]: operation RunDeutschJozsaAlgorithm()
        : Bool {                                                    ❹
            return CheckIfOracleIsBalanced(ApplyNotOracle);
        }
Out[3]: - RunDeutschJozsaAlgorithm
```

❶ То же самое приложение, которое мы встречали ранее, за исключением того, что теперь в нем используется поток within/apply.
❷ Помните, что при использовании среды блокнотов Q# Jupyter Notebook мы должны открывать пространства имен в каждой ячейке, в которой мы хотим их использовать.
❸ Операция CheckIfOracleIsBalanced такая же, как и раньше, за исключением того, что блок within/apply снова используется для замены повторяющихся операций H и X.
❹ В блокнотах Q# операция без аргументов нам нужна для использования с командами `%simulate` и %estimate.

При выполнении волшебной команды `%estimate` в блокноте IQ# мы получаем таблицу, очень похожую на ту, что показана на рис. 11.8. В этой таблице представлены виды ресурсов, которые, по оценкам Комплекта инструментов для квантовой разработки, потребуются для выполнения нашей программы.

```
In [4]: %estimate RunDeutschJozsaAlgorithm
```

Out[4]:

Metric	Sum	Max
CNOT	1	1
QubitClifford	8	8
R	0	0
Measure	1	1
T	0	0
Depth	0	0
Width	2	2
QubitCount	2	2
BorrowedWidth	0	0

Когда вместо `%simulate` мы используем волшебную команду `%estimate`, IQ# печатает ресурсы, необходимые для выполнения нашей квантовой программы, в разбивке по каждому виду ресурса

Рис. 11.8 Результат выполнения команды %estimate RunDeutschJozsaAlgorithm для программы в листинге 11.11. Когда мы используем %estimate, мы получаем количество различных типов ресурсов, которые наше квантовое устройство (или симулятор) должно было бы предоставить, чтобы сделать возможным выполнение программы. Ознакомьтесь с табл. 11.1, чтобы узнать больше о значении каждого из этих показателей

Таблица 11.1 Виды ресурсов, отслеживаемых волшебной командой %estimate

Вид ресурса	Описание
`CNOT`	Сколько раз вызывается операция `CNOT`
`QubitClifford`	Сколько раз вызываются операции `X`, `Y`, `Z`, `H` и `S`
`R`	Сколько раз вызываются операции однокубитового поворота
`Measure`	Сколько раз вызываются операции измерения
`T`	Сколько раз вызывается операция `T`
`Depth`	Сколько операций `T` нужно вызвать подряд на одном кубите
`Width`	Сколько кубитов требуется нашей программе
`BorrowedWidth`	Сколько кубитов наша программа должна иметь возможность заимствовать (более продвинутый технический прием, чем мы описываем в этой книге)

ДЛЯ СПРАВКИ Мы также можем оценивать ресурсы из Python! Просто используйте метод `estimate_resources` вместо метода `simulate`, о котором мы узнали в предыдущих главах.

Как можно видеть из выполнения `%estimate` в блокноте и из табл. 11.1, некоторые категории, вероятно, имеют смысл, например ширина, число выполненных измерений и R для числа используемых однокубитовых поворотов. Другие являются новыми, например подсчет операций `T` и глубина. Операции `T` мы раньше не встречали, но это всего лишь еще один вид однокубитовых операций.

Познакомьтесь с мистером T

Как и большинство других операций, которые мы встречали в книге до этого, операция `T` может быть просимулирована унитарной матрицей:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & (1+i)/\sqrt{2} \end{pmatrix}.$$

То есть `T` – это поворот 45° (π/4) вокруг оси Z.

На операцию `T` можно взглянуть и по-другому, как на четвертый корень операции `Z`, которую мы встречали довольно часто. Поскольку 45° × 4 = 180°, если применить `T` четыре раза подряд, то это будет дорогостоящим способом применить операцию `Z` один раз.

В Q# операция `T` доступна как `Microsoft.Quantum.Intrinsic.T` и имеет тип `Qubit => Unit is Adj + Ctl`.

Упражнение 11.6: четыре T образуют Z

Примените команду `AssertOperationsEqualReferenced`, чтобы доказать, что четырехкратное применение операции `T` делает то же самое, что и однократное применение `Z`. Есть еще одна операция, `S`, которую можно рассматривать как квадратный корень из `Z` (поворот на 90° вокруг оси *Z*); убедитесь, что двухкратное применение `T` совпадает с однократным применением `S`.

Операция `T` делает несколько особенным и, следовательно, достойным столь пристального внимания при оценивании ресурсов то, что их дорого использовать с исправляющими ошибки методами, которые необходимы при работе на больших квантовых устройствах. Большинство операций, которые мы использовали до сих пор, являются частью *группы Клиффорда*: операций, которые проще использовать с исправлением ошибок. Как отмечалось ранее, подробное описание исправления ошибок здесь выходит за рамки данной книги; но если коротко, то чем больше у нас операций, которые не являются операциями Клиффорда, тем сложнее имплементировать нашу программу на оборудовании с исправленными ошибками. Отсюда крайне важно подсчитывать число «дорогих» операций (например, `T`) для работы на нашем доступном в настоящее время оборудовании.

ДЛЯ СПРАВКИ На высоком уровне число операций `T`, которые должны выполняться последовательно (т. е. которые не могут выполняться параллельно), является довольно хорошим приближением времени, которое потребуется квантовой программе для выполнения на квантовых компьютерах с исправленными ошибками. Об этом оценщик ресурсов сообщает в виде метрики глубины.

Итак, каковы типичные или исключительные значения для ресурсов, которые мы можем подсчитать с помощью оценщика ресурсов на Q#? Для такой простой программы, как `RunDeutschJozsaAlgorithm`, требуются очень скромные ресурсы. Однако, глядя на табл. 11.1, мы видим, что операции `T` уделяется большое внимание, поэтому давайте рассмотрим ее чуть-чуть подробнее, чтобы понять, что это за операция и почему она важна для оценивания ресурсов. На рис. 11.9 показаны результаты оценивания ресурсов, необходимых для вызова операции `CCNOT`, о которой мы узнали в главе 9.

Давайте посмотрим, что произойдет при выполнении `%estimate` на операции `CCNOT`! Для этого нам нужно обернуть вызов `CCNOT` во что-то, что выделяет реестр кубитов

`CCNOT`

In [1]:
```
operation EstimateCcnotResources() : Unit {
    using (register = Qubit[3]) {
        CCNOT(register[0], register[1], register[2]);
    }
}
```

Out[1]: • EstimateCcnotResources

In [2]:
```
%estimate EstimateCcnotResources
```

Out[2]:

Metric	Sum	Max
CNOT	10	10
QubitClifford	2	2
R	0	0
Measure	0	0
T	7	7
Depth	5	5
Width	3	3
BorrowedWidth	0	0

Результат `%estimate` может показаться немного удивительным, учитывая, что мы только что вызвали одну операцию! За кулисами оценщик ресурсов разложил наш вызов `CCNOT` на операции, которые легче выполнять на квантовых устройствах с исправленными ошибками

Рис. 11.9 Результаты оценивания ресурсов, необходимых для вызова `CCNOT`. Из исходного кода может показаться, что должна быть только одна операция; но на самом деле `CCNOT` раскладывается на более легко имплементируемые операции в зависимости от целевой машины

Упражнение 11.7: сброс реестров

Почему нам не нужно сбрасывать реестр кубитов, выделенных в `EstimateCcnot-Resources`, как показано на рис. 11.9?

Этот результат немного удивляет тем, что наша крошечная программа требует 10 операций `CNOT`, 5 однокубитовых операций и 7 операций `T`, хотя мы не вызываем ни одну из них напрямую. Как оказалось, очень трудно применять такие операции, как `CCNOT`, непосредственно в квантовой программе с исправленными ошибками. Поэтому оценщик ресурсов Q# сначала превращает нашу программу в нечто более близкое к тому, что на самом деле будет выполняться на оборудовании, используя вызовы более простых операций, таких как `CNOT` и `T`.

Упражнение 11.8: масштабирование операции T

Как изменяется число вызовов операции `T` по мере увеличения числа контрольных кубитов? Ответ в виде грубого тренда будет нормальным.

Подсказка: как мы видели ранее, операция контролируемого-NOT с произвольным числом кубитов может быть записана как `Controlled X(Most(qs), Tail(qs));` используя функции, предоставляемые пространством имен `Microsoft.Quantum.Arrays`.

Это очень пригодится, когда мы захотим оценивать ресурсы, требующиеся для выполнения программы, которая слишком велика для симулирования на классическом компьютере. На рис. 11.10 показаны результаты выполнения алгоритма Гровера на 20-кубитовом списке (около 1 миллиона элементов).

Если выполнить ее для различных размеров списка, то мы получим кривую, подобную показанной на рис. 11.11. Для нашего сценария с 2.5 миллиона ключей квантовое число шагов намного ниже, чем стоимость классического шага. Разумеется, это еще не вся история, поскольку каждый шаг на квантовом компьютере, вероятно, будет намного медленнее, чем соответствующий шаг на классическом компьютере; но это действительно хороший шаг к пониманию того, что конкретно потребуется для выполнения разных квантовых программ на практике.

Теперь мы научились комбинировать оракулы, о которых узнали в главе 7, и новый вид квантовой операции (отражения) для выполнения поиска по передаваемым в функцию входным данным. В данном сценарии это было полезно тем, что помогает быстрее отыскивать ключи дешифровки при жестком ограничении по времени.

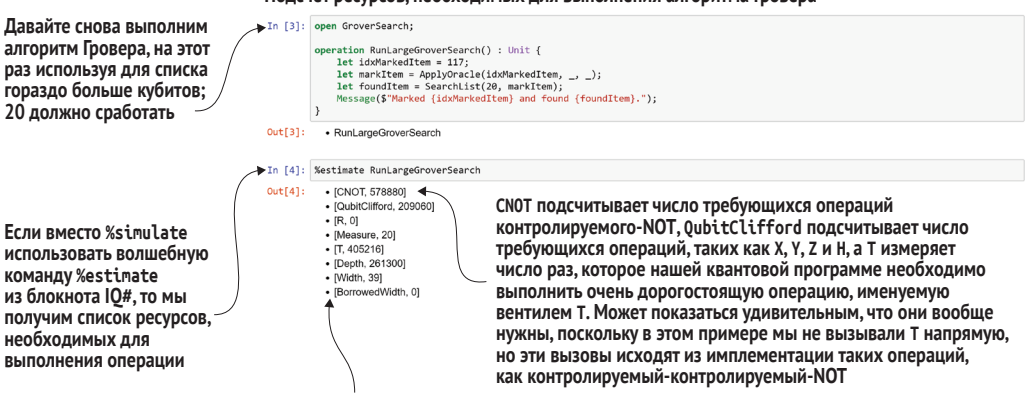


Рис. 11.10 Результат выполнения оценщика ресурсов на алгоритме Гровера. Эти подсчеты ресурсов дают понять причину, почему мы не можем просимулировать экземпляр алгоритма Гровера такого размера напрямую, поскольку нам потребуется 39 кубит. Однако мы можем использовать эти данные из нескольких размеров поиска, чтобы получить представление о том, как будет масштабироваться наша имплементация поискового алгоритма Гровера.

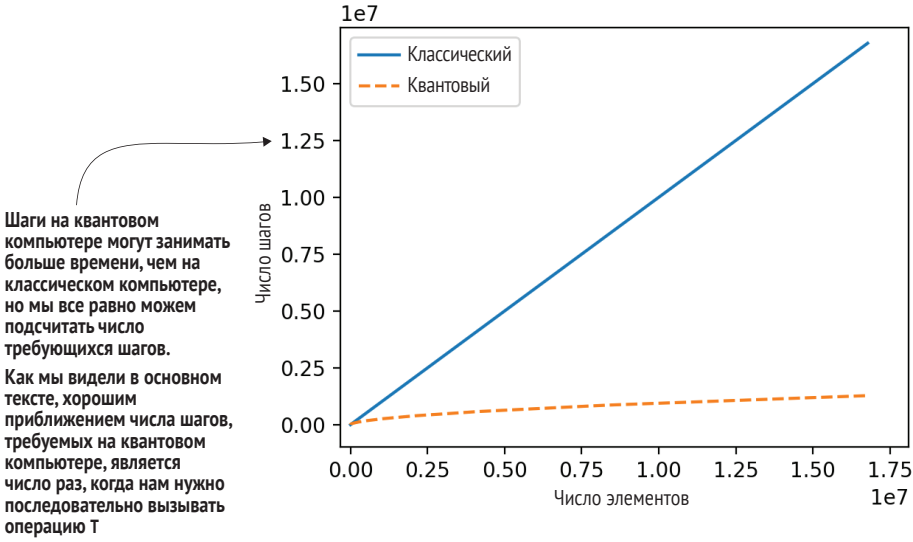


Рис. 11.11 Результаты оценивания ресурсов, необходимых для выполнения алгоритма Гровера для множества списков разного размера

В следующей главе мы воспользуемся навыками из данной главы, чтобы ответить на один из самых важных вопросов, поставленных квантовыми вычислениями: сколько времени потребуется квантовому компьютеру, чтобы взломать современное шифрование?

Резюме

- Еще одно приложение для квантовых компьютеров представлено поиском во входных данных, поступающих в непрозрачную функцию, входного значения, которое производит желаемый результат на выходе (т. е. *маркированного* входного значения). Мы можем использовать алгоритм Гровера для поиска, применяя меньшее число вызовов нашего оракула, чем это возможно при классических вычислениях.
- Алгоритм Гровера использует отражения, квантовые операции, в которых фаза одного входного состояния переворачивается, в то время как все остальные входные состояния остаются нетронутыми. Мы можем создавать самые разные виды отражений, используя повороты вместе с шаблонами «ботинки и носки», с которыми мы познакомились в главе 8.
- Используя различные повороты, предоставляемые языком Q# вместе с `within` и `apply`, мы можем определять оракула, который маркирует конкретный элемент, а затем отражать этот промаркированный элемент с помощью одного оракульного вызова. Все вместе, эти методы позволяют нам имплементировать алгоритм Гровера.
- В целях проверки того, что алгоритм Гровера превосходит классические подходы для достаточно больших задач, мы можем выполнить нашу программу Q# на оценщике ресурсов. В отличие от симулятора, поставляемого с Комплектом инструментов для квантовой разработки, оценщик ресурсов не симулирует квантовые программы, а подсчитывает число требующихся им кубитов и число операций, которые им потребуется для вызова квантового устройства.

12 Арифметика с помощью квантовых компьютеров

Эта глава охватывает следующие ниже темы:

- программирование с помощью численной библиотеки Q# Numerics;
- имплементирование алгоритма Шора для факторизации целых чисел;
- осознание последствий квантовых вычислений для инфраструктуры безопасности.

В главе 11 мы применили технический прием квантового программирования, именуемый амплитудным усилением, в алгоритме Гровера, чтобы ускорить поиск по неструктурированным наборам данных. В то время как подход Гровера не был самым эффективным подходом к поиску в небольших наборах данных, по мере того как мы стали обращаться к масштабированию до все больших и больших задач, наш квантовый подход давал явное преимущество. В этой заключительной главе мы будем опираться на навыки, которые развивали на протяжении всей книги, чтобы справиться с одним из самых известных квантовых алгоритмов: алгоритмом Шора. Мы выполним имплементацию алгоритма Шора и покажем, как он может давать нам преимущество при попытке разложить на факторы большие целые числа. Хотя эта задача, возможно, покажется не самой интересной, сложность факторизации целых чисел на самом деле лежит в основе большей части нашей нынешней криптографической инфраструктуры.

12.1 Включение квантовых вычислений в обеспечение безопасности

В части I книги мы рассмотрели способы применения квантовых концепций для безопасной передачи данных с использованием таких методов, как квантовое распределение ключей. Однако даже в отсутствие квантового распределения ключей критические данные все время тайно передаются через интернет. Интернет используется для обмена платежными данными, личными медицинскими сведениями, предпочтениями в свиданиях и даже для организации политических движений. В этой главе мы рассмотрим способы защиты нашей частной жизни классическими компьютерами и то, как квантовые вычисления меняют наши решения о выборе инструментов, используемых для защиты наших данных.

> **ПРИМЕЧАНИЕ** Теперь, когда у нас есть все необходимое для решения более сложных задач с помощью квантовых компьютеров, сценарий этой главы будет немного сложнее, чем большинство наших предыдущих игр и сценариев. Не волнуйтесь, если что-то будет не вполне понятно с места в карьер: не торопитесь и читайте медленнее. Мы обещаем, что это будет стоить вашего времени!

Для начала давайте рассмотрим современное состояние защиты данных с помощью классических компьютеров. Как оказалось, в классической математике существует много разных задач, некоторые из которых решаются элементарно (например, «каким будет результат 2 + 2?»), тогда как другие решаются с превеликим трудом (например, «Равен ли класс P классу NP?»). Между этими двумя крайностями мы сталкиваемся с задачами, которые решаются с трудом, если кто-то не даст нам подсказки, и в этом случае они становятся легкими. Эти задачи, как правило, больше похожи на *головоломки* и бывают полезны для сокрытия данных: мы должны знать секретную подсказку или использовать огромное количество вычислительного времени, чтобы их решить.

В главе 3 мы рассмотрели квантовое распределение ключей, которое является отличным способом безопасного обмена информацией, основанным не на головоломках, а на квантовой механике. Однако мы не всегда можем отправлять кубиты нашим друзьям, поэтому понимание принципа использования головоломок для безопасного и приватного общения по-прежнему имеет значение.

Как мы увидим далее в этой главе, факторизация чисел бывает одной из таких головоломок, на которые криптографические алгоритмы могут опираться для обеспечения безопасности. В настоящее время используется ряд очень важных алгоритмов и криптографических протоколов, которые основаны на том факте, что компьютерам трудно решать головоломки, связанные с факторизацией больших чисел. Если вы догадались, что квантовые компьютеры содействуют нам в разложении больших чисел на факторы, то вы – на правильном пути.

Поприветствуем *алгоритм Шора*. С помощью классического компьютера мы можем свести задачу или головоломку поиска факторов целых чисел к решению своего рода головоломки о том, как быстро функции повторяются при использовании *модульной арифметики* (также именуемой *часовой арифметикой*, как мы увидим позже в этой главе). Если мы используем алгоритм Шора, то оценивание того, как быстро функции повторяются, является именно той головоломкой, которую мы можем легко решать на квантовом компьютере. Давайте рассмотрим шаги алгоритма Шора подробнее, а затем обратимся к примеру его использования:

> *Сценарий: факторизация целого числа N*
>
> *Предположим, мы пытаемся разложить целое число N на факторы и заранее знаем, что N имеет ровно два простых фактора. Используя Q#, имплементировать алгоритм Шора для факторизации числа N.*

Взаимно простое и полупростое число

В качестве полезной терминологии мы говорим, что два числа, у которых нет общих факторов, помимо 1, являются *взаимно простыми*. Например, ни 15, ни 16 не являются простыми, но 15 и 16 являются взаимно простыми по отношению друг к другу.

Аналогичным образом мы говорим, что число ровно с двумя простыми факторами является *полупростым*. Например, 15 является полупростым, поскольку $15 = 3 \times 5$ и поскольку 3 и 5 являются простыми. С другой стороны, 28 не является полупростым, поскольку $28 = 4 \times 7 = 2 \times 2 \times 7$. При рассмотрении криптографии часто возникают полупростые числа, поэтому нередко бывает полезно принимать это допущение в наших сценариях.

Мы можем выполнить шаги алгоритма 12.1 (показанные в виде блок-схемы на рис. 12.1), чтобы применить то, что мы узнали о фазовом оценивании в главах 9 и 10, вместе с несколькими классическими математическими расчетами, чтобы найти факторы числа *N*.

Отряд mod

В алгоритме 12.1 нам нужен еще один кусочек классической математики: оператор mod. Если этот оператор вы раньше не встречали, то не волнуйтесь; мы рассмотрим его подробнее позже в данной главе.

Алгоритм 12.1: псевдокод факторизации целого числа с помощью алгоритма Шора

1. Выбрать случайное целое число g, которое мы называем *генератором*.
2. Проверить, что генератор нечаянно оказался фактором, выяснив, что g и N являются взаимно простыми числами. Если у них есть

общий фактор, то у нас есть новый фактор числа N; в противном случае продолжить с остальной частью алгоритма.

3. Применить итеративное фазовое оценивание, чтобы найти *частоту* классической функции $f(x) = g^x \bmod N$. Частота говорит о том, как быстро f возвращается к тому же значению по мере увеличения x.
4. Применить классический алгоритм, именуемый *разложением непрерывных дробей*, для конвертирования частоты из предыдущего шага в период (r). Тогда период r должен обладать свойством, что $f(x) = f(x + r)$ для всех входных значений x.
5. Если найденный период r является нечетным, то вернуться к шагу 1 и выдвинуть новую догадку. Если r является четным, то перейти к следующему шагу.
6. $g^{r/2} - 1$ либо $g^{r/2} + 1$ имеет общий фактор с числом N.

ПРИМЕЧАНИЕ В алгоритме 12.1 важно отметить, что только шаг 3 предусматривает какие-либо квантовые вычисления. Большинство шагов алгоритма Шора лучше всего подходят для классического оборудования и демонстрируют то, как, скорее всего, будет использоваться квантовое оборудование. То есть квантовое оборудование и алгоритмы хорошо работают в качестве *подпрограмм* для комбинированных квантово-классических алгоритмов.

Теперь, когда мы рассмотрели шаги алгоритма Шора, в листинге 12.1 показан возможный вариант окончательной его имплементации. Операция `FactorSemiprimeInteger` является точкой входа в алгоритм: на входе она принимает целое число, которое мы хотим факторизовать, и возвращает два его фактора.

Листинг 12.1 Исходный код Q# для факторизации полупростого целого числа

```
operation FactorSemiprimeInteger(number : Int) : (Int, Int) {
    if (number % 2 == 0) {                                          (1)
        Message("Было дано четное число; 2 является фактором.");
        return (number / 2, 2);
    }
    mutable factors = (1, 1);
    mutable foundFactors = false;

    repeat {
        let generator = DrawRandomInt(3,number - 2);                (2)

        if (IsCoprimeI(generator, number)) {
            Message($"Оцениваем период числа {generator}...");
            let period = EstimatePeriod(generator, number);         (3)
            set (foundFactors, factors) = MaybeFactorsFromPeriod(   (4)
                generator, period, number
            );
        } else {
            let gcd = GreatestCommonDivisorI(number, generator);
```

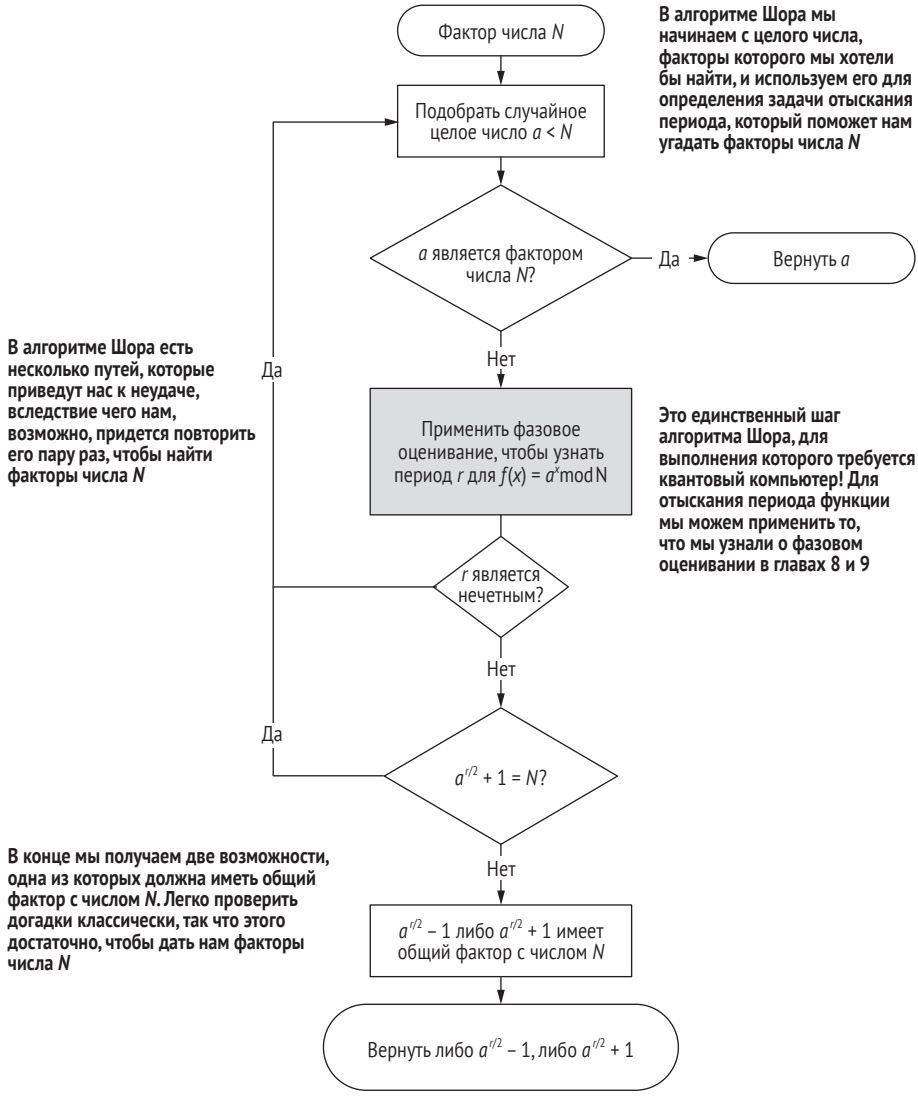


Рис. 12.1 Алгоритм Шора, представленный в виде блок-схемы. В целях разложения целого числа *N* на факторы в алгоритме Шора используется фазовое оценивание и квантовый компьютер, чтобы отыскать период функции, которая берет степени другого целого числа *a*, используя модульную арифметику mod *N*. После небольшой классической постобработки этот период можно использовать для отыскания факторов числа *N*

```
Message(
    $"Мы угадали, что делитель числа {number} равен " +
    $"{gcd}, нечаянно. Больше нечего делать."
);
set foundFactors = true;
set factors = (gcd, number / gcd);
```

```
        }
    }
    until (foundFactors)                                          ❺
    fixup {
        Message(
            "Оцененный период не дал валидного фактора, " +
            "пробуем еще раз."
        );
    }
    return factors;                                               ❻
}
```

❶ Сначала проверяет, что мы получили запрос на факторизацию четного числа, поскольку тогда 2 должно быть фактором.
❷ Следуя шагу 1 алгоритма 12.1, мы подбираем случайное число, чтобы определить периодическую функцию, которую мы используем для факторизации числа number.
❸ В этой главе мы учимся писать операцию EstimatePeriod для обработки шагов 3 и 4 алгоритма 12.1, используя то, что узнали о фазовом оценивании.
❹ Получив период, мы можем применить шаги 5 и 6 алгоритма 12.1, чтобы угадать факторы числа number; мы напишем функцию MaybeFactorsFromPeriod позже в этой главе.
❺ Если что-то пойдет не так (например, наш генератор имеет нечетный период), то мы используем цикл repeat/until, чтобы повторить попытку.
❻ Возвращает два фактора числа number, которые мы нашли с помощью нашей квантовой программы.

Удача – это еще не все, но она помогает!

В листинге 12.1 мы используем функцию `IsCoprimeI`, чтобы проверить, что генератор generator является фактором числа number, перед тем как приступить к остальной части алгоритма Шора. Если нам действительно повезет, то генератор уже является фактором, и в этом случае нам не нужен квантовый компьютер, чтобы помочь факторизовать число.

Хотя нам довольно часто будет везти в малых примерах, которые мы можем просимулировать на ноутбуке или настольном компьютере, по мере увеличения числа становится все труднее и труднее нечаянно угадывать правильные факторы, вследствие чего алгоритм Шора действительно полезен почти во всех случаях.

Поскольку эта глава книги является последней, у нас есть все *квантовые концепции*, необходимые для понимания того, что происходит в листинге 12.1; не хватает только классических частей, которые связывают то, что мы узнали до сих пор, с задачей факторизации полупростых чисел, а также нескольких полезных частей библиотек Q#, которые нам помогут. Как упоминалось ранее, квантовая технология здесь используется лишь в одном шаге, и это делается путем создания оракула, имплементирующего классическую функцию, о которой мы хотим узнать. Используя суперпозиционное состояние, применяя оракула и выполняя фазовое оценивание, мы можем узнать свойства классической функции: здесь это период. В остальной части главы мы подробно рассмотрим алгоритм 12.1 и обратимся к последним частям, которые нам нужны для

выполнения алгоритма Шора. Первая часть, которая нам нужна, чтобы понять алгоритм 12.1, связана с элементом классической математики, именуемым *модульной арифметикой*, так что давайте перейдем к делу!

12.2 Подключение модульной математики к факторизации

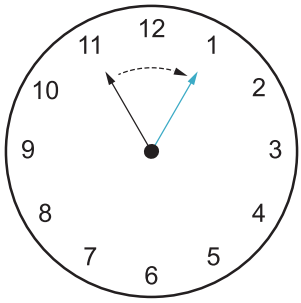
Отыскать головоломки, которые можно использовать в контексте безопасности, можно, если посмотреть на то, как работает *модульная арифметика*. В отличие от обычной арифметики, в модульной арифметике все возвращается по кругу к началу, как часовые промежутки времени на циферблате часов. Например, если кто-то нас спросит, который час будет через два часа после 11, то мы будем выглядеть как-то странно, если ответим «13 часов». Гораздо вероятнее, человек надеялся получить ответ типа «1 час», т. е. если он не использует 24-часовой формат времени.

Используя модульную арифметику, мы можем уловить эту идею, сказав, что $11 + 2 = 1 \bmod 12$. В этом уравнении $\bmod 12$ указывает на то, что мы хотим, чтобы все, что проходит за пределами 12, отсчитывалось по кругу с начала, как показано на рис. 12.2.

Предположим, что в 11 часов кто-то нас спрашивает, сколько времени будет через два часа.

Если бы мы сказали «13 часов», то выглядели бы как-то странно, но если бы мы сказали «1 час», то это было бы намного полезнее!

(11 + 2) можно рассматривать как «отсчет по кругу с начала» при наступлении 12 часов; это пример модульной арифметики



Конечно, мы не обязаны выполнять модульную арифметику только с 12-часовыми периодами. Если бы часы имели, например, 21 час, то, используя модульную арифметику, $5^2 = 25 = 4$

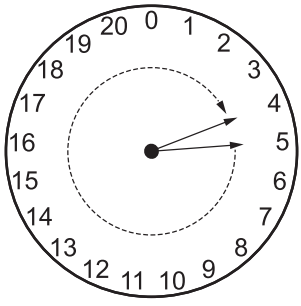


Рис. 12.2 Использование часов для понимания модульной арифметики. При сложении и умножении чисел по «mod *N*» на нормальную числовую ось можно смотреть как на обернутую вокруг циферблата с *N* часами. Подобно тому, как два часа после 11 часов является 1-м часом, $11 + 2 = 1$ при использовании $\bmod 12$

Когда арифметика может отсчитываться по кругу с начала в таком ключе, бывает трудно понять, где начались разные вычисления. Например, если мы работаем с обычными действительными числами, то вычислить b легко, если даны a и a^b; мы можем взять логарифм a^b, чтобы найти b. Если мы попытаемся решить ту же задачу в модульной арифметике, то это может быстро усложниться. Например, степени 5 при вычислении $\bmod 21$ равны $1, 5, 4, 20, 16, 17, 1, \ldots$. На первый взгляд, 5, 4 и 16 не кажутся степенями одного и того же числа, тем более в возрастающем порядке, вследствие чего при движении в обратном направлении от взятия экспоненты $\bmod 21$ у нас есть больше возможных начальных мест, которые нам нужно проверить.

Упражнение 12.1: степени числа 11

Каковы степени числа 11 при вычислении $\bmod 21$? Сколько операций требуется, чтобы вернуться назад к $11^0 = 1$?

Имеет ли значение, берете ли вы модуль по числу 21 в конце либо вычисляете модуль на каждом шаге?

Подсказка: и Python, и Q# отлично для этого подходят, так как в обоих языках определен модульный оператор %.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

Наблюдение, что найти степень b, заданной $a^b \bmod N$, трудно, уже дает нам головоломку, которую мы можем использовать, чтобы скрыть какие-то данные! Указанную головоломку принято называть *задачей о дискретном логарифме*. Если Алиса хочет поделиться с нами секретом, то мы можем начать с публичного согласия о малом числе, таком как $g = 13$, и большом числе, таком как $N = 71$. Затем каждый из нас выбирает секретное число наугад: предположим, Алиса выбирает $a = 4$, а мы выбираем $b = 5$. Затем Алиса отправляет нам $g^a \bmod N = 19$, а мы отправляем обратно $g^b \bmod N = 34$. Если мы вычислим $(g^a)^b \bmod N = 19^5 \bmod 71 = 45$, а Алиса вычислит $(g^b)^a \bmod 71 = 344 \bmod 71 = 45$, то мы оба получим одно и то же число, но злоумышленнику, чтобы ее решить, придется решить головоломку с прыжками по циферблату, которую мы видели ранее (см. рис. 12.3). Поскольку $g^{ab} = 45$ является числом, которое мы с Алисой знаем и никто другой не знает, мы можем использовать g^{ab} в качестве ключа, чтобы скрыть наши сообщения, используя то, что мы узнали в главе 3.

Шаг 1: вы и Алиса каждый думаете о секретном случайном числе и согласились использовать совместное **публичное** число g

Шаг 2: вы и Алиса каждый вычисляете g, возведенное в степень ваших секретных чисел, и объявляете результат

Шаг 3: используя результат, который вы получаете от Алисы вместе со своим собственным секретным числом, вы можете вычислить степень g^{ab}. Алиса может сделать то же самое, используя то, что она знает

Шаг 4: злоумышленник, который подслушивает сообщения g^a и g^b из шага 2, нуждается в решении головоломки о дискретном логарифме, чтобы получить тот же самый ответ, который вы и Алиса выяснили на шаге 3

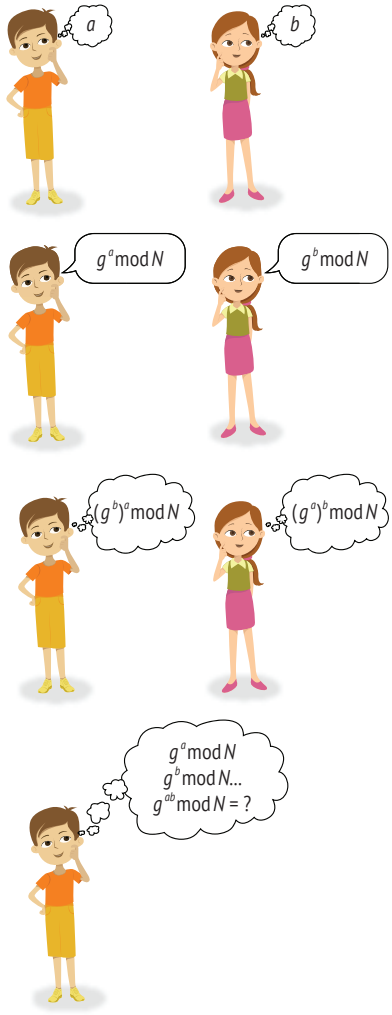


Рис. 12.3 Использование задачи о дискретном логарифме в качестве головоломки для сокрытия секретных сообщений. Здесь сообщения, которыми мы делимся с Алисой, защищены, в случае если кому-то трудно вычислить откат модульных арифметических операций, таких как экспоненциальная функция

Предупреждение: даже не пытайтесь делать это дома

Многие технические условия этого протокола выходят далеко за рамки данной книги. Неправильный выбор g и N может подорвать любую безопасность, предлагаемую этим методом, делая его тривиальным для опытного злоумышленника. Также очень легко ввести дефекты, выполняя его самостоятельно, поэтому, пожалуйста, рассматривайте его исключительно как концептуальный пример!

Если вы заинтересованы в том, чтобы узнать больше о практических аспектах использования таких головоломок для обеспечения безопасности ваших данных, то рекомендуем вам отличную книгу «Конструирование криптографии» Нильса Фергюсона, Брюса Шнайера и Тадаеси Коно (*Cryptography Engineering*, Niels Ferguson, Bruce Schneier, Tadayoshi Kohno, Wiley, 2010), от которой можно отталкиваться, продолжая обучение.

В отличие от квантового распределения ключей (QKD), этот способ обмена секретными данными (именуемый *протоколом Диффи–Хеллмана*) опирается на допущение, что головоломку, которую мы с Алисой использовали, трудно решить без подсказки, к которой наш злоумышленник не имеет доступа. Если кто-то может эффективно решать головоломки, такие как решение $g^a \bmod N$ для заданных g и N, то это все равно, если бы наши данные были публичными.

Еще одна головоломка, которая сегодня обычно используется для защиты данных, представлена *алгоритмом RSA*, в котором используется более продвинутая классическая математика, строя головоломку из факторизации действительно больших целых чисел. Подобно тому, как мы можем взломать алгоритм Диффи–Хеллмана, решив $g^a \bmod N$, мы можем взломать RSA, решив уравнение $N = pq$ для p или q, имея только N. В головоломке с RSA мы называем N *публичным ключом*, а факторы p и q – *приватным ключом*; если мы можем легко разложить N на факторы, то можем получить доступ к приватным ключам, имея только публичные ключи. С учетом этого мы можем сделать описанный ранее сценарий чуть-чуть точнее:

Сценарий: взлом RSA

Предположим, что мы знаем публичный ключ N. Используя Q#, имплементировать алгоритм Шора, чтобы разложить N на факторы для восстановления приватных ключей p и q.

Ломай, пока не сделаем

Этот сценарий может показаться немного … злонамеренным по сравнению с предыдущими главами. На практике, однако, важно понимать атаки на инструменты и протоколы, используемые нами для обеспечения безопасности наших данных, чтобы иметь возможность соответствующим образом корректировать наш подход. Если мы используем криптографический алгоритм, такой как RSA, для защиты данных, имплементация квантовых атак на этот алгоритм помогает нам понимать величину квантового устройства, которое потребуется нашим злоумышленникам, чтобы поставить под угрозу наши данные. В конце концов, существуют претензиозные заявления от «с RSA вообще нет проблем» до «мы должны быть в ужасе»; разведение этих крайностей по разные стороны требует понимания ресурсов, которые понадобятся злоумышленнику.

Другими словами, разведав сценарий, который на мгновение ставит нас в роль злоумышленника, мы можем понять, сколько квантовых вычисли-

тельных мощностей потребуется для успешной атаки на RSA. Мы вернемся к этому вопросу в конце главы, но пока это помогает нам думать как злоумышленник. Понимание того, как квантовые компьютеры могут атаковать классическую криптографию, служит отличным сценарием для практики применения наших навыков квантовых вычислений!

Однако в использовании этого примера стоит соблюдать осторожность. Влияние квантовых вычислений на информационную безопасность зависит от того, какие допущения мы принимаем о классических алгоритмах, улучшениях в квантовых алгоритмах, прогрессе в разработке квантового оборудования, сколько времени нам нужно для сохранения секретности и многих других подобных вопросах. Освещение всего этого на достаточно хорошем уровне, чтобы принять *ответственные решения* о том, как лучше всего развертывать криптографию, заняло бы, к сожалению, больше места, чем в этой книге, поэтому мы рекомендуем иметь в виду, что сценарий RSA в этой главе является примером, а не полным анализом предмета.

Оказывается, несмотря на то что Диффи–Хеллман и RSA выглядят очень по-разному, мы можем применить немного классической математики, чтобы превратить головоломку с факторизацией RSA в еще один пример того, как быстро мы перемещаемся по циферблату часов при выполнении модульной арифметики (головоломка Диффи–Хеллмана). Тогда эту задачу можно легко решить на квантовом компьютере, используя то, что мы узнали в главе 10. Давайте рассмотрим краткий пример использования алгоритма Шора для умножения небольшого целого числа, чтобы иметь возможность увидеть все эти части в работе.

12.2.1 Пример факторизации с использованием алгоритма Шора

Перечисленные для алгоритма Шора шаги могут показаться очень абстрактными, поэтому, прежде чем мы перейдем к принципу их работы, давайте попробуем пример, используя то, что мы узнали ранее о модульной арифметике. Допустим, число, которое мы хотим факторизовать, равно 21; реальные публичные ключи RSA будут намного больше, но давайте использовать 21, чтобы разобраться с математикой вручную. Поверьте, это поможет усвоить математику значительно *легче*.

Ниже приведены шаги алгоритма 12.1 для факторизации числа 21.

1. Выбрать случайное целое число в качестве генератора; допустим, мы используем 11.
2. Мы можем подтвердить, что, поскольку 11 не имеет общих факторов с 21, мы можем его использовать в качестве генератора для следующего шага.
3. К сожалению, мы не можем выполнить квантовый шаг в нашей голове, поэтому используем итеративную операцию Q# фазового оценивания, чтобы оценить фазу, генерируемую путем применения оракула, имплементирующего классическую функцию $f(x)$ =

$11^x \bmod 21$. Он возвращает фазу φ, которую мы можем преобразовать в частоту 427 с помощью уравнения $(\varphi * 2^9))/2\pi$.

4. Мы применяем алгоритм непрерывных дробей на числе 427, чтобы получить догадку о возможном периоде. Делая это вручную, мы получаем расчетную оценку периода, равную 6.
5. Найденный нами период является четным, поэтому мы можем перейти к следующему шагу.
6. Использование периода 6 дает нам, что либо $11^{6/2} - 1 \bmod 21 = 7$, либо $11^{6/2} + 1 \bmod 21 = 9$ имеет общий фактор с 21. Мы можем проверить и подтвердить каждый вариант, убеждаясь, что 7 действительно является фактором 21.

Упражнение 12.2: поиск общих факторов

Попробуйте выполнить шаг 6 из приведенного выше процесса, но используя 35 в качестве факторизуемого числа, 17 в качестве генератора и 12 в качестве периода. Убедитесь, что один или оба ответа, полученных на шаге 6, имеют общий фактор с 35.

Используя Python или Q#, попробуйте то же самое с $N = 143$, $g = 19$ и периодом $r = 60$.

Примечание: в следующем далее разделе мы увидим, как легко использовать классический компьютер для факторизации числа, когда дано еще одно число, которое имеет несколько общих факторов.

Хотя факторизовывать такие малые числа, как 21, 35 или 143, не такая уж большая работа, точно такой же процесс работает и для гораздо более крупных целых чисел, таких как те, с которыми мы можем столкнуться при попытке решить головоломку, которая в алгоритме RSA используется для защиты данных.

Остальная часть главы подробно описывает каждый из этих шагов и показывает принцип их совместной работы по факторизации целых чисел. Запуская этот процесс, давайте рассмотрим классическую математику, лежащую в основе того, как отыскание периода помогает нам факторизовывать целые числа и каким образом можно использовать Q# для имплементирования этой классической математики.

12.3 Классическая алгебра и факторизация

Принимая во внимание конкретный пример использования алгоритма Шора, мы видим, как классическая арифметика и алгебра помогают использовать преимущества квантовых вычислений. Прежде чем перейти к стержневой квантовой части алгоритма, полезно чуть-чуть подробнее разведать классическую часть, чтобы понять, почему отыскание периода генератора помогает факторизовывать целые числа.

Мы, возможно, помним из алгебры, что для любого числа x $x^2 - 1 = (x + 1)(x - 1)$. Как оказалось, это работает и в модульной (часовой) арифметике. Если мы обнаружим, что период r нашего генератора g является четным:

- тогда это означает, что существует целое число k ...
- такое, что $g^r = g^{2k} \bmod N = 1$.

Вычитая по единице с каждой стороны, мы получаем $(g^{2k} - 1) \bmod N = 0$,

- вследствие чего использование $x^2 - 1 = (x + 1)(x - 1)$ дает нам ...
- $(g^k + 1)(g^k - 1) \bmod N = 0$.

Почему это имеет значение? Если у нас есть $x \bmod N = 0$, то это говорит нам о том, что x кратно N. Возвращаясь к аналогии с часами, 0, 12, 24, 36 и т. д. равны нулю mod 12. Говоря по-другому, если $x \bmod N = 0$, то существует некоторое целое число y такое, что $x = yN$. Используя это с тем, что мы получили из периода, мы знаем, что существует некоторое целое число y такое, что $(g^k + 1)(g^k - 1) = yN$. Если либо $g^k - 1$, либо $g^k + 1$ кратно N, то мы мало что узнали; но в любом другом случае это говорит нам о том, что либо $g^k - 1$, либо $g^k + 1$ должны иметь общий фактор с N.

В целях выяснения того, имеет ли $g^k - 1$ либо $g^k + 1$ общий фактор с N, мы можем вычислить *наибольший общий делитель* (*greatest common divisor*, аббр. *GCD*) каждой догадки с N. Это легко делается классическим компьютером, используя технический прием, именуемый *алгоритмом Евклида*.

> **ПРИМЕЧАНИЕ** Поскольку GCD так легко вычисляется классически, зачем нам нужен квантовый компьютер для помощи в факторизации? К этому месту в алгоритме Шора мы уже сузили потенциальные факторы до двух очень хороших догадок и используем GCD только на этих самых догадках. Если бы мы не сузили круг вопросов настолько хорошо, то нам пришлось бы использовать GCD для многих и многих других догадок, чтобы получить хорошие шансы найти факторы N. Даже если GCD и определяется столь просто, нам все равно нужен хороший способ сначала сузить его до хорошего множества догадок.

На языке Q# мы можем вычислить GCD с помощью функции `GreatestCommonDivisorI`, как показано в листинге 12.2, где исходный код выполняется в среде блокнотов Q# Jupyter Notebook. Проверить правильность результата функции `GreatestCommonDivisorI` можно, начав с двух целых чисел, выраженных как произведение простых факторов: например, $a = 2 \times 3 \times 113$ и $b = 2 \times 3 \times 5 \times 13$. Поскольку эти два целых числа имеют в качестве общих факторов только 2 и 3, их GCD должен быть равен $2 \times 3 = 6$.

Листинг 12.2 Отыскание наибольшего общего делителя двух целых чисел

```
In [1]: open Microsoft.Quantum.Math;
        open Microsoft.Quantum.Diagnostics;

        function GcdExample() : Unit {                    ❶
```

```
            let a = 2 * 3 * 113;
            let b = 2 * 3 * 5 * 13;
            let gcd = GreatestCommonDivisorI(a, b);                    ❷
            Message($"GCD чисел {a} и {b} равен {gcd}.");

            EqualityFactI(gcd, 6, "Получен неверный GCD.");            ❸
        }
Out[1]: - GcdExample
In [2]: %simulate GcdExample                                           ❹
GCD чисел 678 и 390 равен 6.
Out[2]: ()
```

❶ Эта функция является простым тестовым случаем, чтобы увидеть принцип работы GCD.
❷ В целях вычисления GCD мы вызываем GreatestCommonDivisorI из пространства имен Microsoft.Quantum.Math, которое мы открыли ранее.
❸ Использует функцию EqualityFactI, чтобы подтвердить, что полученный нами ответ соответствует ожидаемому (2 × 3 = 6).
❹ Как обычно, мы можем применить команду `%simulate` для выполнения функции или операции на симуляторе. Здесь мы получаем назад (), так как GcdExample возвращает результат типа Unit.

Документация по стандартной библиотеке Q#

Как обычно, листинг 12.2 начинается с инструкций open, которые позволяют нам использовать функции и операции, предлагаемые стандартной библиотекой Q#. В данном случае функция Q#, которая вычисляет GCD из двух целых чисел, находится в пространстве имен `Microsoft.Quantum.Math`, поэтому мы начнем с открытия этого пространства имен, чтобы сделать эту функциональность доступной. Аналогичным образом факты и подверждения, необходимые для тестирования нашей новой функции `GcdExample`, можно задействовать, открыв пространство имен `Microsoft.Quantum.Diagnostics`.

Полный список того, что доступно в стандартной библиотеке Q#, можно получить, обратившись по адресу https://docs.microsoft.com/en-us/qsharp/api/, где расположен необходимый вам справочный материал.

Входные типы и соглашения об именах в Q#

Обратите внимание на букву `I` в конце имени `GreatestCommonDivisorI`. Она говорит нам о том, что `GreatestCommonDivisorI` работает на входных значениях типа `Int`. При использовании алгоритма Шора на практике *N* будет намного больше, чем мы можем вместить в обычное значение `Int`, поэтому Q# также предоставляет в помощь еще один тип под названием `BigInt`.

Для работы с входными значениями `BigInt` Q# также предоставляет `GreatestCommondivisorL`. Почему `L`, а не `B`? В этом случае `L` означает «long» (длинный), помогая избавиться от неоднозначности других типов, начинающихся с «B», таких как `Bool`.

Это соглашение также используется во всех остальных стандартных библиотеках Q#. Например, факт равенства, который мы использовали ранее, сравнивал два целых числа и поэтому называется `EqualityFactI`. Соответ-

ствующий факт для сравнения двух больших целых чисел называется `EqualityFactL`, в то время как факт для сравнения двух результирующих значений называется `EqualityFactR`.

Упражнение 12.3: наибольшие общие знаменатели

Каким будет GCD чисел 35 и 30? Помогает ли это вам найти факторы числа 35?

Подсказка: думайте об этом как о шаге 2 предыдущего упражнения.

Собрав все вместе, если у нас есть период нашего генератора, то следующий ниже листинг показывает, каким образом можно использовать его для написания функции `MaybeFactorsFromPeriod` на Q#. Имя этой функции начинается с «maybe» («возможный»), потому что есть вероятность, что найденный период не будет соответствовать условиям, необходимым для того, чтобы узнать что-то о факторах числа.

Листинг 12.3 operations.qs: вычисление возможных факторов из периода

```
function MaybeFactorsFromPeriod(
    generator : Int, period : Int, number : Int)                    (1)
: (Bool, (Int, Int)) {                                              (2)
    if period % 2 == 0 {                                            (3)

        let halfPower = ExpModI(generator,
            period / 2, number);                                    (4)

        if (halfPower != number - 1) {                              (5)
            let factor = MaxI(                                      (6)
                GreatestCommonDivisorI(halfPower - 1, number),
                GreatestCommonDivisorI(halfPower + 1, number)
            );
            return (true, (factor, number / factor));
        } else {
            return (false, (1, 1));
        }
    } else {
        return (false, (1, 1));
    }
}
```

(1) В целях вычисления возможных факторов из периода нам нужно взять входные значения для числа *N*, которое мы пытаемся разложить на факторы, периода *r* и генератора.

(2) Если либо $g^{r/2} + 1$, либо $g^{r/2} - 1$ кратно *N*, то мы не можем найти никаких факторов и должны повторить попытку. Результат типа Bool сообщает источнику вызова о необходимости повторить попытку.

(3) Если период нечетный, то мы не можем применить трюк $x^2 - 1 = (x + 1)(x - 1)$, поэтому мы начинаем с проверки четности периода.

(4) Проверяет, что $g^{r/2} + 1$ не кратно *N*, поэтому мы знаем, что можем безопасно продолжить.

❺ **Функция Q# Microsoft.Quantum.Math.ExpModI возвращает модульно-арифметические экспоненциалы вида g^x mod N. Это можно использовать для отыскания $g^{r/2}$ mod N с учетом *g*, *r* и *N*.**

❻ **GCD сообщает о том, имеет или нет одна из наших догадок общий фактор с *N*. Если наша догадка не имеет общих факторов, то GCD возвращает 1. Это проверяет обе догадки и принимает ту, которая дает что-то, отличное от 1.**

Теперь, когда мы знаем, как конвертировать период в потенциальные факторы, давайте рассмотрим суть алгоритма Шора: использование фазового оценивания для оценивания периода нашего генератора. Для этого мы будем использовать то, что узнали в остальной части книги, вместе с парой новых операций Q# для выполнения арифметики на квантовом компьютере. Давайте приступим!

Глубокое погружение: вот смотрю я на Евклида

Ранее мы использовали функцию `GreatestCommonDivisorI`, поставляемую со стандартными библиотеками Q#, для вычисления GCD двух целых чисел. Эта функция работает, используя алгоритм Евклида, который рекурсивно пытается делить целое число на еще одно целое число до тех пор, пока не останется никакого остатка.

Предположим, что мы хотим найти GCD двух целых чисел *a* и *b*. Мы начинаем алгоритм Евклида с отыскания двух дополнительных целых чисел *q* и *r* (сокращение от англ. *quotient* (частное) и *remainder* (остаток)), таких что $a = qb + r$. Найти *q* и *r*, используя инструкции целочисленного деления, несложно, поэтому данный шаг не слишком труден для классического компьютера. В этом месте если $r = 0$, то мы закончили: *b* является делителем как *a*, так и самого себя, поэтому не может быть большего общего делителя. Если нет, то мы знаем, что GCD чисел *a* и *b* также должен быть делителем *r*, и, значит, вместо этого мы можем рекурсивно найти GCD чисел *b* и *r*. В конечном итоге этот процесс должен закончиться, так как целые числа, чей GCD мы ищем, становятся все меньше и меньше, но никогда не становятся отрицательными.

Для того чтобы выразить все еще чуть-чуть конкретнее, мы можем пройтись по примеру из листинга 12.2 пошагово, как показано в следующей ниже таблице.

Использование алгоритма Евклида для отыскания GCD чисел 678 и 390

a	*b*	*q*	*r*
678	390	1	288
390	288	1	102
288	102	2	84
102	84	1	18
84	18	4	12
18	12	1	6
12	6 (ответ)	2	0 (готово)

12.4 Квантовая арифметика

К настоящему времени мы увидели довольно много разных частей стандартных библиотек Q#, и с учетом того, что эта глава посвящена арифметике, имеет смысл ввести несколько функций и операций из пространства имен `Microsoft.Quantum.Arithmetic`, предоставляемого численной библиотекой Numerics для Q#. Как вы можете догадаться, это пространство имен предоставляет массу полезных функций, операций и типов, которые упрощают выполнение арифметики в квантовых системах. В частности, мы можем использовать имплементации для таких вещей, как сложение и умножение чисел, представленных в кубитовых реестрах, с поддержкой многокубитовых реестровых кодировок, таких как `BigEndian` (где наименее значимый бит находится слева) и `LittleEndian` (где наименее значимый бит находится справа). Давайте рассмотрим пример исходного кода, в котором используется библиотека Q# Numerics для сложения двух целых чисел.

> **ПРИМЕЧАНИЕ** Блокнот Q# в репо образцов исходного кода (https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp) имеет все эти фрагменты кода, подготовленные специально для вас!

Прежде всего, поскольку численный пакет Numerics по умолчанию не загружен, нам нужно запросить ядро Q# загрузить его с помощью волшебной команды `%package`. Волшебная команда `%package` добавляет новый пакет в наш сеанс IQ#, делая функции, операции и пользовательские типы, имплементированные в этом пакете, доступными для нас в нашем сеансе.

Для упрощения задачи мы также можем отключить вывод на экран небольших амплитуд из диагностических функций, таких как `DumpMachine`, как показано в следующем ниже листинге.

Листинг 12.4 Загрузка пакетов и настройка параметров в IQ#

```
In [1]: %package Microsoft.Quantum.Numerics                          ❶
Adding package Microsoft.Quantum.Numerics: done!
Out[1]: - Microsoft.Quantum.Standard::0.15.2101125897                ❷
        - Microsoft.Quantum.Standard.Visualization::0.15.2101125897
        - Microsoft.Quantum.Numerics::0.15.2101125897
In [2]: %config dump.truncateSmallAmplitudes = "true"                ❸
Out[2]: "true"
```

❶ Использует команду %package для загрузки пакета Microsoft.Quantum.Numerics, который предоставляет дополнительные операции и функции для работы с числами, представленными реестрами кубитов.

❷ После выполнения команды %package IQ# сообщает о том, какие пакеты в настоящее время доступны в нашем сеансе IQ#. Ваши номера версий, скорее всего, будут отличаться.

❸ Волшебная команда %config выполняет различные настройки для текущего сеанса IQ#. Здесь, например, мы можем использовать %config, дабы сообщать вызываемым диагностическим функциям DumpRegisterи DumpMachine, чтобы они пропускали части каждого вектора состояния с очень малыми амплитудами. Это значительно облегчает визуализацию состояний на многочисленных кубитах, так как распечатка каждого вычислительно-базисного состояния быстро станет громоздкой.

ДЛЯ СПРАВКИ Во время работы с Q# из среды блокнотов Jupyter Norebook ядро IQ# предоставляет несколько других волшебных команд, помогающих писать квантовые программы, в дополнение к таким командам, как `%simulate`, `%package` и `%config`, которые мы встречали до этого. С полным их списком можно ознакомиться, обратившись к документации по адресу https://docs.microsoft.com/qsharp/api/iqsharp-magic.

12.4.1 Сложение с помощью кубитов

Теперь давайте перейдем к кодированию примера, который складывает два целых числа, когда они закодированы в реестрах кубитов. В листинге 12.5 используется операция `AddI` для сложения содержимого двух квантовых реестров друг с другом. В этом листинге используется пользовательский тип `LittleEndian`, предоставляемый стандартной библиотекой Q#, чтобы отмечать, что каждый реестр кубитов должен интерпретироваться как целое число с использованием кодировки от младшего к старшему (также известной как *кодировка справа налево от наименее значимого бита*). То есть, интерпретируя реестр `LittleEndian` как целое число, трактовать наименьший индекс кубита как наименее значимый бит. Например, чтобы представить целое число 6 в виде трехкубитового квантового состояния в представлении от младшего к старшему, мы пишем $|011\rangle$, так как $6 = 0 \times 20 + 1 \times 21 + 1 \times 22 = 2 + 4$.

Листинг 12.5 Использование численной библиотеки Numerics для сложения целых чисел, кодированных в кубитах

```
In [3]: open Microsoft.Quantum.Arithmetic;
        open Microsoft.Quantum.Diagnostics;
        open Microsoft.Quantum.Math;

        operation AddCustom(num1 : Int, num2 : Int) : Int {
            let bitSize = BitSizeI(MaxI((num1, num2))) + 1;          ❶
            use reg1 = Qubit[bitSize];
            use reg2 = Qubit[bitSize];
            let qubits1 = LittleEndian(reg1);                        ❷
            let qubits2 = LittleEndian(reg2);                        ❷

            ApplyXorInPlace(num1, qubits1);                          ❸
            ApplyXorInPlace(num2, qubits2);

            Message("Перед сложением:");
            DumpRegister((), reg2);

            AddI(qubits1, qubits2);                                  ❹

            Message("После сложения:");
            DumpRegister((), reg2);

            ResetAll(reg1);                                          ❺
            return MeasureInteger(qubits2);
```

```
        }
Out[3]: - AddCustom
```

1. Реестры должны быть достаточно большими, чтобы вместить максимально возможную сумму двух целых чисел. Самое большее, нам нужно на один бит больше, чем требуется для представления наибольшего входного значения.
2. Указывает на то, что мы хотим интерпретировать reg1 и reg2 как целые числа, представленные с использованием кодировки от младшего к старшему.
3. Подготавливает представление целого числа в реестре кубитов, поскольку $x \oplus 0 = x$ при x, равном 0 либо 1.
4. Используя операцию AddI, загруженную из численного пакета Numerics, мы можем сложить целые числа, представленные двумя входными реестрами qubits1 и qubits2.
5. Сбрасывает первый реестр, чтобы иметь возможность его высвободить, а затем измеряет реестр с результатами.

Результат выполнения этого фрагмента кода мы видим на рис. 12.4.

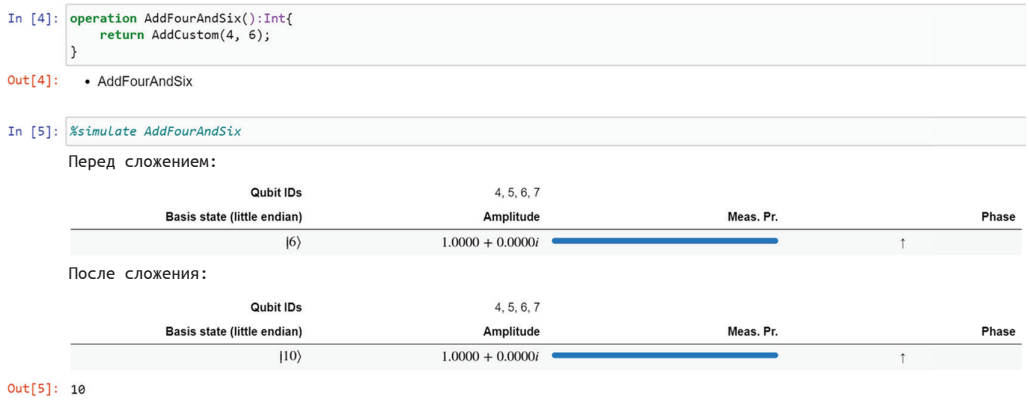


Рис. 12.4 Результат использования численной библиотеки Numerics для сложения целых чисел, кодированных в кубитовых реестрах

12.4.2 Умножение с кубитами в суперпозиции

Мы познакомились с принципом выполнения нескольких базовых модульно-арифметических вычислений на Q#, но, как и большинство наших квантовых алгоритмов, если мы не используем уникальные квантовые свойства/операции, то у нас получатся просто очень и очень дорогие вычисления. В этом разделе мы будем использовать тот факт, что у нас могут быть кубиты в *суперпозициях* чисел, что поможет нам получить преимущество, необходимое для работы алгоритма Шора. К счастью, `AddI` и многие другие подобные арифметические операции работают в суперпозиции: свойство, которое нам необходимо использовать для этих арифметических операций с фазовым оцениванием в следующей части главы. Однако, прежде чем перейти к этой теме, полезно немного поиграть со сложением и умножением целых чисел в суперпозиции, чтобы получить представление о том, что это значит.

Здесь мы увидим, как применять то, что мы узнали о суперпозиции, к арифметике, используя `MultiplyByModularInteger` в качестве примера. Вскоре мы будем использовать ту же операцию для строительства оракула, который нам потребуется для алгоритма факторизации Шора позже, так что это довольно практическое применение.

Сначала давайте рассмотрим операцию, которую можно использовать для подготовки реестра в суперпозиции двух целых чисел. В листинге 12.6 показано, как это сделать, используя то, что мы узнали об операции `ApplyXorInPlace` в предыдущем разделе и функторе `Controlled` в главе 9.

Как мы убедились в других случаях использования Controlled, контролируемые операции что-то делают, когда их контрольные реестры находятся в состоянии «все единицы» (|11⋯1⟩). Листинг 12.6 вместо этого контролирует нулевое состояние, используя операцию *X* для соотнесения состояния |0⟩ с состоянием |1⟩. Помещая вызов `X` в блок `within/apply`, мы обеспечиваем, чтобы Q# откатывал наш вызов `X` после применения контролируемой операции.

> **ДЛЯ СПРАВКИ** При использовании `within/apply` в таком ключе достигается нечто очень похожее на функцию `ControlledOnInt`, которую мы использовали в главе 11, и представляет собой подход, принятый при имплементировании этой функции в стандартных библиотеках Q#.

Листинг 12.6 Подготовка реестра в суперпозиции целых чисел

```
open Microsoft.Quantum.Arithmetic;
open Microsoft.Quantum.Diagnostics;
open Microsoft.Quantum.Math;

operation PrepareSuperpositionOfTwoInts(                          ❶
    intPair : (Int, Int),
    register : LittleEndian,
) : Unit is Adj + Ctl {
    use ctrl = Qubit();
    H(ctrl);                                                      ❷

    within {
        X(ctrl);                                                  ❸
    } apply {
        Controlled ApplyXorInPlace(                               ❹
            [ctrl],
            (Fst(intPair), register)
        );
    }
    Controlled ApplyXorInPlace(                                   ❺
        [ctrl],
        (Snd(intPair), register)
    );
    (ControlledOnInt(Snd(intPair), Y))(register!, ctrl);          ❻
}
```

❶ Берет реестр и пару целых чисел и подготавливает указанный реестр в суперпозиции этих целых чисел в кодировке LittleEndian.
❷ Подготавливает контрольный кубит в состоянии $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$, вследствие чего при управлении последующими операциями на этом кубите они также находятся в суперпозиции.
❸ Как отмечалось ранее, использование X в блоке within/apply позволяет контролировать на состоянии $|0\rangle$ вместо состояния $|1\rangle$.
❹ Добавляет новое входное значение для контрольного реестра (см. главу 9). Другим входным значением является кортеж с изначальными аргументами: целое число, которое мы хотим подготовить как состояние, и реестр, на котором мы хотим подготовить это состояние.
❺ Делает то же самое со вторым целым числом в intPair и кодирует его в реестре как контролируемое на кубите ctrl.
❻ Добавляет некоторую фазу в одну из двух ветвей нашей суперпозиции, используя операцию Y для поворота, контролируемого на нашем контрольном кубите, находящемся в состоянии $|1\rangle$.

ДЛЯ СПРАВКИ В этом случае операция Y не нужна, но она помогает нам увидеть, как фаза, применяемая операцией контролируемого-Y, распространяется на последующие шаги.

Имея квантовый реестр, представляющий суперпозицию двух целых чисел, мы можем применять к этой суперпозиции другие арифметические операции. В следующем ниже листинге, например, мы используем диагностическую функцию `DumpMachine`, чтобы увидеть, как изменяется состояние нашего реестра при использовании операции `MultiplyByModularInteger`, предоставляемой стандартными библиотеками Q#.

Листинг 12.7 Использование численной библиотеки Numerics для умножения в суперпозиции

```
operation MultiplyInSuperpositionMod(
    superpositionInts : (Int, Int),
    multiplier : Int,
    modulus : Int
) : Int {
    use target = Qubit[BitSizeI(modulus - 1)];                    ❶
    let register = LittleEndian(target);
    PrepareSuperpositionOfTwoInts(superpositionInts, register);   ❷

    Message("Перед умножением:");
    DumpMachine();

    MultiplyByModularInteger(                                     ❸
        multiplier, modulus, register
    );

    Message("После умножения:");
    DumpMachine();

    return MeasureInteger(register);                              ❹
}
```

❶ Прежде всего нам нужно выделить достаточно большой реестр. Поскольку здесь мы выполняем модульное умножение, максимально возможное значение, которое должен иметь наш реестр, равно modulus – 1.

2. **Использует операцию, определенную в листинге 12.6, для подготовки целевого реестра в суперпозиции целых чисел.**
3. **Численный пакет Numerics предоставляет операцию MultiplyByModularInteger, которая берет реестр типа LittleEndian и умножает его на значение классического множителя по модулю заданного modulus.**
4. **Измеряет реестр и возвращает классическое значение типа Int, представленное этим реестром.**

Если выполнить исходный код из листингов 12.6 и 12.7 в образце блокнота, то мы увидим результат, показанный на рис. 12.5: реестры правильно показывают суперпозицию 2 и 3 перед умножением, а затем показывают суперпозицию 1 и 6 после него. Чего мы ожидаем от правильного результата? Если умножить, как обычно, то должно быть 6 и 9; но так как мы выполняем арифметику mod 8, 9 соответствует 1. Когда мы затем измеряем этот реестр, мы получаем 6 в первой половине случаев и 1 в другой половине, поскольку они находятся в равной суперпозиции, как мы видим из полос амплитуд состояний, изображенных на рис. 12.5.

Упражнение 12.4: модульное умножение

Предположим, что вы подготовили реестр в состоянии $1/\sqrt{2}(|2\rangle + |7\rangle)$, причем каждый кет представляет собой целое число в кодировке от младшего к старшему. В каком состоянии будет находиться ваш реестр после умножения на 5 mod 9? Напишите программу на Q#, которая использует функцию `DumpMachine` для подтверждения вашего ответа.

```
In [18]: operation MultiplyInSuperpostionTest() :  Int {
             // Here we are multiplying 3 by a superposition of 2 and
             // 3 all mod 8.
             return MultiplyInSuperpositionMod((2, 3), 3, 8);
         }
Out[18]:  • MultiplyInSuperpostionTest

In [19]: %simulate MultiplyInSuperpostionTest
         Перед умножением:
```

Qubit IDs	0, 1, 2, 3		
Basis state (little endian)	Amplitude	Meas. Pr.	Phase
|2⟩	$0.7071 + 0.0000i$		↑
|3⟩	$0.0000 - 0.7071i$		←

После умножения:

Qubit IDs	0, 1, 2, 3		
Basis state (little endian)	Amplitude	Meas. Pr.	Phase
|1⟩	$0.0000 - 0.7071i$		←
|6⟩	$0.7071 - 0.0000i$		↑

```
Out[19]: 6
```

Рис. 12.5 Результат умножения 3 на 2 и 3 в суперпозиции mod 8

Упражнение 12.5: бонус

Если вы выполните ту же программу, что и в предыдущем упражнении, но попытаетесь умножить на 3 mod 9, то получите ошибку. Почему?

Подсказка: Учтите, что то, о чем вы узнали в главе 8, должно быть справедливо для классической функции, чтобы ее можно было представлять квантовым оракулом. Если вы застряли, то ответ предоставлен ниже, в слегка скрытом виде с использованием https://rot13.com.

Ответ: Zhygvcylvat ol guerr zbq avar vf abg erirefvoyr. Sbe vafgnapr, obgu bar gvzrf guerr naq sbhe gvzrf guerr zbq avar tvir mreb, rira gubhtu bar naq sbhe nera'g rdhny zbq avar. Fvapr pynffvpny shapgvbaf unir gb or erirefvoyr va beqre gb or ercerfragrq ol dhnaghz bcrengvbaf, gur `ZhygvcylOlZbqhyneVagrtre` envfrf na reebe va guvf pnfr[1].

ПРИМЕЧАНИЕ Обратите внимание на то, что мы здесь сделали: мы использовали квантовую программу для умножения целого числа, представленного квантовым реестром, на классическое целое число. Вычисление происходит полностью на квантовом устройстве и не использует никаких измерений; это означает, что когда наш реестр начинается в суперпозиции, *умножение также происходит в суперпозиции*.

12.4.3 Модульное умножение в алгоритме Шора

Теперь, когда мы немного ознакомились с численной библиотекой Numerics, давайте вернемся к факторизации. Главные места алгоритма 12.1, где нам нужно выполнить некоторую модульную арифметику и применить численную библиотеку Numerics, находятся в шагах 3 и 4 (повторенных далее). Мы можем имплементировать три операции для быстрого выполнения этих шагов алгоритма Шора, чтобы иметь возможность перейти к факторизации целых чисел.

Первая операция, на которую мы можем взглянуть из примера исходного кода этой главы, имплементирует шаг 3 алгоритма 12.1, операцию оценивания частоты (листинг 12.8).

Алгоритм 12.1, шаги 3 и 4 (псевдокод факторизации целого числа с помощью алгоритма Шора)

3. Применить итеративное фазовое оценивание, чтобы найти *частоту* классической функции $f(x) = g^x \bmod N$. Частота говорит о том, как быстро f возвращается к тому же значению по мере увеличения x.
4. Применить классический алгоритм, именуемый *разложением непрерывных дробей*, для конвертирования частоты из предыдущего шага в период (r). Тогда период r должен обладать свойством, что $f(x) = f(x + r)$ для всех входных значений x.

[1] *Бонус от переводчика.* Вот перевод шифротекста: Умножение на три mod девять необратимо. Например, и единица умножить на три, и четыре умножить три, mod девять дают ноль, хотя единица и четыре не равны mod девять. Поскольку, для того чтобы быть представленными квантовыми операциями, классические функции должны быть обратимыми, в данном случае функция `MultiplyByModularInteger` вызывает ошибку.

ДЛЯ СПРАВКИ В этой операции используются операции фазового оценивания, предоставляемые стандартными библиотеками Q#, которые мы видели в главе 10. Если вам нужно освежить свои знания, то рекомендуем вернуться к главе 9, чтобы перечитать материал по фазовому оцениванию, либо главе 10 в отношении того, как выполнять фазовое оценивание с использованием стандартных библиотек.

Листинг 12.8 operations.qs: выяснение частоты генератора с помощью фазового оценивания

```
operation EstimateFrequency(
    inputOracle : ((Int, Qubit[]) => Unit is Adj+Ctl),
    nBitsPrecision : Int,
    bitSize : Int)
: Int {
    use register = Qubit[bitSize];                  ❶

    let registerLE = LittleEndian(register);        ❷
    ApplyXorInPlace(1, registerLE);                 ❸

    let phase = RobustPhaseEstimation(              ❹
        nBitsPrecision,
        DiscreteOracle(inputOracle),                ❺
        registerLE!
    );
    ResetAll(register);                             ❻

    return Round(                                   ❼
        (phase * IntAsDouble(2 ^ nBitsPrecision)) / (2.0 * PI())
    );
}
```

❶ Поскольку это основной шаг, на котором используются кубиты, нам нужно выделить реестр, достаточно большой для представления модуля (modulus).
❷ Только что выделенный реестр должен указывать на то, как кодировать целые числа, которые он будет представлять, чтобы иметь возможность использовать пользовательский тип LittleEndian для обертывания только что выделенного реестра.
❸ Берет значение типа Int и выполняет с ним и целым числом, хранящимся в реестре, указанном во втором аргументе, операцию XOR. Поскольку registerLE начинается как 0, это подготавливает реестр как 1.
❹ Использует RobustPhaseEstimation (см. главу 10) для выяснения фазы inputOracle и передает квантовый реестр и число битов точности, до которых мы хотим оценивать фазу.
❺ Обертывание inputOracle в пользовательский тип DiscreteOracle дает понять RobustPhaseEstimation, что мы хотим, чтобы inputOracle интерпретировался как оракул.
❻ После оценивания фазы сбрасывает все кубиты в реестре.
❼ Оцененная фаза – это всего лишь фаза. Вот уравнение, которое конвертирует ее в частоту: $(\text{phase} * 2^{\text{nBitsPrecision} - 1})/\pi$.

Теперь, когда у нас есть строительные леса для оценивания частоты, мы можем взглянуть на операцию, которой имплементируется оракул, необходимый нам для этого алгоритма. Операция `ApplyPeriodFindingOracle` – это всего лишь операция, которая структурирована как оракул для функции *f*(степень) = генератор$^{\text{степень}}$ mod modulus. В следующем ниже

листинге показана имплементация операции `ApplyPeriodFindingOracle`.

Листинг 12.9 operations.qs: имплементирование оракула для функции *f*

```
operation ApplyPeriodFindingOracle(
    generator : Int, modulus : Int, power : Int, target : Qubit[])
: Unit is Adj + Ctl {
    Fact(                                          ❶
        IsCoprimeI(generator, modulus),
        "generator и modulus должны быть взаимно простыми."
    );
    MultiplyByModularInteger(                      ❷
        ExpModI(generator, power, modulus),        ❸
        modulus,
        LittleEndian(target)                       ❹
    );
}
```

❶ Выполняет небольшую проверку входных данных в отношении того, что предоставленные аргументы generator и modulus являются взаимно простыми.
❷ То же самое, что и в листинге 12.7. Здесь это помогает данному оракулу умножить целое число, представленное в целевом реестре, на f(power) = generator^{power} mod modulus.
❸ В Microsoft.Quantum.Math также есть функция ExpModI, которая позволяет нам легко вычислять f(power) = generator^{power} mod modulus.
❹ LittleEndian говорит о том, что реестр кубитов, который берет операция ApplyPeriodFindingOracle, интерпретируется как целое число в кодировке от младшего к старшему.

Предыдущие две операции формируют основу для шага 3 алгоритма 12.1. Теперь нам нужна операция, которая позаботится о шаге 4, где мы конвертируем оцененную частоту генератора в период. Операция `EstimatePeriod` в следующем ниже листинге делает именно это: имея `generator` и `modulus`, она повторяет оценивание частоты с использованием операции `EstimateFrequency` и применяет алгоритм непрерывных дробей с целью обеспечить, чтобы оцененная частота давала валидный период.

Листинг 12.10 operations.qs: оценивание периодов из частот

```
operation EstimatePeriod(generator : Int, modulus : Int) : Int {
    Fact(                                                  ❶
        IsCoprimeI(generator, modulus),                    ❷
        "`generator` и `modulus` должны быть взаимно простыми"
    );

    let bitSize = BitSizeI(modulus);                       ❸
    let nBitsPrecision = 2 * bitSize + 1;                  ❹
    mutable result = 1;                                    ❺
    mutable frequencyEstimate = 0;

    repeat {                                               ❻
        set frequencyEstimate =
            EstimateFrequency(                             ❼
                ApplyPeriodFindingOracle(                  ❽
                    generator, modulus, _, _),
```

```
                nBitsPrecision, bitSize
            );

        if frequencyEstimate != 0 {                          ❾
            set result =
                PeriodFromFrequency(                         ❿
                    frequencyEstimate, nBitsPrecision,
                    modulus, result
                );
        } else {
            Message("Оцененная частота была равна 0, пробуем еще раз.");
        }
    }
    until ExpModI(generator, result, modulus) == 1           ⓫
    fixup {
        Message(                                             ⓬
            "Оцененный период из непрерывных дробей не сработал, " +
            "пробуем еще раз."
        );
    }
    return result;
}
```

❶ Выполняет небольшую проверку входных данных в отношении того, что предоставленный generator и modulus являются взаимно простыми.

❷ Функция IsCoprimeI из пространства имен Microsoft.Quantum.Math упрощает проверку, что generator и modulus являются взаимно простыми.

❸ Наибольшим целым числом, которое должен содержать реестр кубитов, является modulus, поэтому мы используем BitSizeI, чтобы рассчитать число битов такое, что modulus ⩽ $2^{\{\text{число битов}\}}$.

❹ В целях использования плавающей точки для представления k/r, где r – это период, а k – некое другое целое число, нам нужно достаточно битов точности для аппроксимирования k/r: т. е. на один бит больше, чем требуется для представления как k, так и r.

❺ Мутируемая переменная результата отслеживает текущую наилучшую догадку для периода, по мере того как мы повторяем блок reapeat.

❻ Повторяет шаги оценивания частоты столько раз, сколько необходимо в целях обеспечения того, чтобы у нас была приемлемая оценка периода, с которой можно двигаться дальше.

❼ Вызывает рассмотренную ранее операцию EstimateFrequency и передает ей соответствующие аргументы.

❽ Частично применяет ApplyPeriodFindingOracle, обеспечивая, чтобы EstimateFrequency могла применить ее к правильным значениям power и реестра.

❾ Если frequencyEstimate равна 0, то нам нужно повторить попытку, так как она не имеет смысла в качестве периода (1/0). Если мы получаем 0, то блок repeat выполняется снова, так как в этом случае условие until не будет удовлетворено.

❿ Захватывает шаг 4 алгоритма 12.1, который применяет алгоритм непрерывных дробей из стандартной библиотеки Q# для вычисления периода из частоты.

⓫ Если условие until не удовлетворяется, то выполняется блок fixup, который просто выдает сообщение о том, что он собирается повторить попытку.

⓬ Повторяет оценивание частоты и вычисление периода до тех пор, пока у нас не будет периода такого, что $\text{generator}^{\text{результат}} \bmod \text{modulus} = 1$.

Имея в своем распоряжении эту последнюю операцию, у нас есть весь исходный код, необходимый для полного имплементирования алгоритма Шора! В следующем разделе мы окончательно соберем все это вместе

и рассмотрим последствия указанного алгоритма целочисленной факторизации.

12.5 Окончательная сборка

Теперь мы изучили и отработали все навыки, необходимые для программирования и выполнения алгоритма Шора. Квантовая часть алгоритма Шора была довольно хорошо знакома благодаря тому, что мы узнали в главах 9 и 10 о фазовом оценивании; и мы прошлись по классической алгебре, которая связывает задачи факторизации чисел и отыскания периода генератора. Это немалый подвиг – и вы должны гордиться собой за то, что зашли так далеко в своем квантовом путешествии!

Конвергенты непрерывных дробей

Возможно, вы заметили, что в алгоритме Шора происходит еще одна часть классической математики, которую мы еще не затронули. В частности, перед тем как продолжить, функция `PeriodFromFrequency` вызывается на выходном значении, которое мы получаем из оценивания фазы:

```
function PeriodFromFrequency(
    frequencyEstimate : Int, nBitsPrecision : Int,
    modulus : Int, result : Int)
: Int {
    let continuedFraction = ContinuedFractionConvergentI(
        Fraction(frequencyEstimate, 2^nBitsPrecision), modulus
    );
    let denominator = AbsI(Snd(continuedFraction!));
    return (denominator * result) / GreatestCommonDivisorI(
        result, denominator
    );
}
```

Это необходимо, потому что оценивание фазы не говорит нам точно, что нам нужно. Вместо того чтобы сообщать нам о количестве времени, которое требуется нашей функции, чтобы вращаться по кругу по часовой стрелке (периоду нашей функции), оно сообщает нам о скорости, с которой наша функция вращается по указанному кругу: нечто, более похожее на частоту. К сожалению, мы не можем сделать что-то вроде взятия обратной частоты, чтобы вернуться к нашему периоду, поскольку мы ищем период как целое число. Следовательно, если мы получим оценку частоты f, то нам нужно искать вблизи $f/2n$ ближайшую дробь формы N/r, чтобы найти наш период r.

Это полностью классическая арифметическая задача, и, к счастью, она была хорошо решена с использованием технического приема, именуемого *конвергенты непрерывных дробей*. Это решение доступно в стандартных библиотеках Q# с помощью функции `ContinuedFractionConvergentI`, что позволяет легко переходить от оценки, которую мы получаем из оценивания фазы, к некоторой информации о периоде нашей функции.

Давайте воспользуемся моментом, чтобы рассмотреть операцию `FactorSemiprimeInteger`, которую мы увидели в начале главы, в свете того, что мы теперь узнали. Если вам нужно освежить свои знания, то рекомендуем ознакомиться с рис. 12.6, приводившимся ранее.

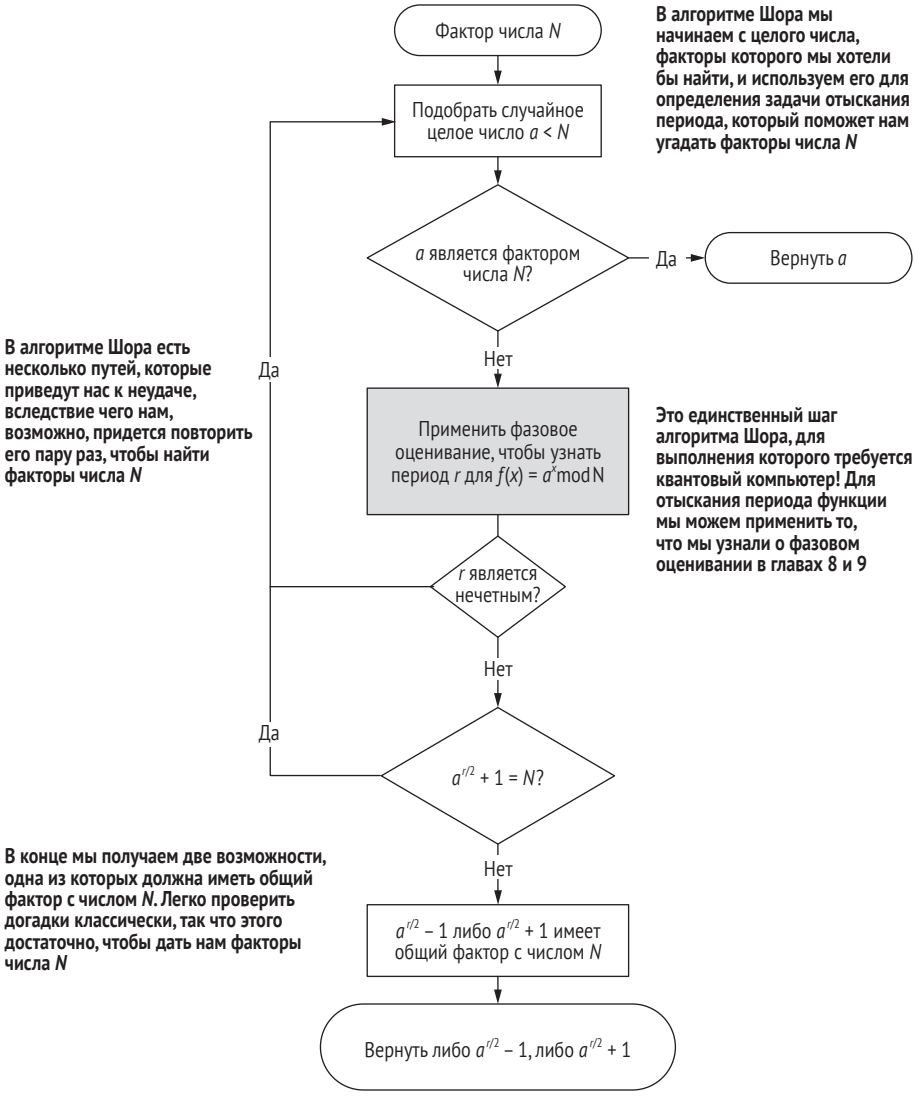


Рис. 12.6 Алгоритм Шора, представленный в виде блок-схемы. В целях разложения целого числа *N* на факторы в алгоритме Шора используется фазовое оценивание и квантовый компьютер, чтобы отыскать период функции, которая берет степени другого целого числа *a*, используя модульную арифметику mod *N*. После небольшой классической постобработки этот период можно использовать для отыскания факторов числа *N*

Листинг 12.11 operations.qs: факторизация полупростых целых чисел с помощью алгоритма Шора

```
operation FactorSemiprimeInteger(number : Int) : (Int, Int) {
    if number % 2 == 0 {                                          ❶
        Message("Дано четное число; 2 является фактором.");
        return (number / 2, 2);
    }
    mutable factors = (1, 1);                                     ❷
    mutable foundFactors = false;                                 ❸

    repeat {
        let generator = DrawRandomInt(3, number - 2);             ❹

        if IsCoprimeI(generator, number) {                        ❺
            Message($"Оцениваем период числа {generator}...");
            let period = EstimatePeriod(generator, number);       ❻
            set (foundFactors, factors) =
                MaybeFactorsFromPeriod(
                    generator, period, number
                );
        } else {
            let gcd = GreatestCommonDivisorI(number, generator);  ❼
            Message(
                $"Мы угадали, что делитель числа {number} равен " +
                $"{gcd}, нечаянно. Больше нечего делать."
            );
            set foundFactors = true;
            set factors = (gcd, number / gcd);
        }
    }
    until foundFactors                                            ❽
    fixup {                                                       ❾
        Message(
            "Оцененный период не дал валидного фактора, " +
            "пробуем еще раз."
        );
    }
    return factors;                                               ❿
}
```

❶ Проверяет, что факторизуемое целое число является четным. Если это так, то 2 является фактором, и мы можем остановиться досрочно.

❷ Использует мутируемую переменную factors для отслеживания факторов, которые мы находим для number по мере прохождения по алгоритму.

❸ Использует мутируемую флаговую переменную foundFactors, чтобы отслеживать отыскание каких-либо факторов для number при прохождении по алгоритму.

❹ Имплементирует шаг 1 алгоритма 12.1, используя DrawRandomInt для отбора случайного целого числа в диапазоне от 1 до number – 1 в качестве нашего генератора.

❺ Шаг 2 алгоритма 12.1, где мы проверяем, что generator является взаимно простым для числа, которое мы хотим факторизовать; если это не так, то блок else занимается возвращением общего фактора между ними.

❻ Охватывает шаги 3 и 4 алгоритма 12.1. Возвращает период, который вычисляется с помощью непрерывных дробей из оценивания частоты внутри EstimatePeriod.
❼ Использует алгебру для превращения оцененного периода в целые числа, которые могут быть факторами. Иногда это не удается, поэтому он также возвращает значение типа Bool, указывающее на то, что это удалось либо не удалось.
❽ Обрабатывает случай, когда генератор, о котором мы выдвинули догадку вначале, имеет общий фактор с числом, которое мы пытаемся факторизовать.
❾ Добавляет условие о том, как долго мы должны повторять предыдущий блок. Здесь мы хотим продолжать поиск до тех пор, пока не найдем факторы, которые ищем.
❿ Сообщает программе о том, что делать, прежде чем повторять главный цикл. Здесь мы это как раз и видим – дает нам знать, что наша программа Q# повторит попытку.
⓫ Возвращает кортеж факторов для входного целого числа.

Вот как это будет выглядеть, если выполнить эту операцию в блокноте IQ#, чтобы факторизовать 21.

Листинг 12.12 Результат выполнения FactorSemiprimeInteger

```
In [1]: open IntegerFactorization;                          ❶
        operation Factor21() : (Int, Int) {
            return FactorSemiprimeInteger(21);
        }

Out[1]: - Factor21

In [2]: %simulate Factor21
        Мы угадали, что делитель числа 21 равен 3, нечаянно. Больше нечего делать.
Out[2]: (3, 7)

In [3]: %simulate Factor21
        Оцениваем период числа 19...
        Оцененный период из непрерывных дробей не сработал, пробуем еще раз.
Out[3]: (7, 3)

In [4]: %simulate Factor21
        Оцениваем период числа 17...
        Оцененный период не дал валидного фактора, пробуем еще раз.
        Мы угадали, что делитель числа 21 равен 3, нечаянно. Больше нечего делать.
Out[4]: (3, 7)
```

❶ Для того чтобы вызвать операцию FactorSemiprimeInteger из блокнота, полезно написать новую операцию, которая предоставляет входное значение 21.

С каждым разом наш код правильно возвращал простые факторы числа 21: 3 и 7. Операция была выполнена три раза, показывая возможные разные результаты, которые мы могли бы получить. В `In [2]`, когда мы попытались угадать генератор, наш вызов `DrawRandomInt` в конечном итоге вернул целое число, которое не было равно 21. Следовательно, мы получили возможность использовать `GreatestCommonDivisorI` для отыскания факторизации. В `In [3]` наш вызов `DrawRandomInt` выбрал генератор, равный 19, а затем должен был дважды выполнить оценивание частоты в целях обеспечения того, чтобы алгоритм непрерывных дробей

выполнился успешно. В завершающем прогоне, в `In [4]`, был выполнен один полный раунд задачи отыскания периода, но он не дал правильного фактора; и когда он попытался выбрать новый генератор, он угадал фактор нечаянно.

> **ПРИМЕЧАНИЕ** Учитывая пределы, связанные с выполнением этого исходного кода на симуляторах или небольших аппаратных устройствах, при выборе генератора мы нередко будем угадывать правильные факторы. Кроме того, нам чаще не везет с малыми целыми числами, угадывая тривиальные факторы, такие как 1. Эти крайние случаи происходят реже, по мере того как число, которое мы пытаемся факторизовать, увеличивается.

Используя симулятор на ноутбуке, настольном компьютере или в облаке, мы, вероятно, не сможем факторизовывать с помощью алгоритма Шора нечто очень большое. Например, было бы довольно сложно факторизовать 30-битовое число, симулируя алгоритм Шора на классическом компьютере, при этом 40-битовые числа уже в 1992 году считались прискорбно недостаточными для противостояния классическим алгоритмам факторизации. Может показаться, что это делает алгоритм Шора бесполезным, но на самом деле это говорит нам только о том, что использовать классический компьютер для симулирования больших квантовых программ весьма трудно; мы увидели причину, почему это так, в главах 4 и 5.

И действительно, поскольку один и тот же алгоритм работает для гораздо более крупных чисел (например, использование 4096-битовых ключей не является перебором для защиты персональных данных), таких как те, которые обычно используются для защиты данных в интернете, понимание принципа работы алгоритма Шора и других подобных квантовых алгоритмов поможет нам оценивать допущения, которые входят в современное использование криптографии, и то, что еще нам потребуется, чтобы продвинуться вперед.

Что дальше для уединения?

Учитывая то, что мы узнали об алгоритме Шора, может показаться, что криптография, которая защищает все, от наших медицинских записей до истории нашего чата, обречена. К счастью, существуют как квантовые технологии (такие как квантовое распределение ключей, о котором мы узнали в главе 3), так и новые классические технологии, предназначенные для противодействия алгоритмам, таким как алгоритм Шора. Последний класс технологий, именуемый *постквантовой криптографией*, является предметом многочисленных научных исследований и разведывательных изучений.

Как оказалось, Q# может сыграть важную роль в исследованиях криптографии, облегчая понимание величины квантового компьютера, который потребуется для атаки на данную криптосистему. Например, исследователи из

Google недавно использовали Q# и Python для повышения стоимости, необходимой для имплементирования шага модульного умножения алгоритма Шора, что помогло им оценить, что для атаки разумных экземпляров RSA с использованием современных квантовых алгоритмов потребуется 20 миллионов кубитов[1]. Схожим образом http://github.com/Microsoft/grover-blocks является отличным примером использования языка Q#, чтобы понять, как алгоритм Гровера (из главы 11) влияет на алгоритмы с симметричным ключом, такие как AES.

В обоих случаях язык Q# является ценным инструментом для понимания количества квантово-вычислительных мощностей, которые потребуются злоумышленнику для компрометирования существующих криптосистем. Вместе с допущениями о скорости, с которой квантовые алгоритмы и аппаратура будут продолжать совершенствоваться, допущениями о количестве квантово-вычислительных мощностей, которые будет возможно приобрести злоумышленникам, и требованиями к тому, как долго алгоритмы, такие как RSA, должны быть безопасными, чтобы гарантировать нашу приватность, понимание, разработанное с помощью Q#, помогает нам осознать, как быстро необходимо менять существующие криптосистемы. Как и все в области информационной безопасности, обеспечение приватности от квантовых злоумышленников – очень сложная тема, не говоря уже о том, что на нее нет простых ответов. К счастью, такие инструменты, как Q# и Комплект инструментов для квантовой разработки, помогают сделать эту задачу чуть-чуть более сговорчивой.

Резюме

- Современная криптография работает, скрывая секреты с помощью математических головоломок, которые трудно решать классическими компьютерами, например факторизации чисел. Большие квантовые компьютеры можно использовать для факторизации чисел, меняя наше представление о криптографии.
- Модульная арифметика обобщает движение стрелок часов: например, 25 + 5 равно 3 на циферблате с 27 часами.
- Целые числа ровно с двумя простыми факторами называются *полупростыми* и могут факторизовываться квантовыми компьютерами для решения модульно-арифметических задач вместе с фазовым оцениванием.
- Численная библиотека Q# Numerics предоставляет полезные функции и операции для работы с модульными целыми числами на квантовых компьютерах.

---

[1] *Крейг Гидни.* Асимптотически эффективное квантовое умножение Карацуба (2018) // https://arxiv.org/abs/1904.07356; *Крейг Гидни и Мартин Экеро.* Как разложить 2048-битовые целые числа RSA за 8 часов, используя 20 миллионов шумных кубитов (2019) // https://arxiv.org/abs/1905.09749.

- Алгоритм Шора сочетает в себе классическую пред- и постобработку, необходимую для фазового оценивания на квантовом компьютере, чтобы быстро раскладывать целые числа на факторы с использованием модульной арифметики.

Заключение

Прежде чем мы попрощаемся, полезно сделать шаг назад и оценить, как все разнообразие навыков, которым мы учились на протяжении всей книги, сошлось в этой главе, чтобы помочь нам понять реальное применение квантовых компьютеров. В части I мы узнали основные принципы того, как можно описывать и симулировать квантовые компьютеры, а также узнали об основных квантовых эффектах, которые делают квантовые вычисления уникальными. В главе 3 мы узнали, как использовать одиночные кубиты и суперпозицию для безопасного обмена криптографическими ключами с квантовым распределением ключей. В главах 4–6 мы связали несколько кубитов, чтобы поиграть в игры и перемещать данные по квантовому устройству. Мы даже создали свой собственный квантовый симулятор на Python, чтобы имплементировать эти игры и узнать о математике, которая помогает нам описывать квантовые эффекты.

Имея все эти основы за плечами, в части II мы начали писать квантовые алгоритмы, чтобы помочь команде в Камелоте играть в некоторые игры. В главе 7 мы узнали о Q#, новом языке программирования, специально разработанном в целях простого написания программ для квантового компьютера. В главе 8 мы имплементировали алгоритм Дойча–Йожи для подбора нового короля, но по пути мы также узнали об оракулах и о том, как они помогают нам оценивать классические функции в квантовой программе. Мы также разработали нашу собственную программу фазового оценивания в главе 9, где узнали, как манипулировать фазой и использовать ее с фазовой отдачей для изучения операций в наших квантовых программах.

Имея в своем распоряжении новый набор технических приемов квантовой разработки, мы взялись за несколько самых захватывающих приложений квантовых вычислений. В главе 10 мы узнали о гамильтониановой симуляции и о том, каким образом можно использовать квантовые системы в наших квантовых компьютерах для симулирования уровней энергии в различных химических веществах. В главе 11 мы имплементировали алгоритм Гровера для поиска информации в неструктурированных данных с амплитудным усилением. В этой главе мы использовали все – от диагностических функций и операций языка Q# до фазового оценивания и от блоков `within/apply`, оракульного представления классических функций до факторизации чисел на квантовом компьютере. Используя то, что мы узнали в остальной части книги, большинство сложных частей написания алгоритма Шора были *классическими* частя-

ми, необходимыми для соединения задачи факторизации чисел с задачей отыскания периода.

Хотя эта книга не исчерпала всего, что можно узнать о квантовых вычислениях – в конце концов, с 1985 года многое произошло! – то, что вы узнали, дает вам все необходимое для продолжения своих занятий, разведки и продвижения вперед квантовых вычислений. Используя Python и Q# совместно, у вас есть инструменты, чтобы принять участие в одном из самых захватывающих достижений в области вычислительной техники, помочь своим напарникам и коллегам учиться вместе с вами и создавать сообщество, способное эффективно использовать квантовые вычисления. Идите и получайте удовольствие!

Что дальше?

Хотя всегда есть что узнать о квантовых вычислениях, теперь вы имеете все необходимое, чтобы начать разрабатывать квантовые приложения с использованием Python и Q# совместно. Если вы заинтересованы в том, чтобы учиться и добиться большего с помощью квантовых вычислений, то ниже приведено несколько ресурсов, которые помогут вам сделать следующий шаг:

- *Сообщество Q#* (qsharp.community) – сообщество с открытым исходным кодом, посвященное квантовому программированию на языке Q#, включая блоги, репозитории исходного кода и онлайновые встречи;
- *Microsoft Quantum Docs* (https://docs.microsoft.com/azure/quantum/) – полная справочная документация по всем вопросам, связанным с Комплектом инструментов для квантовой разработки (Quantum Development Kit);
- *arXiv* (arxiv.org) – онлайновое хранилище научных статей и рукописей, включая огромное количество исследований о квантовых вычислениях;
- *Унитарный фонд* (unitary.fund) – некоммерческая организация, предоставляющая гранты и финансовую поддержку для квантового программного обеспечения с открытым исходным кодом, а также заманчивые предложения по проектам с открытым исходным кодом, которые мы можем взять на себя;
- *Квантовый фонд с открытым исходным кодом* (www.qosf.org) – фонд для разработки квантового программного обеспечения с открытым исходным кодом, включая список текущих проектов и ресурсов для дальнейшего самостоятельного обучения;
- *Этика QC* (qcethics.org) – ресурсы по этике в квантовых вычислениях;
- *Q-поворот* (q-turn.org) – инклюзивная серия конференций по квантовым вычислениям;
- *Зоопарк квантовых алгоритмов* (quantumalgorithmzoo.org) – список известных квантовых алгоритмов со ссылками на статьи о каждом из них;
- книга «Квантовые вычисления: щадящее введение» Джека Д. Хайдари (*Quantum Computing: A Gentle Introduction*, Jack D. Hidary, Springer, 2019) – еще больше подробностей о математике, лежащей в основе квантовых алгоритмов, которые мы узнали в этой книге.

Во многих университетах и колледжах также есть курсы или исследовательские программы, которые, возможно, вызовут интерес по мере того, как вы продолжите изучать квантовые вычисления. Как бы вы ни решили продолжать, мы надеемся, что вы получите удовольствие и будете работать над тем, чтобы сделать сообщество квантовых вычислений еще более замечательным!

Приложение А

Инсталлирование требуемого программно-информационного обеспечения

Начало почти любого проекта предусматривает отыскание или настройку среды разработки на вашем компьютере. Это приложение к книге поможет вам подготовиться к использованию образцов, прилагаемых к этой книге в интернете (с помощью службы Binder или кодовых пространств GitHub) либо инсталлировать среду Python и Комплект инструментов для квантовой разработки от Microsoft, которые вы сможете использовать локально. Если у вас возникнут проблемы, то, пожалуйста, ознакомьтесь с самой последней документацией по использованию Комплекта инструментов для квантовой разработки (https://docs.microsoft.com/azure/quantum/install-overview-qdk) и/или разместите вопрос на репо этой книги на Github (https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp).

A.1 Выполнение образцов исходного кода в онлайновом режиме

Если вы хотите попробовать образцы исходного кода из этой книги, ничего не устанавливая, то существует два отличных варианта это сделать:

- Binder (mybinder.org), бесплатная служба для проведения разведывательного анализа размещенных на хосте блокнотов Jupyter;
- кодовые пространства GitHub, облачная среда разработки GitHub Codespaces.

A.1.1 Использование службы Binder

В целях использования Binder просто перейдите на https://mybinder.org/v2/gh/crazy4pi314/learn-qc-with-python-and-qsharp/master. Организация работы может занять несколько минут, но Binder выполнит новую инсталляцию среды блокнотов Jupyter Notebook, в комплекте с необходимыми пакетами языка Python и поддержкой языка Q#.

> **ПРЕДУПРЕЖДЕНИЕ** Служба Binder предназначена только для разведывательного анализа и будет удалять ваши изменения примерно через 20 минут бездействия. Хотя Binder и является отличным способом начать работу, если вы хотите продолжить разработку квантовых программ, то целесообразно либо использовать кодовые пространства GitHub, либо инсталлировать Python и Комплект инструментов для квантовой разработки локально на своем компьютере.

A.1.2 Использование облачной среды GitHub CodeSpaces

На момент написания этой книги Кодовые пространства GitHub находятся на этапе предварительного просмотра по адресу: https://github.com/features/codespaces. Инструкции по использованию примеров исходного кода из этой книги вместе с Кодовыми пространствами см. в репозитории примеров по адресу: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

A.2 Локальное инсталлирование с использованием Anaconda

В части I книги мы активно используем Python в качестве инструмента для разведывательного анализа квантового программирования, тогда как в частях II и III мы используем Python и Q# совместно. При этом мы опираемся на Комплект инструментов для квантовой разработки и несколько библиотек Python, которые облегчают написание научных программ. Как следствие при локальном инсталлировании бывает намного проще использовать дистрибутив научного программно-информационного обеспечения, такой как дистрибутив Anaconda (anaconda.com), который помогает управлять инструментами языка Python и другими инструментами научного программирования.

A.2.1 Инсталлирование Anaconda

В целях инсталлирования дистрибутива Anaconda следуйте инструкциям по адресу: https://docs.anaconda.com/anaconda/install.

ПРЕДУПРЕЖДЕНИЕ На момент написания этой книги дистрибутив Anaconda поставляется с Python 2.7 или 3.8. Python 2.7 официально выведен из эксплуатации в январе 2020 года и предоставляется только по соображениям совместимости. В этой книге мы исходим из наличия Python 3.7 или более поздней версии, поэтому убедитесь, что вы установили версию дистрибутива Anaconda, которая поддерживает Python 3.

A.2.2 Инсталлирование пакетов с помощью Anaconda

Пакеты – это отличный способ совместной работы и экономии времени, когда мы пытаемся изучить или разработать новый код. Они представляют собой механизм сбора родственного кода и его обертывания, чтобы иметь возможность легко им делиться с другими машинами. Пакеты могут быть инсталлированы на нашей машине с помощью того, что разумно называется *менеджером пакетов*, из которых есть несколько широко используемых вариантов для Python. Мы можем предпочесть один менеджер другому, потому что у каждого есть свой список пакетов, и пакет, который мы хотим установить, может быть известен только конкретному человеку (в зависимости от того, как его развернул автор). Давайте начнем с рассмотрения менеджеров пакетов, которые мы уже инсталлировали в составе дистрибутива Anaconda.

По умолчанию дистрибутив Anaconda поставляется с двумя менеджерами пакетов: pip и conda. Учитывая, что мы инсталлировали дистрибутив Anaconda, менеджер пакетов conda имеет несколько дополнительных функциональностей pip, которые делают его неплохим вариантом выбора по умолчанию для управления пакетами. Менеджер пакетов conda поддерживает автоматическую установку зависимостей при инсталлировании пакета; и у него есть концепция *сред*, которые действительно полезны для создания специальных песочниц Python для каждого проекта, над которым мы работаем. Хорошей общей стратегией является инсталлирование пакетов из conda при их доступности и установка пакетов из pip в противном случае.

ПРИМЕЧАНИЕ Менеджер пакетов conda можно использовать с большинством распространенных сред командной строки. Но в целях применения conda с PowerShell вам нужна версия 4.6.0 или более поздняя. Для проверки своей версии менеджера выполните команду `conda --version`. Если вам нужно обновиться, то выполните команду `conda update conda`.

Среды Conda

Мы можем столкнуться с ситуацией, когда пакеты, необходимые нам для двух разных проектов, противоречат друг другу. Для того чтобы помочь изолировать проекты друг от друга, дистрибутив Anaconda предоставляет conda env в качестве инструмента для управления *несколькими средами*. Каждая среда является полностью независимой копией Python, содержащей только те пакеты, которые необходимы для конкретного проекта или приложения. Среды могут даже использовать разные версии Python, независимо друг от друга, причем одна среда может использовать 2.7, а другая – 3.8. Среды также отлично подходят для сотрудничества с другими разработчиками, так как мы можем отправлять нашим товарищам по команде один небольшой текстовый файл environment.yml с сообщением их conda env о том, как создавать среду, идентичную нашей.

Дополнительную информацию можно получить, обратившись по адресу: https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html.

Следуя этой стратегии, давайте начнем с создания новой среды с необходимыми вам пакетами. Примеры исходного кода этой книги, находящиеся по адресу https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp, сопровождаются файлом environment.yml, который сообщает менеджеру conda перечень пакетов, которые вам нужны в вашей новой среде. Сделайте клон исходного кода из репозитория этой книги либо его скачайте, а затем выполните следующую ниже команду в своей любимой командной оболочке:

```
conda env create -f environment.yml
```

Эта команда создает новую среду под названием `qsharp-book` с использованием пакетов из канала `conda-forge` и инсталлирует в новую среду блокнотов Jupyter Notebook, Python (версии 3.6 или более поздней) ядро IQ# для Jupyter, интерпретатор IPython, пакет NumPy, графопостроительный механизм Matplotlib и пакет QuTiP. Менеджер пакетов conda предложит вам подтвердить список пакетов, которые будут инсталлированы. В целях продолжения нажмите «y», а затем нажмите **Enter** и можете насладиться чашечкой кофе.

ДЛЯ СПРАВКИ Файл environment.yml, предоставляемый по адресу https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp, также устанавливает пакет qsharp, который обеспечивает интеграцию между Python и Комплектом инструментов для квантовой разработки (который вы установите на следующем шаге).

После того как conda закончит создавать вашу новую среду, давайте ее попробуем. Для того чтобы протестировать новую среду, сначала ее активируйте:

```
conda activate qsharp-book
```

После активирования среды qsharp-book команда python должна вызвать версию, инсталлированную в этой среде. В целях ее проверки вы можете распечатать путь к команде python в вашей среде. Запустите python, а затем выполните следующую ниже команду в командной строке Python:

```
>>> import sys; print(sys.executable)
C:\Users\Chris\Anaconda3\envs\qsharp-book\python.exe          (1)
```

(1) В зависимости от вашей системы у вас, возможно, будет другой путь.

Если ваша среда была создана успешно, то вы можете ее использовать в командной строке с IPython либо в браузере со средой Jupyter Notebook. Для того чтобы начать работу с IPython, выполните команду ipython из командной строки (сначала убедитесь, что вы активировали qsharp-book):

```
$ ipython
In [1]: import qutip as qt
In [2]: qt.basis(2, 0)
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[1.]
 [0.]]
```

Если вы хотите узнать больше об использовании пакета NumPy, то продолжите чтение главы 2. Если вы хотите узнать больше об использовании пакета QuTiP, то продолжите чтение главы 5. Если же вы хотите узнать о Комплекте инструментов для квантовой разработки, то продолжайте читать здесь.

Дополнительная информация расположена по приведенным ниже адресам:

- документация по дистрибутиву Anaconda: https://docs.anaconda.com/anaconda;
- документация пакету NumPy: http://numpy.org;
- документация по среде блокнотов Jupyter Notebook: https://jupyter-notebook.readthedocs.io/en/stable/notebook.html.

A.3 Инсталлирование Комплекта инструментов для квантовой разработки

ДЛЯ СПРАВКИ Последняя версия инструкций по инсталлированию Комплекта инструментов для квантовой разработки находится по адресу: https://docs.microsoft.com/azure/quantum/install-overview-qdk. В целях использования примеров исходного кода, прилагаемых к этой книге, убедитесь, что вы установили поддержку языка Python и среды блокнотов Jupyter Notebook, следуя руководству по инсталляции Комплекта инструментов для квантовой разработки.

Комплект инструментов для квантовой разработки от Microsoft представляет собой набор инструментов для работы и программирования на Q#, новом языке квантового программирования. Если вы инсталлировали свою среду Python с помощью дистрибутива Anaconda, то можете начать работу с Комплектом инструментов для квантовой разработки с помощью команды conda:

```
conda install -c quantum-engineering qsharp
```

Если вы предпочитаете использовать Q# с автономными программами либо вне среды conda, то можете последовать инструкциям в этом разделе, чтобы инсталлировать Комплект инструментов для квантовой разработки локально на свой компьютер.

> **ПРИМЕЧАНИЕ** Эта книга сосредоточена на использовании редактора Visual Studio Code, однако Комплект инструментов для квантовой разработки можно использовать и с любым другим текстовым редактором, следуя инструкциям командной строки в основном тексте. Комплект инструментов для квантовой разработки можно использовать с Visual Studio 2019 или более поздней ее версией, используя расширение, расположенное по адресу: https://marketplace.visualstudio.com/items?itemName=quantum.DevKit.

Используя инсталляцию редактора Visual Studio Code из среды Python, вам нужно сделать несколько вещей, чтобы иметь возможность использовать Комплект инструментов для квантовой разработки с C#, Python и Jupyter Notebook.

1. Установить пакет SDK .NET Core.
2. Установить заготовки проектов для Q#.
3. Установить расширение Комплект инструментов для квантовой разработки для редактора Visual Studio Code.
4. Установить поддержку Q# для среды блокнотов Jupyter Notebook.
5. Установить пакет qsharp для Python.

После того как вы это сделаете, у вас будет все необходимое для написания и выполнения квантовых программ, написанных на языке Q#.

Знакомьтесь, это компоненты .NET

Ответить на вопрос о том, что такое .NET, стало немного сложнее, чем раньше. Исторически сложилось так, что *.NET* был разумной аббревиатурой для *.NET Framework*, виртуальной машины и компиляторной инфраструктуры для любого из языков .NET (в частности, C#, F# и Visual Basic .NET). .NET Framework предназначен для работы только в операционной системе Windows, но сторонние дополнения, такие как Mono, существуют и для других платформ, включая macOS и Linux.

Однако пару лет назад Microsoft и .NET Foundation открыли исходные коды новой версии .NET под названием .NET Core. В отличие от .NET Framework,

.NET Core является кросс-платформенным и готовым к применению каркасом. .NET Core также намного меньше, и большая часть его функциональности подразделена на опциональные пакеты. Это значительно упрощает использование нескольких версий .NET Core на одной машине и позволяет вводить новые функциональности .NET Core без проблем с совместимостью.

Однако раздвоение .NET на Framework и Core имеет свои собственные слабости. В целях обеспечения более качественной работы .NET Core как кросс-платформенной среды программирования было изменено несколько вещей в стандартных библиотеках .NET в ключе, который не совсем совместим с .NET Framework. В целях урегулирования этой проблемы .NET Foundation представила концепцию .NET Standard, набор API, предлагаемых со стороны как .NET Framework, так и .NET Core. Как следствие SDK .NET Core можно использовать для создания библиотек для .NET Core либо .NET Standard и создания приложений для .NET Core. Многие библиотеки, представленные в Комплекте инструментов для квантовой разработки, нацелены на .NET Standard, вследствие чего программы Q# можно использовать из традиционных приложений .NET Framework либо новых приложений, созданных с использованием SDK .NET Core.

Забегая вперед, самая последняя версия .NET называется просто «.NET 5», хотя это всего лишь следующая версия после .NET Core 3.1. С .NET 5 существует только одна платформа .NET вместо .NET Framework, .NET Core и .NET Standard, что значительно снижает путаницу. Дополнительная информация находится по адресу: https://devblogs.microsoft.com/dotnet/introducing-net-5/.

Однако на данный момент .NET Core 3.1 по-прежнему является самым последним *долгосрочным релизом* платформы .NET, что делает его наилучшим вариантом выбора для написания стабильного производственного программно-информационного обеспечения до выхода следующего долгосрочного релиза .NET 6. Как следствие Комплект инструментов для квантовой разработки написан для использования .NET Core SDK 3.1.

A.3.1 Инсталлирование SDK .NET Core

В целях инсталлирования SDK .NET Core перейдите по ссылке https://dotnet.microsoft.com/download/dotnet-core/3.1 и выберите свою операционную систему из списка в верхней части страницы. В разделе «Build Apps–SDK» («Сборка приложений SDK») скачайте инсталлятор для вашей операционной системы, и вы будете готовы продолжить!

A.3.2 Инсталлирование заготовок проектов

Одна вещь, которая может отличаться от того, к чему вы привыкли, – это то, что разработка в среде .NET сосредоточена вокруг идеи *проекта*, который задает то, как компилятор вызывается для создания нового двоичного файла. Например, файл проекта C# (`*.csproj`) сообщает компилятору C# о том, какие файлы исходного кода должны быть созданы, какие библиотеки необходимы, какие предупреждения включены и выключены

и т. д. Благодаря этому файлы проекта работают аналогично файлам makefile или другим системам управления сборкой. Большая разница заключается в том, как файлы проектов в .NET Core ссылаются на библиотеки.

Язык Q# использует эту инфраструктуру в многоразовом режиме, чтобы упростить получение новых библиотек для ваших квантовых программ, таких как те, которые перечислены по адресу https://docs.microsoft.com/azure/quantum/user-guide/libraries/additional-libraries, или те, которые предоставляются разработчиками сообщества по адресу https://qsharp.community/projects/. Используя эту инфраструктуру, файл проекта Q# может задавать одну или несколько ссылок на *пакеты* в репозитории пакетов для библиотек программно-информационного обеспечения NuGet.org. В свою очередь, каждый пакет может предоставлять несколько разных библиотек. При сборке проекта, который зависит от пакета NuGet, пакет SDK .NET Core автоматически скачивает нужный пакет, а затем использует библиотеки в этом пакете для сборки проекта.

С точки зрения квантового программирования это позволяет распространять Комплект инструментов для квантовой разработки в виде небольшого числа пакетов NuGet, которые могут быть инсталлированы не на машине, а в каждом проекте. Это позволяет легко использовать разные версии Комплекта инструментов для квантовой разработки в разных проектах или включать только те части, которые требуются для конкретного проекта. В целях оказания вам помощи во время начала работы с разумным набором пакетов NuGet Комплект инструментов для квантовой разработки снабжен заготовками для создания новых проектов, которые ссылаются на все, что вам нужно.

> **ДЛЯ СПРАВКИ** Если вы предпочитаете работать в IDE, то расширение редактора Visual Studio Code для Комплекта инструментов для квантовой разработки (см. ниже) также можно использовать для создания новых проектов.

В целях инсталлирования заготовок проектов выполните следующую ниже команду в своем любимом терминале:

```
dotnet new -i "Microsoft.Quantum.ProjectTemplates"
```

После установки заготовок проектов вы сможете их использовать, снова выполнив команду `dotnet new`:

```
dotnet new console -lang Q# -o ИмяПроекта     ❶
```

❶ Не забудьте поменять «ИмяПроекта» на имя проекта, который вы хотели бы создать.

A.3.3 Инсталлирование расширения редактора Visual Studio Code

После установки редактора Visual Studio Code из https://code.visualstudio.com/ вам потребуется расширение для Комплекта инструментов для

квантовой разработки, чтобы получить поддержку языка Q# указанным редактором, включая автозаполнение, встроенную подсветку синтаксических ошибок и т. д.

В целях инсталлирования этого расширения откройте новое окно редактора Visual Studio Code и нажмите **Ctrl**+**Shift**+**X** (в Windows и Linux) или ⌘+**Shift**+**X**, чтобы открыть боковую панель расширений. В строке поиска введите `Microsoft Quantum Development Kit` и нажмите кнопку **Установить**. После того как редактор Visual Studio Code установит расширение, кнопка **Установить** изменится на кнопку **Перезагрузить**. Нажмите на нее, чтобы закрыть редактор Visual Studio Code и снова открыть его окно с установленным расширением Quantum Development Kit. В качестве альтернативы нажмите **Ctrl**+**P** или ⌘+**P**, чтобы открыть палитру команд. В палитре введите `ext install quantum.quantum-devkit-vscode` и нажмите **Enter**.

В любом случае, после того как расширение будет установлено, для того чтобы его использовать, откройте папку (**Ctrl**+**Shift**+**O** или ⌘+**Shift**+**O**), содержащую проект Q#, над которым вы хотели бы поработать. В этом месте у вас должно быть все необходимое, чтобы приступить к программированию с помощью Комплекта инструментов для квантовой разработки!

Дополнительная информация расположена по приведенным ниже адресам:

- документация по Комплекту инструментов для квантовой разработки: https://docs.microsoft.com/azure/quantum/;
- использование команды dotnet: https://docs.microsoft.com/dotnet/core/tools/dotnet;
- основы работы с редактором Visual Studio Code: https://code.visualstudio.com/docs/introvideos/basics.

A.3.4 Инсталлирование IQ# для среды блокнотов Jupyter Notebook

Выполните следующие ниже команды из вашей любимой командной строки:

```
dotnet tool install -g Microsoft.Quantum.IQSharp
dotnet iqsharp install
```

> **ДЛЯ СПРАВКИ** В некоторых инсталляциях Linux вместо этого может потребоваться выполнить команду `dotnet iqsharp install --user`.

Это сделает Q# доступным в качестве языка для блокнотов Jupyter Notebook, таких как те, которые используются в главе 7.

Приложение B
Глоссарий и краткий справочник

В этом приложении к книге содержится краткая ссылка на многие квантовые концепции, описанные в данной книге, а также на язык Q# (версия 0.15). Бóльшая часть содержимого этого приложения изложена в основном тексте, но собрана здесь для удобства.

B.1 Глоссарий

BB84 (сокращение от «Беннетт и Брассард 1984») – протокол для выполнения квантового распределения ключей (QKD) путем отправки по одному кубиту за раз.

Адъюнктная операция (adjoint operation) – квантовая операция, которая полностью обращает или откатывает действие другой квантовой операции. Операции, которые являются своим *собственным адъюнктом*, такие как `X` и `H`, являются самоадъюнктными. Если операция может быть просимулирована унитарной матрицей U, ее адъюнктная операция может быть просимулирована комплексной транспозицией U, также именуемой адъюнктом U и оформляемой на письме как $U^{\dagger}$. В Q# операции, которые имеют адъюнктные операции, обозначаются как `is Adj`.

Алгоритм (algorithm) – процедура решения задачи, обычно заданная в виде последовательности шагов.

Взаимно простое число (coprime) – два положительных целых числа, которые не имеют общих простых факторов (множителей). Напри-

мер, 21 = 3 × 7 и 10 = 2 × 5 являются взаимно простыми, в то время как 21 = 3 × 7 и 15 = 3 × 5 оба имеют 3 в качестве фактора и, следовательно, не являются взимно простыми.

Вычислительно-базисное состояние (computational basis state) – состояние, помеченное строкой классических битов. Например, $|01101\rangle$ – это вычислительно-базисное состояние на пятикубитовом реестре.

Глобальная фаза (global phase) – любые два квантовых состояния, равных с точностью до умножения на комплексное число магнитуды 1, отличаются глобальной фазой. В этом случае два состояния полностью эквивалентны. Например, $(|0\rangle - |1\rangle)/\sqrt{2}$ и $(|1\rangle - |0\rangle)/\sqrt{2}$ представляют одно и то же состояние, так как они отличаются фазой $-1 = e^{i\pi}$.

Запутанность (entanglement) – ситуация, когда состояния двух или более кубитов невозможно записать независимо. Например, если два кубита находятся в состоянии $(|00\rangle + |11\rangle)/\sqrt{2}$, тогда не существует двух однокубитовых состояний $|\psi\rangle$ и $|\varphi\rangle$ таких, что $(|00\rangle + |11\rangle)/\sqrt{2} = |\psi\rangle \otimes |\varphi\rangle$, и два кубита запутаны.

Измерение (measurement) – квантовая операция, которая возвращает классические данные о состоянии квантового реестра.

Квантовый компьютер (quantum computer) – квантовое устройство, разработанное и используемое для решения вычислительных задач, которые являются трудными для классических компьютеров.

Квантовое устройство (quantum device) – квантовая система, построенная для достижения какой-то цели или выполнения какой-то задачи.

Квантовое распределение ключей (quantum key distribution, аббр. QKD) – коммуникационный протокол для обмена случайными числами между двумя сторонами, такой, что при выполнении устройствами, которые работают правильно, безопасность протокола гарантируется квантовой механикой (в частности, теоремой о запрете клонирования).

Квантовая операция (quantum operation) – подпрограмма в квантовой программе, представляющая собой последовательность инструкций, посылаемых квантовому устройству и классическому потоку управления. Некоторые квантовые операции, такие как `X` и `H`, встроены в квантовое устройство и считаются внутренними.

Квантовая программа (quantum program) – классическая программа, которая контролирует квантовое устройство, посылая инструкции этому устройству и обрабатывая данные измерений, возвращаемые устройством. Обычно квантовые программы пишутся на языке квантового программирования, таком как Q#.

Квантовый реестр (quantum register) – коллекция кубитов. Реестр может находиться в любом вычислительно-базисном состоянии, помеченном строками классических битов либо любой их суперпозицией.

Квантовое состояние (quantum state) – состояние квантового реестра (т. е. реестра кубитов), обычно записываемого в виде вектора из 2^n комплексных чисел, где n – это число кубитов в реестре.

Квантовая система (quantum system) – физическая система, для описания и симулирования которой требуется квантовая механика.

Классический бит (classical bit) – наименьшая функциональная единица хранения и обработки в классическом компьютере. Классический бит может находиться в состоянии «0» либо «1».

Классический компьютер (classical computer) – обычный компьютер, в котором для выполнения вычислений используются законы классической физики.

Комплексное число (complex number) – число вида $z = a + bi$, где $i^2 = -1$.

Компьютер (computer) – устройство, которое на входе принимает данные и выполняет какие-то операции с этими данными.

Контролируемая операция (controlled operation) – квантовая операция, применяемая на основе состояния контрольного реестра без измерения, вследствие чего суперпозиции правильно сохраняются. Например, операция CNOT является операцией контролируемого-NOT или контролируемого-X. Аналогичным образом операция Фредкина – это операция контролируемого-SWAP. В Q# операции, которые можно контролировать, обозначаются как is `Ctl`.

Кубит (qubit) – наименьшая функциональная единица в квантовом компьютере. Один кубит может находиться в состоянии $|0\rangle$, в состоянии $|1\rangle$ или в любой их суперпозиции.

Матрицы Паули (Pauli matrices) – однокубитовые унитарные матрицы $\mathbb{1}$, X, Y и Z.

Обратимая функция (reversible function) – классическая функция, которую можно идеально инвертировать. Например, $f(x) = x$ может быть инвертирована, так как $f(f(x)) = x$. Аналогичным образом $g(x, y) = (x, x \otimes y)$ является обратимой, так как $g(g(x, y)) = (x, y)$. С другой стороны, $h(x, y) = (x, x \text{ AND } y)$ не является обратимой, так как $h(0, 0) = h(0, 1) = (0, 0)$, вследствие чего мы не можем определить значение на входе в h при наличии значения (0, 0) на выходе.

Оракул (oracle) – квантовая операция, имплементирующая классическую функцию, применяемую к квантовым реестрам.

Полупростое число (semiprime) – положительное целое число ровно с двумя простыми факторами. Например, $21 = 3 \times 7$ является полупростым, в то время как $105 = 3 \times 5 \times 7$ имеет три простых фактора и, следовательно, не является полупростым.

Правило Борна (Born's rule) – математическое выражение, которое можно использовать для предсказания вероятности квантового измерения, учитывая описания этого измерения и состояния измеряемого реестра.

Программа (program) – последовательность инструкций, которые могут быть проинтерпретированы классическим компьютером для выполнения желаемой задачи.

Собственная фаза (eigenphase) – глобальная фаза, назначенная квантовой операцией собственному состоянию. Например, собственное со-

стояние $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$ операции X имеет собственную фазу –1, так как $X|-\rangle = (|1\rangle - |0\rangle)/\sqrt{2} = -|-\rangle$.

Собственное состояние (eigenstate) – состояние, которое не изменяется при применении квантовой операции, вплоть до возможной глобальной фазы. Например, состояние $|+\rangle$ является собственным состоянием операции X, так как $X|+\rangle = |+\rangle$.

Собственное значение (eigenvalue) – при заданной матрице A число λ является *собственным значением* A, если $Ax = \lambda\vec{x}$ для некоторого вектора $\vec{x}$.

Собственный вектор (eigenvector) – при заданной матрице A вектор $\vec{x}$ матрицы A, если $Ax = \lambda x$ для некоторого числа λ.

Состояние (state) – описание физической системы или устройства, достаточно полное, чтобы иметь возможность это устройство просимулировать.

Суперпозиция (superposition) – квантовое состояние, которое может быть записано как линейная комбинация других состояний, находится в суперпозиции этих состояний. Например, $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ является суперпозицией состояний $|0\rangle$ и $|1\rangle$, в то время как $|0\rangle$ является суперпозицией $|+\rangle$ и $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$.

Теорема о запрете клонирования (no-cloning theorem) – математическая теорема, доказывающая, что не может существовать квантовой операции, которая идеально копирует квантовую информацию. Например, невозможно выполнить операцию, которая преобразовывает состояние $|\psi\rangle \otimes |0\rangle$ в $|\psi\rangle \otimes |\psi\rangle$ для произвольных квантовых состояний $|\psi\rangle$.

Унитарная матрица (unitary matrix) – матрица U такая, что $UU^\dagger = \mathbb{1}$, где $U^\dagger$ – это конъюгатная трансформанта или адъюнкт матрицы U. Аналогично классической таблице истинности, унитарная матрица – это матрица, которая описывает то, как квантовая операция преобразовывает состояние ее входных данных, давая возможность ее использовать для симулирования этой операции для произвольных входных данных.

Унитарная операция (unitary operation) – квантовая операция, которая может быть представлена унитарной матрицей. В Q# унитарные операции являются адъюнктными и контролируемыми (is `Adj + Ctl`).

Фаза (phase) – комплексное число магнитудой 1 (т. е. $a + bi$, где $|a|^2 + |b|^2 = 1$). Фаза может быть записана как $e^{i\theta}$, где θ – это действительное число. Обратите внимание, что когда это ясно из контекста, в качестве сокращения иногда само θ называется фазой.

Фазовая отдача (phase kickback) – технический прием квантового программирования для ассоциирования фазы, применяемой контролируемой квантовой операцией, с состоянием контрольного реестра вместо состояния целевого реестра. Этот прием может использоваться для конвертирования того, что в противном случае было бы глобальной фазой, применяемой унитарной операцией, в физически наблюдаемую фазу.

Фазовое оценивание (aka-оценивание фазы, phase estimation) – любой квантовый алгоритм для выяснения собственной фазы, связанной с заданным собственным состоянием квантовой операции.

B.2 Нотация Дирака

Нередко в квантовых вычислениях мы используем своего рода сжатую нотацию для векторов и матриц, именуемую *нотацией Дирака*. Это более подробно рассматривается в книге, но несколько ключевых моментов нотации Дирака кратко изложены в приведенной ниже таблице.

Нотация Дирака	Матричная нотация
$\|0\rangle$	$\begin{pmatrix}1\\0\end{pmatrix}$
$\|1\rangle$	$\begin{pmatrix}0\\1\end{pmatrix}$
$\langle 0\|$	$\begin{pmatrix}1 & 0\end{pmatrix}$
$\|+\rangle = \frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$	$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}$
$\|-\rangle = \frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$	$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}$
$\|00\rangle = \|0\rangle \otimes \|0\rangle$	$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$
$\|++\rangle = \|+\rangle \otimes \|+\rangle = \frac{1}{2}(\|00\rangle + \|01\rangle + \|10\rangle + \|11\rangle)$	$\frac{1}{2}\begin{pmatrix}1\\1\\1\\1\end{pmatrix}$
$\|0\rangle\langle 0\|$	$\begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}$
$\|1\rangle\langle 1\|$	$\begin{pmatrix}0 & 0\\0 & 1\end{pmatrix}$
$\|0\rangle\langle 0\| \otimes \mathbb{1} + \|1\rangle\langle 1\| \otimes U$ (контролируемый-U)	$\begin{pmatrix}1 & 0\\0 & U\end{pmatrix} = \begin{pmatrix}1 & \dots & 0 & 0 & \dots & 0\\ \vdots & \dots & \vdots & \vdots & \dots & \vdots\\ 0 & \dots & 1 & 0 & \dots & 0\\ 0 & \dots & 0 & u_{11} & \dots & u_{1n}\\ \vdots & \dots & \vdots & \vdots & \dots & \vdots\\ 0 & 0 & 0 & u_{m1} & \dots & u_{mn}\end{pmatrix}$, где $U = \begin{pmatrix}u_{11} & \dots & u_{1n}\\ \vdots & \dots & \vdots\\ u_{m1} & \dots & u_{mn}\end{pmatrix}$

B.3 Квантовые операции

В этом разделе кратко излагаются несколько распространенных квантовых операций, которые встречаются на протяжении всей книги. В частности, мы покажем, как вызывать каждую операцию из Q#, передавая на вход кубиты; как симулировать эту операцию в пакете QuTiP с помощью матрицы, которая воздействует на состояния унитарной матрицы, которую мы используем для симулирования каждой операции; и некоторые примеры поведения этой операции математически.

> **ПРИМЕЧАНИЕ** Полный список всех встроенных в язык Q# операций можно найти, обратившись к справочнику API Q# по адресу: https://docs.microsoft.com/qsharp/api.

Во всех примерах Q# мы исходили из наличия следующей ниже инструкции open:

```
open Microsoft.Quantum.Intrinsic;
```

Во всех примерах Python/QuTiP мы исходили из наличия следующих инструкций import:

```
import qutip as qt
import qutip.qip.operations as qtops
```

Второе допущение не используется в книге, но используется здесь для краткости.

Операции Q# воздействуют на *кубиты*, тогда как кубиты представляют операции с помощью унитарных матриц, которые умножают *состояния*. Отсюда, в отличие от операций Q#, объекты QuTiP не перечисляют свои входные данные в явном виде.

Описание	Код (Q# и QuTiP)	Унитарная матрица	Математические примеры

Битовый переворот (*X* Паули)	X(target); // Q#
	qt.sigmax() # QuTiP
		$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$X\|0\rangle = \|+\rangle$
			$X\|1\rangle = \|-\rangle$
			$X\|+\rangle = \|+\rangle$
			$X\|-\rangle = -\|-\rangle$

Битовый и фазовый переворот (*Y* Паули)	Y(target); // Q#
	qt.sigmay() # QuTiP
		$Y = \begin{pmatrix} 0 & -1i \\ 1i & 0 \end{pmatrix}$	$Y\|0\rangle = i\|1\rangle$
			$Y\|1\rangle = -i\|0\rangle$
			$Y\|+\rangle = -i\|-\rangle$
			$Y\|-\rangle = i\|+\rangle$

Фазовый переворот (*Z* Паули)	Z(target); // Q#
	qt.sigmaz() # QuTiP
		$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$Z\|0\rangle = \|0\rangle$
			$Z\|1\rangle = -\|1\rangle$
			$Z\|+\rangle = \|-\rangle$
			$Z\|-\rangle = \|+\rangle$

Адамар	H(target); // Q#
	qtops.hadamard_transform() # QuTiP
		$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$H\|0\rangle = \|+\rangle$
			$H\|1\rangle = \|-\rangle$
			$H\|+\rangle = \|0\rangle$
			$H\|-\rangle = \|1\rangle$

Описание	Код (Q# и QuTiP)	Унитарная матрица	Математические примеры

Операция контролируемый-NOT (CNOT)	CNOT(control, target); // Q# (укороченная нотация)
	Controlled X(
	[control],
	target
	); // Q#
		$U_{\text{CNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$	$U_{\text{CNOT}}(\|0\rangle \otimes \|x\rangle) = (\|0\rangle \otimes \|x\rangle)$
			$U_{\text{CNOT}}(\|1\rangle \otimes \|x\rangle) = (\|1\rangle \otimes \|\neg x\rangle)$
			$U_{\text{CNOT}}\|+-\rangle = -\|--\rangle$

	qtops.cnot() # QuTiP		

CCNOT (Тоффоли)	CCNOT(control1, control2, target); // Q# (укороченная нотация)
	Controlled X(
	[control1, control2], target
	); // Q#
		$U_{\text{CCNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$U_{\text{CCNOT}}\|000\rangle = \|000\rangle$
			$U_{\text{CCNOT}}\|110\rangle = \|111\rangle$

	qtops.toffoli() # QuTiP		
X-поворот	Rx(angle, target); // Q#	$R_x(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$	$R_x(\theta\|0\rangle) = \cos\frac{\theta}{2}\|0\rangle - i\sin\frac{\theta}{2}\|1\rangle$
	qtops.rx(angle) # QuTiP		
Y-поворот	Ry(angle, target); // Q#	$R_y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$	$R_y(\theta\|0\rangle) = \cos\frac{\theta}{2}\|0\rangle + \sin\frac{\theta}{2}\|1\rangle$
	qtops.ry(angle) # QuTiP		
Z-поворот	Rz(angle, target); // Q#	$R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$	$R_z(\theta\|0\rangle) = e^{i\theta/2}\|0\rangle$
	qtops.rz(angle) # QuTiP		
Измерить один кубит	M(target); // Q#	отсутствует	отсутствует
	отсутствует		

B.4 Язык Q#

B.4.1 Типы

В следующей ниже таблице мы используем *курсивный моноширинный шрифт* для обозначения местозаполнителя. Например, местозаполнитель *`BaseType`* в *`BaseType`*[] может означать `Int`, `Double`, `Qubit`, `(Qubit, Qubit[])` или любой другой тип Q#.

Для выразительности в некоторых примерах мы добавили тип каждого примера в качестве аннотации после каждого значения. Например, `Sin : Double -> Double` указывает на то, что `Sin` – это значение, тип которого `Double -> Double`.

Описание	Тип Q#	Примеры

Целочисленный	`Int`	`3`
		`-42`
		`108`

Вещественный	`Double`	`-3.1415` `2.17`
Булев	`Bool`	`true` `false`

Диапазон целочисленных	`Range`	`0..3`
		`0..Length(arr)`
		`12..-1..0`

Пустой кортеж	`Unit`	`()`
Результаты измерений	`Result`	`Zero` `One`

Операторы Паули	`Pauli`	`PauliI`
		`PauliX`
		`PauliY`
		`PauliZ`

Строковый литерал	`String`	`"Привет, мир!"`
Кубиты	`Qubit`	(см. инструкцию `use`)
Массивы	`BaseType[]`	`new Qubit[0]` `[42, -101]`
Кортежи	`(T1)`, `(T1, T2)`, `(T1, T2, T3)` и т. д.	`(PauliX, "X")` `(1, true, Zero)`
Функции	`InputType -> OutputType`	`Sin : Double -> Double` `Message : String -> Unit`

Операции	`InputType => OutuptType`
	`InputType => Unit is Adj`
	(если адъюнктабельная)
	`InputType => Unit is Ctl`
	(если контролируемая)
	`InputType => Unit is Adj + Ctl`
	(если и адъюнктабельная,
	и контролируемая)
		`H : Qubit => Unit is Adj + Ctl`
		`CNOT : (Qubit, Qubit) => Unit is Adj + Ctl`
		`M : Qubit => Result`
		`Measure : (Pauli[], Qubit[]) => Result`

B.4.2 Объявления и инструкции Q#

В следующей ниже таблице мы используем *курсивный моноширинный шрифт* для обозначения местозаполнителя. Например, местозаполнитель *`functionName`* в функции *`functionName`*`(input1 : Input-Type1) : OutputType` обозначает имя определяемой функции, тогда как местозаполнители `Input-Type1` и `OutputType` обозначают типы из приведенной выше таблицы.

Ключевые слова языка Q# выделены **жирным** шрифтом.

Описание	Синтаксис Q#
Комментарий до конца строки кода	`// comment text`

Документационный комментарий (перед операцией либо функцией)	`/// # Сводная информация`
	`/// тело сводной информации`
	`///`
	`/// # Описание`
	`/// тело описания`
	`///`
	`/// # Вход`
	`/// ## input1`
	`/// описания входа`

Объявление пространства имен	**namespace** *NamespaceName* {
	  // ...
	}

Объявление функции	**function** *FunctionName*(
	  input1 : *InputType1*,
	  input2 : *InputType2*,
	  ...
	) : *OutputType* {
	  // тело функции
	}

Объявление операции	**operation** *OperationName*(
	  input1 : *InputType1*,
	  input2 : *InputType2*,
	  ...
	) : *OutputType* {
	  // тело операции
	}

Объявление операции (адъюнктабельной и контролируемой)	**operation** *OperationName*(
	  input1 : *InputType1*,
	  input2 : *InputType2*,
	  ...
	) : Unit **is** Adj + Ctl {
	  // тело операции
	}

Объявление пользовательского типа	**newtype** *TypeName* = (
	  ItemType1,
	  ItemType2,
	  ...
	);

Объявление пользовательского типа с именованными элементами	**newtype** *TypeName* = (
	  ItemName1: *ItemType1*,
	  ItemName2: *ItemType2*,
	  ...
	);

Открыть пространство имен (делает элементы в пространстве имен доступными в файле Q# или ячейке блокнота)	**open** *NamespaceName*;
Открыть пространство имен с псевдонимом	**open** *NamespaceName* **as** *AliasName*; *Пример*: **open** Microsoft.Quantum.Diagnostics as Diag;
Объявление локальной переменной	**let** *name* = *value*; *Пример*: **let** foo = «Bar»;
Объявление мутируемой переменной	**mutable** *name* = *value*;
Переназначить (обновить) мутируемую переменную	**set** *name* = *newValue*;

Описание	Синтаксис Q#
Применить-и-переназначить мутируемую переменную	**set** *name operator*= *expression*; *Примеры*: **set** count += 1; **set** array w/= 2 <- PauliX; (см. оператор w/)

Классические условные инструкции	**if** *condition* {
	  // ...
	}

	if *condition* {
	  // ...
	} **else** {
	  // ...
	}

	if *condition* {
	  // ...
	} **elseif** *condition* {
	  // ...
	} **else** {
	  // ...
	}

Прокручивать массив в цикле	**for** *element* **in** *array* {
	  // тело цикла
	}
	Примечание: *array* должен быть значением типа Array

Прокручивать диапазон в цикле	**for** *index* **in** *range* {
	  // тело цикла
	}
	Примечание: *array* должен быть значением типа Range

Цикл повторять-до-достижения-успеха	**repeat** {
	  // тело цикла
	}
	until *condition*;

Цикл повторять-до-достижения-успеха с блоком fixup	**repeat** {
	  // тело цикла
	}
	until *condition*
	fixup {
	  // тело отладочного блока
	}

Цикл while (только в функциях)	**while** *condition* {
	  // тело цикла
	}

Терминировать с ошибкой	**fail** "сообщение";
Вернуть значение из функции или операции	**return** *value*;

Применить шаблон «ботинки и носки» (подробности см. в главе 7)	**within** {
	  // внешнее тело
	} **apply** {
	  // внутреннее тело
	}

Выделить один новый кубит (только в операциях)	**use** *name* = **Qubit**();
Выделить массив кубитов (только в операциях)	**use** *name* = **Qubit**[*size*];

Описание	Синтаксис Q#
Выделить кортеж кубитов и реестров (только в операциях)	**use** (*name1*, *name2*, ...) = (*QubitOrArray1*, *QubitOrArray2*, ...);

Выделить один кубит с явно заданной областью видимости (только в операциях)	**use** *name* = *Qubit*() {
	  // ...
	  // здесь имя высвобождается.
	}

B.4.3 Выражения и операторы Q#

В следующей ниже таблице мы используем *курсивный моноширинный шрифт* для обозначения местозаполнителя. Например, местозаполнитель *Type* в new *Type*[*length*] обозначает базовый тип нового массива, а *length* – его длину.

Ключевые слова языка Q# выделены **жирным** шрифтом.

Описание	Синтаксис Q#
Арифметика	+, –, *, ...
Форматировать значения в качестве строковых литералов	$"... {выражение} ..." *Пример*: $"Результат измерения составил {result}"
Конкатенировать два массива	*array1* + *array2*
Выделить массив	**new** *Type*[*length*]
Получить элемент массива	*array*[*index*]
Нарезать массив	*array*[*start*...] *array*[...*end*] *array*[*start*..*end*] *array*[*start*..*step*..*end*]
Скопировать-и-обновить элемент в массиве	*array* **w/** *index* <- *newValue* *Пример*: [10, 100, 1000] w/ 1 <- 200 // [10, 200, 1000]
Обратиться к именованному элементу пользовательского типа	*value*::*itemName* *Пример*: **let** imagUnit = Complex(0.0, 1.0); Message($"{imagUnit::Real}"); // печатает 1.0
Распаковать пользовательский тип	*value*! *Пример*: **let** imagUnit = Complex(0.0, 1.0); Message($"{imagUnit!}"); // печатает (0.0, 1.0)
Скопировать-и-обновить именованный элемент пользовательского типа	*value* **w/** *itemName* <- *newValue* *Пример*: **let** imagUnit = Complex(0.0, 1.0); **let** onePlusI = imagUnit w/ Real <- 1.0; Message($"{onePlusI!}"); // печатает (1.0, 1.0)

B.4.4 Стандартные библиотеки Q#

Мы исходили из наличия следующих ниже инструкций open:

```
open Microsoft.Quantum.Intrinsic;
open Microsoft.Quantum.Canon;
open Microsoft.Quantum.Arrays as Arrays;
open Microsoft.Quantum.Diagnostics as Diag;
```

Полный список функций, операций и пользовательских типов Q# см. по адресу: https://docs.microsoft.com/qsharp/api/.

Суффиксы A, C и CA обозначают операции, которые поддерживают входные данные, являющиеся (соответственно) адъюнктабельными, контролируемыми либо обоими.

Описание	Функция или операция	Примеры
Применить операцию к каждому элементу массива	`ApplyToEachCA`	`ApplyToEachCA(H, register);`
Вызвать операцию несколько раз	`Repeat`	`Repeat(PrintRandomNumber, 10, ());`
Вернуть первый или второй элемент в паре	`Fst` или `Snd`	▪ `Fst((1.0, false)) // 1.0` ▪ `Snd((1.0, false)) // false`
Применить операцию к каждому элементу массива или собрать результаты	`Arrays.ForEach`	`let results = ForEach(M, register);`
Вызвать функцию с каждым элементом массива	`Arrays.Mapped`	`let sines = Mapped(Sin, angles);`
Отказать, если условие является ложным	`Diag.Fact`	`Fact(2 == 2, "Ожидалось, что два равно двум.");`
Отказать, если два результата измерения не равны	`Diag.EqualityFactR`	`EqualityFactR(M(qubit), Zero, "Ожидалось, что кубит находится в состоянии |0⟩.");`
Отказать, если *гипотетическое* измерение не имеет ожидаемого результата: ▪ физически возможно только на симуляторе; ▪ не применяет измерение фактически, оставляя кубит нетронутым	`Diag.AssertMeasurement`	`AssertMeasement([PauliZ], [target], Zero, "Ожидалось, что кубит находится в состоянии |0⟩.");`
Отказать, если две операции отличаются	`Diag.AssertOperations EqualReferenced`	`AssertOperationsEqualReferenced(2, actualOperation, expectedOperation );`
Запросить симулятор показать диагностическую информацию обо всех выделенных кубитах	`Diag.DumpMachine`	`DumpMachine();`
Запросить симулятор показать диагностическую информацию о конкретном реестре	`Diag.DumpRegister`	`DumpRegister((), register);`

B.4.5 Волшебные команды IQ#

Полный список волшебных команд IQ# см. по адресу: https://docs.microsoft.com/qsharp/api/iqsharp-magic.

Описание	Волшебная команда	Пример
Просимулировать функцию или операцию	`%simulate`	`%simulate PlayMorganasGame winProbability=0.999`
Добавить новый пакет в сеанс IQ#	`%package`	`%package Microsoft.Quantum.Numerics`
Перезагрузить файлы Q# с текущими пакетами	`%workspace reload`	
Перечислить все доступные волшебные команды	`%lsmagic`	
Перечислить пространства имен, открытые в настоящее время	`%lsopen`	
Выполнить оценщика ресурсов на операции	`%estimate`	`%estimate FindMarkedItem`
Установить опции конфигурации IQ#	`%config`	`%config dump.truncateSmallAmplitudes = true`
Перечислить все функции и операции, определенные в настоящее время	`%who`	

Приложение C
Памятка по линейной алгебре

В этом приложении наша цель состоит в том, чтобы кратко освежить некоторые навыки линейной алгебры, которые будут полезны в этой книге. Мы обсудим понятия векторов и матриц, коснемся приемов работы с векторами и матрицами для представления линейных функций и способов использования языка Python и пакета NumPy для работы с векторами и матрицами.

C.1 Подходы к векторам

Прежде чем мы сможем понять, что такое кубиты, нам нужно разобраться в понятии вектора.

> *Предположим, наш товарищ приглашает друзей отпраздновать починку своего дверного звонка, и мы очень хотели бы найти его дом и отпраздновать это событие вместе с ним. Каким образом наш друг мог бы помочь нам найти свой дом?*

Набросок нашей дилеммы направления движения можно найти на рис. C.1. Векторы – это математический инструмент, который используется для представления самых разных идей – в основном всего того, что мы можем записать, составив упорядоченный список чисел:

- точки на карте;
- цвета пикселов на дисплее;
- элементы ущерба в компьютерной игре;

- скорость самолета;
- ориентация гироскопа.

Например, если мы заблудились в незнакомом городе, то кто-то может нам подсказать, как пройти, дав нам вектор с инструкциями, который направит нас *a* кварталов на восток, а затем *b* кварталов на север (отложим в сторону проблему маршрутизации вокруг зданий). Мы записываем эти инструкции в форме вектора `[[a], [b]]` (рис. C.2).

Как и обычные числа, мы можем складывать различные векторы между собой.

Рис. C.1 В поисках вечеринки у нашего друга

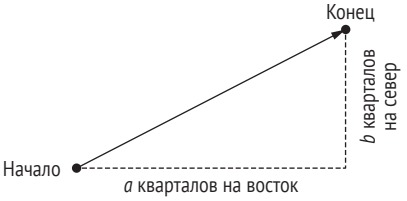


Рис. C.2 Векторы в качестве координат

ПРИМЕЧАНИЕ Обычное число часто называется *скаляром*, чтобы отличить его от вектора.

Используя этот образ мышления о векторах, мы можем рассуждать о сложении между векторами как определенном в поэлементном порядке. То есть мы интерпретируем `[[a], [b]] + [[c], [d]]` как инструкции идти `a` кварталов на восток, `b` кварталов на север, `c` кварталов на восток

и, наконец, `d` кварталов на север. Поскольку не имеет значения, в каком порядке мы выступаем, это эквивалентно движению `a + c` кварталов на восток, а затем `b + d` кварталов на север, поэтому мы пишем, что `[[a], [b]] + [[c], [d]]` равно `[[a + c], [b + d]]` (рис. C.3).

Вектор $\vec{v}$ в d размерностях можно записать в форме списка из d чисел. Например, $\vec{v}$ = [[2], [3]] представляет собой вектор в двух размерностях (рис. C.4).

Схожим образом мы можем умножать векторы на обычные числа для преобразовывания векторов. Мы можем заблудиться не просто в любом городе, например, а в городе, в котором используются метры вместо футов, как мы привыкли. В целях преобразования вектора, заданного в метрах, в вектор в футах нам нужно умножить каждый элемент нашего вектора примерно на 3.28, тем самым выполнив перемасштабирование. Давайте проделаем это с помощью Python'овского пакета под названием *NumPy*, который поможет нам управлять тем, как мы представляем векторы в компьютере.

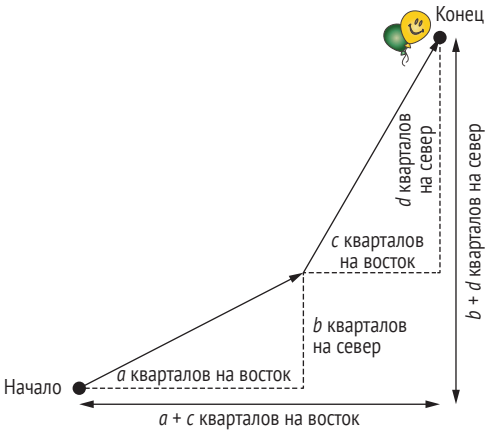


Рис. C.3. Сложение векторов, чтобы найти вечеринку

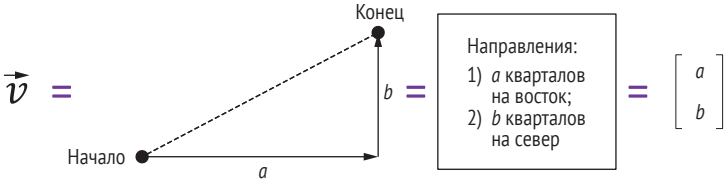


Рис. C.4 Нарисованные векторы имеют ту же информацию, что и список направлений или столбец чисел

ДЛЯ СПРАВКИ Полные инструкции по инсталляции пакета NumPy приведены в приложении A к книге.

Листинг C.1 Представление векторов на Python с помощью пакета NumPy

```
>>> import numpy as np
>>> directions_in_meters = np.array(                    (1)
...     [[30], [50]])                                   (2)
>>> directions_in_feet = 3.28 * directions_in_meters    (3)
>>> directions_in_feet
array([[ 98.4],                                         (4)
       [164. ]])
```

(1) Векторы являются частным случаем массивов NumPy. Мы создаем массивы с помощью функции array, передавая список строк в нашем векторе. Тогда каждая строка представляет собой список столбцов – для векторов у нас всегда есть только один столбец в строке, но в книге у нас будут примеры, где это не так.
(2) Начинается с примера движения на 30 метров на восток, а затем на 50 метров на север.
(3) NumPy представляет умножение между скалярами и векторами Python'овским оператором умножения *.
(4) Печатает результат умножения. Нам нужно пройти 98.4 фута на восток, а затем 164 фута на север.

Упражнение C.1: конверсия единиц измерения

Сколько будет 25 метров на запад и 110 метров на север в футах?

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

Эта структура облегчает обмен информацией о направлениях. Если бы мы не использовали векторы, то каждый скаляр нуждался бы в своем собственном направлении, и было бы очень важно держать направления и скаляры в одном месте.

C.2 Видеть матрицу своими глазами

Как мы вскоре увидим, мы можем описывать способы преобразования кубитов при применении к ним инструкций в том же ключе, в каком мы описываем преобразование векторов, используя понятие из линейной алгебры, именуемое *матрицей*. Это особенно важно, когда мы рассматриваем преобразования векторов, которые являются более сложными, чем сложение или перемасштабирование.

В целях ознакомления с тем, как использовать матрицы, давайте вернемся к задаче поиска вечеринки – в конце концов, этот дверной звонок не будет звонить сам по себе! До этого мы просто допускали, что первый компонент каждого вектора означает восток, а второй – север, но кто-то вполне мог выбрать другое правило. Без способа согласования преобразования между этими двумя правилами мы никогда не найдем вечеринку. К счастью, матрицы не только помогут нам симулировать кубиты позже в этом приложении к книге, но и помогают нам отыскивать путь к нашим друзьям.

Хорошо, что преобразование между указанием двигаться сперва на север и указанием двигаться сперва на восток легко имплементировать: мы меняем координаты `[[a], [b]]`, чтобы получить `[[b], [a]]`. Предположим, что эта перемена мест имплементирована функцией `swap`. Тогда `swap` отлично сочетается с векторным сложением, которое мы видели ранее, в том, что `swap(v + w)` всегда совпадает с `swap(v) + swap(w)`. Аналогичным образом если мы растянем вектор, а затем поменяем местами (т. е. выполним скалярное умножение), то это то же самое, как если бы мы поменяли местами, а затем растянули: `swap(a * v) = a * swap(v)`. Любая функция, обладающая этими двумя свойствами, является линейной функцией.

Линейная функция – это функция f такая, что $f(ax + by) = af(x) + bf(y)$ для всех скаляров a и b и всех векторов x и y.

Линейные функции широко распространены в компьютерной графике и машинном обучении, поскольку они включают в себя большое разнообразие способов преобразования векторов чисел.

Вот некоторые примеры линейных функций:

- повороты;
- масштабирование и растяжение;
- отражения.

Упражнение C.2: проверка линейности

Какие из следующих ниже функций являются линейными?

- $f(x) = 2^x$
- $f(x) = x^2$
- $f(x) = 2x$

Все эти линейные функции имеют общее в том, что мы можем их разбить на части и понять их по частям. Еще раз подумав о карте местности, если мы пытаемся найти дорогу на вечеринку (надеемся, еще осталось хоть немного пунша), а карта, которую нам дали, была растянута на 10 % в направлении север–юг и перевернута в направлении восток–запад, то это не так уж и трудно понять. Поскольку и растяжение, и переворачивание являются линейными функциями, кто-то может направить нас на правильный путь, рассказав нам, что конкретно произошло с направле-

нием север–юг и направлением восток–запад в отдельности. На самом деле мы только что сделали это в начале данного абзаца!

> **ДЛЯ СПРАВКИ** Если вы вынесете из этой книги хоть одну вещь, то самый важный вывод, который мы можем вам предложить, заключается в том, что вы сможете понимать линейные функции и, следовательно, квантовые операции, *разбивая их на компоненты*. В остальной части книги мы увидим, что поскольку операции в квантовых вычислениях описываются линейными функциями, то мы можем понять квантовые алгоритмы, разбив их на части так же, как мы разбили наш пример с картой. Не волнуйтесь, если в данный момент это не вполне понятно – к этому образу мышления надо еще привыкнуть.

Это связано с тем, что как только мы поймем, что происходит с северным вектором (назовем его `[[1], [0]]`, как и раньше) и западным вектором (назовем его `[[0], [1]]`), мы сможем выяснить, что происходит со *всеми* векторами, используя свойство линейности. Например, если нам говорят, что в трех кварталах к северу и в четырех кварталах к западу от нас есть реально потрясающее зрелище, и мы хотим выяснить, где оно находится на нашей карте, то мы можем сделать это по частям:

- нам нужно растянуть северный вектор на 10 % и умножить его на 3, получив `[[3.3], [0]]`;
- нам нужно перевернуть западный вектор и умножить его на 4, получив `[[0], [-4]]`;
- мы заканчиваем, складывая то, что происходит в каждом направлении, получив `[[3.3], [-4]]`.

Следовательно, нам нужно проследить по нашей карте 3.3 квартала на север и 4 квартала на восток.

Линейные функции – это нечто особенное! 💔

В предыдущем примере мы смогли растянуть наши векторы с помощью линейной функции. Это связано с тем, что линейные функции не чувствительны к масштабу. Переключение с севера на юг и с востока на запад делает с векторами то же самое, независимо от того, представлены ли они в шагах, кварталах, милях, фарлонгах или парсеках. Однако это не относится к большинству функций. Возьмем функцию, которая возводит свое входное значение в квадрат, $f(x) = x^2$. Чем больше x, тем больше он растягивается.

Линейные функции работают одинаково независимо от того, насколько велики или малы их входные данные, и именно это позволяет нам разбивать их на части: зная то, как линейная функция работает в любом конкретном масштабе, мы знаем, как она работает во *всех* масштабах.

Позже мы увидим, как биты «0» и «1» можно рассматривать как направления или векторы, не слишком отличающиеся от севера или вос-

тока. Точно так же, как север и восток не являются наилучшими векторами, помогающими понять Миннеаполис, мы обнаружим, что «0» и «1» не всегда являются наилучшими векторами, помогающими понять квантовые вычисления (рис. C.5).

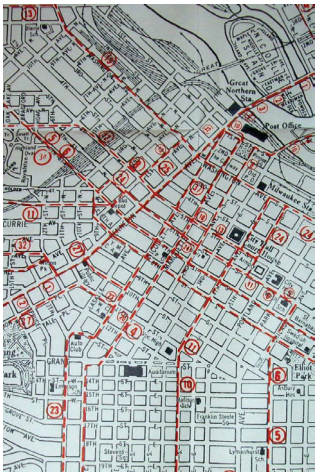


Рис. C.5 Север и запад не всегда являются наилучшими направлениями для использования, если мы хотим понять, куда идем. На этой карте центра Миннеаполиса бóльшая часть сетки центра города повернута в соответствии с изгибом реки Миссисипи. Автор фото davecito

Этот подход к пониманию линейных функций путем разбиения их на части работает и для поворотов. Если на нашей карте компас повернут на 45° по часовой стрелке (ого, нам нужен серьезный урок картографии), вследствие чего север становится северо-востоком, а запад – северо-западом, то мы все равно сможем выяснить, где что находится по частям. Используя тот же пример, северный вектор теперь на карте соотносится примерно с `[[0.707], [0.707]]`, а западный вектор соотносится с `[[-0.707], [0.707]]`.

Подытоживая то, что происходит в примере, мы получаем `3 * [[0.707], [0.707]] + 4 * [[-0.707], [0.707]]`, что равно `(3 - 4) * [[0.707], [0]] + (3 + 4) [[0], [0.707]]`, в результате давая нам `[[-0.707], [4.95]]`. Может показаться, что это связано не столько с линейностью, сколько с тем, что север и запад являются чем-то особенным. Тем не менее мы могли бы провести точно такую же аргументацию, но написать юго-запад как `[[1], [0]]` и северо-запад как `[[0], [1]]`. Это работает, потому что юго-запад и северо-запад перпендикулярны друг другу, что позволяет нам разбить любое другое направление как комбинацию северо-запада и юго-запада. Кроме простоты чтения компаса, который мы покупаем с магазинной полки, нет ничего, что делает север или запад особенными. Если вы когда-либо пытались проехать по центру Миннеаполиса (см. рис. C.5), то быстро становится очевидным, что север и запад – не всегда самый лучший способ понять направление движения!

Формально любое множество векторов, которое позволяет нам понять направления, разбивая их на части в таком ключе, называется *базисом*.

ПРИМЕЧАНИЕ В техническом плане мы здесь имеем дело с тем, что математики называют *ортонормальным базисом*, поскольку он наиболее часто приносит пользу в квантовых вычислениях. Все это означает, что векторы в базисе перпендикулярны всем другим векторам базиса и имеют длину 1.

Давайте попробуем пример написания вектора в терминах базиса. Вектор $\vec{v}$ = [[2], [3]] можно записать как $2\vec{b}_0 + 3\vec{b}_1$, используя базис $\vec{b}_0$ = [[1], [0]] и $\vec{b}_1$ = [[0], [1]]].

БАЗИС Если любой вектор $\vec{v}$ в d размерностях может быть записан в виде суммы кратных $\vec{b}_0, \vec{b}_1, \ldots, \vec{b}_{d-1}$, то мы говорим, что $\vec{b}_0, \vec{b}_1, \ldots, \vec{b}_{d-1}$ являются *базисом*. В двух размерностях один общий базис является горизонталью и вертикалью.

В более общем случае если мы знаем значения на выходе из функции *f* для каждого вектора в базисе, то мы можем вычислить *f* для любого значения на входе. Это похоже на то, как мы используем таблицы истинности для описания классической операции, перечисляя значения на выходе из операции для каждого возможного значения на входе.

Решение задач с линейностью

Допустим, `f` – это линейная функция, которая представляет то, как наша карта растягивается и скручивается. Как найти, куда нам нужно идти? Мы хотим вычислить значение `f(np.array([[2], [3]]))` (несколько произвольное значение) с учетом нашего базиса `f(np.array([[1], [0]]))` (горизонталь) и `f(np.array([[0], [1]]))` (вертикаль). Мы также знаем, глядя на части легенды карты, что карта искривляет горизонтальное направление в `np.array([[1], [1]])` и вертикальное направление в `np.array([[1], [-1]])`.

Шаги для вычисления `f(np.array([[2], [3]]))` таковы:

- мы используем наш базис, `np.array([[1], [0]])` и `np.array([[0], [1]])`, чтобы записать, что `np.array([[2], [3]])` равно `2 * np.array([[1], [0]]) + 3 * np.массив([[0], [1]])`;
- используя этот новый способ записи наших данных, подаваемых на вход в функцию, мы хотим вычислить `f(2 * np.array([[1], [0]]) + 3 * np.array([[0], [1]]))`;
- мы используем, что `f` является линейной, записав `f(2 * np.array([[1], [0]]) + 3 * np.array([[0], [1]]))` как `2 * f(np.array([[1], [0]])) + 3 * f(np.array([[0], [1]]))`:

```
>>> import numpy as np
>>> horizontal = np.array([[1], [0]])                    ❶
>>> vertical = np.array([[0], [1]])
>>> vec = 2 * horizontal + 3 * vertical                  ❷
>>> vec
array([[2],
```

```
       [3]])
>>> f_horizontal = np.array([[1], [1]])                    ❸
>>> f_vertical = np.array([[1], [-1]])
>>> 2 * f_horizontal + 3 * f_vertical                      ❹
array([[ 5],
       [-1]])
```

❶ Определяет переменные horizontal и vertical для представления базиса, который мы будем использовать для представления [[2], [3]].
❷ Мы можем написать [[2], [3]], сложив кратные переменных horizontal и vertical.
❸ Определяет, как f действует на horizontal и vertical, введя новые переменные f_horizontal и f_vertical для представления соответственно f(horizontal) и f(vertical).
❹ Поскольку f является линейной, мы можем определить, как это работает для [[2], [3]], заменив horizontal и vertical выходами из f_horizontal и f_vertical.

Упражнение C.3: вычисление линейных функций

Предположим, что у вас есть линейная функция g такая, что g([[1], [0]]) = [[2.3], [−3.1]] и g([[0], [1]]) = [[−5.2], [0.7]]. Вычислите g([[2], [−2]]).

Используя это понимание, мы можем составить таблицу того, как линейная функция преобразовывает каждый элемент своих входных данных. Эти таблицы называются *матрицами* и представляют собой полные описания линейных функций. Если мы назовем вам матрицу для линейной функции, то вы сможете вычислить эту функцию для *любого* вектора. Например, преобразование из правила север/восток в правило восток/север для направлений карты преобразовывает инструкцию «пройти одну единицу дистанции на север» из написания `[[1], [0]]` в написание `[[0], [1]]`. Аналогичным образом инструкция «пройти одну единицу дистанции на восток» переходит из написания `[[0], [1]]` в написание `[[1], [0]]`. Если сложить выходные данные для обоих множеств инструкций, то мы получим следующую ниже матрицу:

```
>>> swap_north_east = np.array([[0, 1], [1, 0]])
>>> swap_north_east
array([[0, 1],
       [1, 0]])
```

ДЛЯ СПРАВКИ Это очень важная матрица и в квантовых вычислениях тоже! Мы узнаем об этой матрице гораздо больше на протяжении всей книги.

В целях применения линейной функции, представленной матрицей, к определенному вектору мы умножаем матрицу и вектор, как показано на рис. C.6.

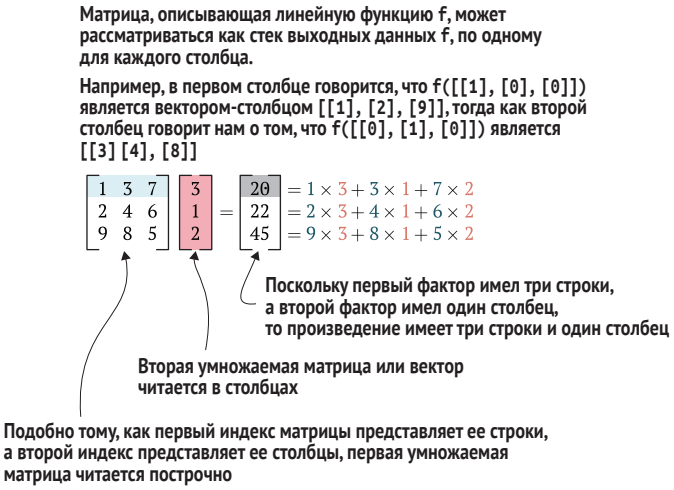


Рис. C.6 Как умножить матрицу на вектор. В этом примере матрица для *f* говорит нам о том, что f([[1], [0], [0]]) является [[1], [2], [9]]

ПРЕДУПРЕЖДЕНИЕ Порядок, в котором мы складываем векторы, не имеет значения, тогда как порядок, в котором мы умножаем матрицы, имеет огромное значение. Если повернуть нашу карту на 90°, а затем посмотреть на нее в зеркало, то мы получим совсем другую картину, чем если бы мы повернули отражение в зеркале на 90°. И поворот, и переворачивание являются линейными функциями, поэтому мы можем написать матрицу для каждой из них; давайте назовем их соответственно R и F. Если перевернуть вектор $\vec{x}$, то мы получим $F\vec{x}$. Поворот выходного значения дает нам $RF\vec{x}$, совсем другой вектор, чем если бы мы сначала повернули $FR\vec{x}$.

Матричное умножение формализует способ, которым мы вычисляли `f` при наличии ее значений на выходе для конкретного множества значений на входе, путем «укладки поверх друг друга» значений на выходе из `f` для векторов, таких как `[[1], [0], [0]]` и `[[0], [1], [0]]`, как показано на рис. C.6. Хотя фактические размеры матриц и векторов могут изменяться, идея о том, что матрица может описывать линейное преобразование, остается неизменной. В остальной части этого приложения к книге мы рассмотрим линейные преобразования на векторах длины 2. Каждая строка (самый внешний индекс в NumPy) матрицы может рассматриваться как то, каким образом функция действует на конкретное значение на входе.

Глубокое погружение: почему мы умножаем функции?

Когда мы умножаем матрицу на вектор (или даже на матрицу на матрицу), мы делаем то, что поначалу кажется немного странным. В конце концов, матрицы – это еще один способ представления линейных функций, и что же значит умножить функцию на ее входные данные, не говоря уже о еще одной функции?

Для ответа на этот вопрос полезно на мгновение вернуться к обычной алгебре, где для любых переменных a, b и c $a(b + c) = ab + ac$. Это свойство, именуемое *свойством дистрибутивности*, является фундаментальным для того, как умножение и сложение взаимодействуют друг с другом. На самом деле это настолько фундаментально, что свойство дистрибутивности является одним из ключевых способов определения понятия умножения – в теории чисел и других, более абстрактных частях математики исследователи часто работают с объектами, именуемыми *кольцами*, где все, что мы действительно знаем о умножении, – это то, что оно распределяется по сложению чисел. Несмотря на то что это понятие является абстрактным, изучение колец и других подобных алгебраических объектов имеет широкое применение, в особенности в криптографии и исправлении ошибок.

Однако дистрибутивное свойство очень похоже на свойство линейности, что $f(x + y) = f(x) + f(y)$. Если думать об f как о части кольца, то дистрибутивное свойство идентично свойству линейности.

Говоря по-другому, подобно тому, как программисты любят многократно использовать исходный код, математики любят многократно использовать *концепции*. Размышление о совместном умножении матриц позволяет нам рассматривать линейные функции во многом теми же способами, к которым мы привыкли из алгебры.

Таким образом, если мы хотим знать i-й элемент вектора $\vec{x}$, который был повернут матрицей M, то мы можем найти выходное значение из M для каждого элемента в $\vec{x}$, просуммировать полученные векторы и взять i-й элемент. В пакете NumPy матричное умножение представлено оператором @.

ПРИМЕЧАНИЕ Следующий ниже пример исходного кода работает только в Python 3.5 или более поздней версии.

Листинг C.2 Матричное умножение с помощью оператора @

```
>>> M = np.array([
...     [1, 1],
...     [1, -1]
... ], dtype=complex)
>>> M @ np.array([[2], [3]], dtype=complex)
array([[ 5.+0.j],
       [-1.+0.j]])
```

Упражнение C.4: матричное умножение

Обозначим через X матрицу [[0, 1], [1, 0]] и обозначим через $\vec{y}$ вектор [[2], [3]]. Используя пакет NumPy, вычислите $X\vec{y}$ и XX.

Почему NumPy?

Мы могли бы написать все предыдущее матричное умножение вручную, но есть несколько причин, по которым очень приятно работать с пакетом NumPy. В бóльшей части ядра пакета NumPy используется постоянно-временная индексация, а само ядро имплементировано в нативном исходном коде, поэтому оно может использовать преимущества встроенных процессорных инструкций для быстрой линейной алгебры. Как следствие пакет NumPy часто работает намного быстрее, чем манипулирование списками вручную. В листинге C.3 мы показываем пример, в котором NumPy может ускорить умножение даже очень маленьких матриц в 10 раз. Когда мы обращаемся к большим матрицам в главе 4 и последующих, использование пакета NumPy по сравнению с выполнением линейно-алгебраических вычислений вручную дает нам еще больше преимуществ.

Листинг C.3 Хронометрирование вычисления матричного умножения в NumPy

```
$ ipython                                                    ❶
In [1]: def matmul(A, B):
   ...:     n_rows_A = len(A)                                ❷
   ...:     n_cols_A = len(A[0])
   ...:     n_rows_B = len(B)
   ...:     n_cols_B = len(B[0])
   ...:     assert n_cols_A == n_rows_B                      ❸
   ...:     return [                                         ❹
   ...:         [
   ...:             sum(                                     ❺
   ...:                 A[idx_row][idx_inner] * B[idx_inner][idx_col]
   ...:                 for idx_inner in range(n_cols_A)
   ...:             )
   ...:             for idx_col in range(n_cols_B)
   ...:         ]
   ...:         for idx_row in range(n_rows_A)
   ...:     ]
   ...:
In [2]: import numpy as np                                   ❻
In [3]: X = np.array([[0+0j, 1+0j], [1+0j, 0+0j]])           ❼
In [4]: Z = np.array([[1+0j, 0+0j], [0+0j, -1+0j]])
In [5]: matmul(X, Z)
Out[5]: [[0j, (-1+0j)], [(1+0j), 0j]]
In [6]: X @ Z                                                ❽
Out[6]:
array([[ 0.+0.j, -1.+0.j],
```

```
       [ 1.+0.j, 0.+0.j]])
In [7]: %timeit matmul(X, Z)                                    ❾
10.3 µs ± 176 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
In [8]: %timeit X @ Z
926 ns ± 4.42 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

❶ На этот раз мы используем интерпретатор IPython для языка Python, поскольку он предлагает несколько дополнительных инструментов, которые полезны в данном примере. Инструкции по установке IPython см. в приложении A к книге.

❷ Находит размеры каждой матрицы, подлежащей умножению. Если мы представляем матрицы списками списков, то каждый элемент внешнего списка является строкой. То есть при такой записи матрица $n \times m$ имеет n строк и m столбцов.

❸ Для того чтобы матричное умножение имело смысл, внутренняя размерность обеих матриц должна быть согласована. Рассматривая каждую матрицу как линейную функцию, первый индекс (число строк) говорит нам о величине каждого выходного значения, тогда как второй индекс (число столбцов) говорит о величине каждого входного значения. Таким образом, нам нужно, чтобы выходы из первой функции (справа) были того же размера, что и входы во вторую функцию. Эта строка проверяет данное условие.

❹ В целях фактического вычисления матричного произведения A и B нам нужно вычислить каждый элемент в произведении и упаковать их в список списков.

❺ Мы можем найти каждый элемент, вычислив сумму там, где выходное значение из B передается в качестве входного значения в A, аналогично тому, как мы представляли матричное произведение вектором на рис. C.6.

❻ Для сравнения мы можем импортировать пакет NumPy, который предлагает имплементацию матричного умножения, использующую современные процессорные инструкции для ускорения вычислений.

❼ Инициализирует две матрицы в виде массивов NumPy в качестве тестовых случаев. Мы узна́ем гораздо больше об этих двух конкретных матрицах на протяжении всей книги.

❽ Матричное умножение в NumPy представлено оператором @ в Python 3.5 и более поздних версиях.

❾ «Волшебная команда» %timeit говорит IPython выполнить небольшой фрагмент кода Python много раз и сообщить о среднем количестве времени, которое это занимает.

C.2.1 Вечеринка при участии внутренних произведений

Есть еще одна последняя вещь, о которой нам нужно побеспокоиться, чтобы найти вечеринку. Ранее мы говорили, что игнорировали проблему наличия дороги, которая позволила бы нам идти в нужном направлении, но ведь бродить по незнакомому городу – и вправду плохая идея. Для того чтобы дойти, нам нужен способ оценить расстояние, которое мы должны пройти по данной дороге, чтобы добраться туда, куда мы идем. К счастью, линейная алгебра дает нам для этого инструмент: *внутреннее произведение* (рис. C.7). Внутренние произведения – это способ проецирования одного вектора $\vec{v}$ на другой вектор $\vec{w}$, сообщающий нам о том, сколько «тени» $\vec{v}$ отбрасывает на $\vec{w}$.

Мы можем вычислить внутреннее произведение двух векторов, умножив их соответствующие элементы и просуммировав результат. Обратите внимание, что этот рецепт умножения и суммирования совпадает с тем, что мы делаем при матричном умножении! Умножение матрицы, которая имеет одну строку, на матрицу с одним столбцом делает именно

то, что нам нужно. Таким образом, чтобы найти проекцию $\vec{v}$ на $\vec{w}$, нам нужно превратить $\vec{v}$ в вектор-строку, взяв его *транспозицию*, записываемую как $\vec{v}^{T}$.

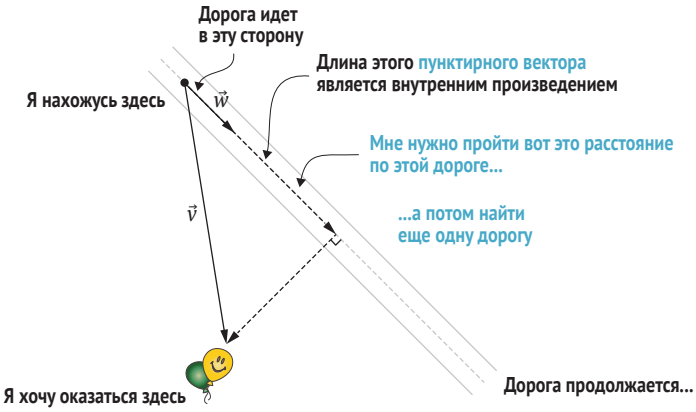


Рис. C.7 Как найти вечеринку с помощью внутренних произведений

Пример

Транспозиция $\vec{w}$ = (3 2) равна $\vec{w}^{T}$ = (2 3).

ПРИМЕЧАНИЕ В главе 3 мы увидим, что нам также нужно взять комплексный конъюгат каждого элемента, но пока мы отложим это в сторону.

В частности, матричное произведение $\vec{v}^{T}$ (транспозиции $\vec{v}$) с $\vec{w}$ дает нам матрицу 1×1, содержащую нужное нам внутреннее произведение. Предположим, нам нужно пройти два квартала на юг и три квартала на восток, но мы можем идти только по дороге, которая указывает на юго-юго-восток. Поскольку нам все еще нужно ехать на юг, эта дорога поможет нам добраться туда, куда нам нужно. Но как далеко мы должны пройти, прежде чем эта дорога перестанет помогать?

Листинг C.4 Вычисление векторных точечных произведений с помощью NumPy

```
>>> import numpy as np
>>> v = np.array([[-2], [-3]])                    (1)
>>> south_east = np.array([[1], [-1]])            (2)
>>> np.linalg.norm(south_east)                    (3)
1.4142135623730951                                (4)

>>> w = np.array([[1], [-1]]) / np.sqrt(2)        (5)
>>> np.linalg.norm(w)                             (6)
0.9999999999999999
```

```
>>> v.transpose()                    ❼
array([[-2, -3]])
>>> v.transpose() @ w                ❽
array([[ 0.70710678]])               ❾
```

❶ В этом случае $\vec{v}$ – это вектор, описывающий то, куда нам нужно идти: два квартала на север и три квартала на восток.
❷ Если доступная дорога указывает на юго-восток, то она идет на один квартал на юг для каждого квартала на восток.
❸ Мы можем найти длину вектора, используя теорему Пифагора, взяв сумму абсолютных значений каждого элемента, а затем взяв квадратный корень. В NumPy это делается с помощью функции np.linalg.norm, так как длина вектора иногда также называется его нормой.
❹ Длина [[1], [–1]], следовательно, равна $\sqrt{(+1)^2 + (-1)^2} = \sqrt{2} \approx 1.4142$.
❺ Когда мы определяем $\vec{w}$ как направление на юго-восток, нам нужно разделить на $\sqrt{2}$.
❻ Проверяя, мы видим, что длина $\vec{w}$ теперь приближенно равна 1.
❼ Транспозиция превращает $\vec{v}$ = [[–2], [–3]] в «строку» [[–2, –3]].
❽ Затем мы можем умножить транспозицию $\vec{v}$ на $\vec{w}$ так же, как мы ранее умножали матрицы на векторы.
❾ Нам нужно пройти $1/\sqrt{2} \approx 0.707$ квартала по этой дороге, прежде чем она перестанет помогать нам добраться до вечеринки.

Упражнение C.5: нормализация векторов

Дан вектор [[2], [3]]; найдите вектор, который указывает в том же направлении, но длиной 1.

Подсказка: это можно сделать с помощью внутреннего произведения либо функции `np.linalg.norm`.

Наконец-то мы добрались до вечеринки (только немного опоздали) и готовы опробовать этот новый дверной звонок!

Квадратные корни и длины

Квадратный корень из числа x – это число $y = \sqrt{x}$ такое, что мы получаем x обратно, когда возводим в квадрат y, $y^2 = x$. Мы часто используем квадратные корни на протяжении всей книги, поскольку они необходимы для определения длины векторов. Например, в компьютерной графике быстрое отыскание длин векторов необходимо для обеспечения надлежащей работы игр (см. https://en.wikipedia.org/wiki/Fast_inverse_square_root с забавной историей о том, как квадратные корни используются в играх).

Независимо от того, описывают ли векторы то, как мы добираемся до вечеринок, либо эти векторы описывают информацию, которую представляет квантовый бит, мы используем квадратные корни, чтобы рассуждать об их длинах.

Приложение D
Разведывательный анализ алгоритма Дойча–Йожи на примере

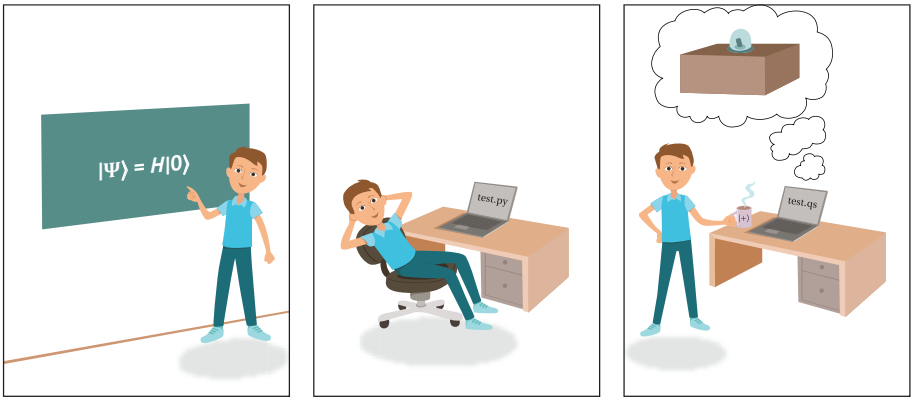
В этом приложении к книге мы уйдем с головой в алгоритм Дойча–Йожи, чтобы показать принцип его работы и то, каким образом можно использовать навыки и инструменты, разработанные в главе 8, чтобы проверить наше понимание. Мы имплементируем алгоритм Дойча–Йожи в главе 7 и активно используем пакет QuTiP для проверки нашей математики в определенных шагах.

D.1 Использование наших навыков, чтобы пробовать новое

В главах 2 и 5 мы учимся использовать пакеты NumPy и QuTiP для симулирования того, как состояния кубитов преобразовываются, когда мы посылаем инструкции квантовому компьютеру. Мы эффективно используем эти пакеты, которые выполняют математические вычисления за нас, чтобы выяснять, что происходит с квантовыми состояниями. Это похоже на подход «Дать компьютеру делать математику» на рис. D.1.

При программировании более крупных алгоритмов на Q# мы можем использовать и подход «Дать компьютеру делать математику», и немного из подхода «Нажимать все кнопки», чтобы помочь нам предсказывать то, что конкретная операция будет делать. Три подхода, показанных на

рис. D.1, используемых вместе, являются мощными инструментами решения задач при изучении квантового программирования. Если мы застрянем, используя один подход, то мы всегда можем попробовать другой, чтобы посмотреть, а не поможет ли он.



Писать математические формулы | Дать компьютеру делать математику | Нажимать кнопки и смотреть, что происходит

Рис. D.1 Три разных подхода к изучению принципа работы квантовой программы или алгоритма

Давайте попробуем применить этот комбинированный подход к алгоритму Дойча–Йожи из главы 8. В следующем ниже листинге показаны четыре шага алгоритма.

Листинг D.1 Четыре шага алгоритма Дойча–Йожи

```
H(control);
X(target);
H(target);
```
Подготовить входное состояние |+−⟩.

```
oracle(control, target);                    ❶
```

```
H(target);
X(target);
```
Откатить подготовку на целевом кубите.

```
set result = MResetX(control);              ❷
```

❶ Применить оракула.
❷ Наконец, измерить в *X*-базисе.

Ключом к пониманию принципа работы алгоритма Дойча–Йожи является понимание шага, на котором мы вызываем оракула, `oracle(control, target)`. Однако, прежде чем мы сможем к нему перейти, нам нужно понять шаг 1, на котором мы готовим наше состояние, подаваемое на вход в оракул.

D.2 Шаг 1: подготовить входное состояние для алгоритма Дойча–Йожи

Давайте воспользуемся языком Python, чтобы попытаться понять, что происходит, когда мы готовим наше состояние |+−⟩. Операции, которые мы используем для подготовки входного состояния на Q#, таковы:

```
H(control);
X(target);
H(target);
```

Каждая применяемая здесь операция представляет собой однокубитовый вентиль, поэтому мы можем рассмотреть, что происходит с каждым кубитом независимо. Давайте посмотрим, что происходит с контрольным кубитом после операции Адамара. Мы используем пакет QuTiP для моделирования подготовки состояния контрольного кубита:

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> H = hadamard_transform()                          (1)
>>> H
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper,
➥ isherm = True
Qobj data =
[[ 0.70710678  0.70710678]                            (2)
 [ 0.70710678 -0.70710678]]
>>> control_state = H * qt.basis(2, 0)                (3)
>>> control_state
Quantum object: dims = [[2], [1]], shape = (2, 1),
➥ type = ket                                         (4)
Qobj data =
[[0.70710678]
 [0.70710678]]
```

(1) В то время как H на языке Q# является инструкцией, hadamard_transform в пакете QuTiP дает нам унитарную матрицу, которую мы можем использовать для симулирования того, как инструкция H преобразовывает состояния.

(2) $1/\sqrt{2} \approx 0.707$, поэтому это выходное значение говорит, что $H = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} / \sqrt{2}$.

(3) В пакете QuTiP можно получать вектор для состояния |0⟩, вызвав basis(2, 0). 2 говорит, что нам нужен кубит (необходима размерность |0⟩), тогда как 0 говорит, что мы хотим, чтобы состояние имело значение |0⟩. Поскольку |+⟩ = H|0⟩, это устанавливает control_state равным |+⟩.

(4) Используя, что $1/\sqrt{2} \approx 0.707$, мы читаем это как говорящее нам о том, что $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$.

Это было довольно просто: контрольный кубит теперь находится в состоянии |+⟩. Теперь давайте рассмотрим подготовку целевого кубита в следующем ниже фрагменте кода.

```
>>> target_state = H * (qt.sigmax() * qt.basis(2, 0))   (1)
>>> target_state
```

```
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket      ❷
Qobj data =
[[ 0.70710678]
 [-0.70710678]]
```

❶ Повторяет ту же операцию H, что и раньше, но на этот раз на $X|0\rangle = |1\rangle$.
❷ QuTiP говорит о том, что $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$: то же самое, что и $|+\rangle$, но с перевернутым знаком в $|1\rangle$.

Теперь, когда мы увидели, как подготовить каждый кубит, давайте попросим QuTiP помочь нам написать состояние входного *реестра*.

```
>>> register_state = qt.tensor(control_state, target_state)           ❶
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket    ❷
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]
 [-0.5]]
```

❶ Как и в главе 4, мы комбинируем состояния разных кубитов, чтобы получить состояние всего реестра кубитов с помощью функции tensor.
❷ QuTiP говорит нам, что $|+\rangle \otimes |-\rangle = |+-\rangle = (|00\rangle - |01\rangle + |10\rangle - |11\rangle)/2$. То есть у нас есть равная суперпозиция по всем четырем возможным вычислительно-базисным состояниям со знаком минус перед вычислительно-базисными состояниями, в которых целевой кубит находится в состоянии $|1\rangle$.

ПРИМЕЧАНИЕ Как мы видим в главе 4, при описании состояния многокубитовой системы тензорные произведения бывают слегка многословными. Отсюда мы часто записываем многокубитовые состояния, такие как $|0\rangle \otimes |1\rangle$, конкатенируя их метки внутри одного кета, как в $|01\rangle$. Аналогичным образом $|+-\rangle$ является тем же самым, что и $|+\rangle \otimes |-\rangle$.

D.3 Шаг 2: применить оракула

Подготовив входные данные, давайте вернемся к ядру алгоритма Дойча–Йожи, где мы обращаемся к нашему оракулу:

```
oracle(control, target);
```

Подобно тому, как мы понимаем такие операции, как `H(control)`, записывая состояние контрольного кубита и применяя унитарный оператор H к этому состоянию, мы можем понять, что делает оракул U_f, проанализировав его действие на состояние, которое мы ему передаем.

Вспомните нашу конфигурацию игры в главе 8, где Нимуэ и Мерлин играют в Воспитателя короля. Наш квантовый оракул оперирует двумя кубитами, что ставит вопрос о том, как мы должны интерпретировать

каждый из этих кубитов. В классическом случае интерпретация входных и выходных классических битов из *f* была бы ясна: Нимуэ задала однобитовый вопрос и получила однобитовый ответ.

В целях понимания того, что конкретно каждый кубит для нас делает, вспомним, что при использовании обратимой классической функции нам также нужны два входных значения: первый действует как вопрос, который мы задаем в необратимом случае, а второе значение дает нам место для ответа (в качестве напоминания см. рис. D.2).

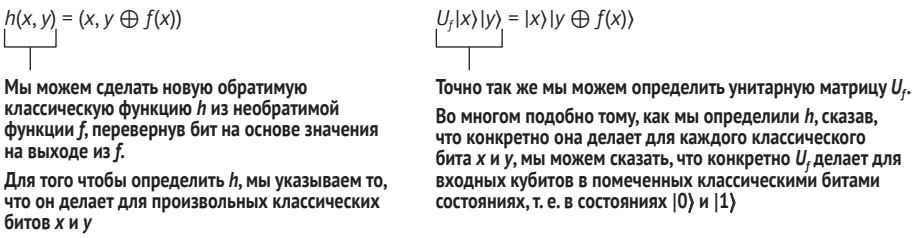


Рис. D.2 Конструирование обратимых классических функций и унитарных матриц из необратимых классических функций

Мы можем рассуждать об оракуле примерно так же: первый кубит (`control` в приведенном выше фрагменте кода) представляет наш вопрос, в то время как второй кубит (`target`) дает нам некоторое представление о том, где Мерлин может применить свой ответ. Эта интерпретация имеет смысл, когда `control` начинается в состоянии $|0\rangle$ либо $|1\rangle$, но как интерпретировать этот случай, когда мы передаем оракулу кубиты в состоянии $|+-\rangle$? Наш контрольный кубит начинается в состоянии $|+\rangle$, но $f(+)$ не имеет никакого смысла. Поскольку *f* является классической функцией, ее входным значением должен быть либо 0, либо 1 – мы не можем передавать + в классическую функцию *f*. Может показаться, что мы зашли в тупик. Однако, к счастью, существует способ это выяснить.

Квантовая механика линейна, а это означает, что мы всегда можем понять, что конкретно делает квантовая операция, разбив ее на ее действие на репрезентативном множестве состояний.

ДЛЯ СПРАВКИ Как мы видим в главе 2, множество состояний, которые можно использовать в таком ключе, называется базисом.

В целях понимания того, что делает наш оракул, когда контрольный кубит находится в состоянии $|+\rangle$, мы можем использовать тот факт, что $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$, чтобы разбить действие оракула на то, что он делает с $|0\rangle$, плюс его действие с $|1\rangle$, а затем просуммировать обе части (обязательно разделите на $\sqrt{2}$ в конце). Это помогает, потому что вместо того, чтобы путаться, пытаясь понять, что означает «$f(+)$», мы можем свести действие U_f к случаям, которые мы знаем, как вычислять, подобным $f(0)$ и $f(1)$!

Вычислительно-базисные состояния

Разложение действия квантовой операции с точки зрения того, как она действует на |0⟩ и |1⟩, очень распространено в квантовом программировании. Учитывая, насколько это полезно, мы используем для этих двух входных состояний специальное имя и называем |0⟩ и |1⟩ *вычислительным базисом*, чтобы отделить их от других базисов, которые мы могли бы использовать, таких как |+⟩ и |−⟩.

Использование линейности для понимания квантовых операций не ограничивается одним кубитом, как мы увидим в остальной части данного приложения к книге. Например, для двух кубитов вычислительный базис состоит из состояний |00⟩, |01⟩, |10⟩ и |11⟩.

Если у нас есть еще больше (скажем, пять) кубитов, то мы можем записать такие состояния, как |1⟩ ⊗ |0⟩ ⊗ |0⟩ ⊗ |1⟩ ⊗ |0⟩, как строки в таком же ключе, получив |10010⟩. Вычислительный базис для пяти кубитов можно записать в виде {|00000⟩, |00001⟩, |00010⟩, …, |11110⟩, |11111⟩}.

В более общем случае, если у нас есть n кубитов, вычислительный базис состоит из всех строк из n классических битов, каждая из которых является меткой кета. Иными словами, вычислительный базис для многокубитовой системы состоит из всех тензорных произведений |0⟩ и |1⟩: т. е. всех состояний, помеченных строкой классических битов.

При таком подходе к разбору принципа работы оракула давайте рассмотрим некоторые примеры оракулов, которые мы имплементировали в главе 8.

D.3.1 Пример 1: оракул id

Предположим, что дан оракул, который имплементирует стратегию, в которой Мерлин выбирает королем Артура (раздел 8.2). Напомним, что классической однобитовой функцией, представляющей эту стратегию, является `id`. Из табл. D.1 мы знаем, что это означает, что U_f имплементируется инструкцией CNOT, поэтому давайте посмотрим, что она делает с `register_state`.

Таблица D.1 Представление однобитовой функции в виде двухкубитовых оракулов

Имя функции	Функция	Значение на выходе из оракула	Операция Q#
`id`	$f(x) = x$	$\|x\rangle\|y \oplus x\rangle$	`CNOT(control, target)`

ДЛЯ СПРАВКИ Напомним, что инструкция контролируемого-NOT переворачивает свой второй кубит, если первый кубит находится в |1⟩.

Листинг D.2 Как оракул id преобразовывает свое входное состояние

```
>>> cnot = qt.cnot(2, 0, 1)                          ❶
>>> cnot
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
➥ isherm = True
Qobj data =                                          ❷
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]                                       ❸
 [0. 0. 1. 0.]]
>>> register_state = cnot * register_state
>>> register_state                                   ❹
Quantum object: dims = [[2, 2], [1, 1]],
➥ shape = (4, 1), type = ket
Qobj data =
[[ 0.5]
 [-0.5]
 [-0.5]
 [ 0.5]]
```

❶ Запрашивает у QuTiP матрицу, которая позволяет нам просимулировать инструкцию CNOT с помощью функции cnot. Здесь 2 указывает на то, что мы хотим просимулировать CNOT на двухкубитовом реестре, 0 указывает на то, что 0-й кубит является контрольным (control), а 1 указывает на то, что первый кубит является целевым (target).

❷ Вспомните, что унитарные операторы для квантовых вычислений являются тем же самым, что и таблицы истинности для классической логики. Каждая строка в этой таблице сообщает о том, что происходит с вычислительно-базисным состоянием.

❸ Например, строка в индексе 2 (с индексацией начиная с нуля) может быть записана как 10 в двоичном формате. Отсюда эта строка является вектором, который мы получим, если входное значение равно |10⟩, и это говорит, что инструкция CNOT оставляет кубиты в |11⟩ (3 в десятичном формате, следовательно, в третьем столбце есть 1).

❹ QuTiP говорит о том, что реестр с контрольным и целевым кубитами теперь находится в состоянии (|00⟩ – |01⟩ – |10⟩ + |11⟩)/2.

Теперь, когда мы разобрались с действием оракула `id`, давайте посмотрим, что делает с входным состоянием оракул `not`.

D.3.2 Пример 2: оракул not

Давайте повторим анализ, используя оракула `not`, другую сбалансированную функцию. Оракул, представляющий Мерлина, выбирающего Мордреда, имплементируется с помощью серии операций X и CNOT, как показано в табл. D.2.

Таблица D.2 Однобитовая функция not как двухкубитовый оракул

Имя функции	Функция	Значение на выходе из оракула	Операция Q#
`not`	$f(x) = \neg x$	$\|x\rangle\|y \oplus \neg x\rangle$	`X(control); CNOT(control, target); X(control);`

Давайте перейдем на язык Python, чтобы увидеть, как разложить работу оракула not.

Листинг D.3 Повторное использование пакета QuTiP, теперь с оракулом not

```
>>> control_state = H * qt.basis(2, 0)                                    ❶
>>> target_state = H * qt.basis(2, 1)
>>> register_state = qt.tensor(control_state, target_state)
>>> I = qt.qeye(2)                                                        ❷
>>> X = qt.sigmax()
>>> oracle = qt.tensor(X, I) * qt.cnot(2, 0, 1) *
... qt.tensor(X, I)                                                       ❸

>>> oracle
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
➥ isherm = True
Qobj data =                                                               ❹
[[0. 1. 0. 0.]                                                           ❺
 [1. 0. 0. 0.]
 [0. 0. 1. 0.]                                                            ❻
 [0. 0. 0. 1.]]
>>> register_state = oracle * register_state
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket       ❼
Qobj data =
[[-0.5]
 [ 0.5]
 [ 0.5]
 [-0.5]]
```

❶ Подготавливает контрольный и целевой кубиты в состоянии |+−⟩ точно так же, как и раньше.
❷ Как и в главе 5, полезно определить переменные I и X в качестве укороченной нотации соответственно для матрицы тождественности (qt.qeye) и матрицы, представляющей операцию X.
❸ На этот раз наш оракул является оракулом not, который мы имплементируем с помощью последовательности инструкций X(control); CNOT(control, target); X(control); в соответствии с табл. D.2.
❹ Унитарный оператор для оракульной операции на этот раз выглядит немного иначе: он переворачивает целевой кубит, когда контрольный кубит равен |0⟩.
❺ Например, строка 0 (00 в двоичном формате) сообщает о том, что |00⟩ преобразовывается в |01⟩.
❻ Аналогичным образом строка 2 (10 в двоичном формате) говорит о том, что |10⟩ преобразовывается в |10⟩; оракул оставляет это входное значение нетронутым.
❼ Состояние после применения оракула равно (−|00⟩ + |01⟩ + |10⟩ − |11⟩)/2 = (−1)|+−⟩, точно так же, как и раньше, за исключением глобальной фазы −1.

Глядя на эти два примера, мы получили одно и то же выходное состояние, за исключением того, что все знаки перевернуты. Это означает, что если умножить один из векторов состояния на −1, то они оба будут одинаковыми. Умножение всего вектора на константу называется *добавлением глобальной фазы*. Поскольку глобальные фазы не могут наблюдаться с помощью измерений, мы получили точно такую же информацию, применив оракулы id и not. Мы ничего не узнали о том, какой оракул, id

или `not`, мы применяли; и если бы мы могли сравнить векторы, то мы бы узнали только то, что применили сбалансированного оракула.

Для сравнения давайте посмотрим, как выглядит реестр после применения оракула, представляющего *постоянную функцию*.

D.3.3 Пример 3: оракул zero

Еще раз и с чувством: давайте воспользуемся Python для разложения на части принципа работы оракула, представляющего постоянную функцию `zero`. Мы хотим использовать оракул `zero`, чтобы показать, что происходит по-другому, когда мы применяем оракула, представляющего постоянную функцию. Этот оракул особенно прост в применении, поскольку он состоит в том, чтобы вообще не применять никаких инструкций. Вы можете увидеть все способы представления этого правила в табл. D. 3.

Таблица D.3 Однобитовая функция zero в качестве двухкубитовых оракулов

Имя функции	Функция	Значение на выходе из оракула	Операция Q#
`zero`	$f(x) = 0$	$\|x\rangle\|y \oplus 0\rangle = \|x\rangle\|y\rangle$	(empty)

В листинге D.4 мы видим, что ничегонеделание на контрольном кубите и ничегонеделание на целевом кубите можно просимулировать, ничего не делая на всем реестре. Отсюда создаваемый нами оракул является двухкубитовой матрицей тождественности $\mathbb{1} \otimes \mathbb{1}$ для оракула `zero`.

Листинг D.4 Вычисление преобразования оракула zero

```
>>> control_state = H * qt.basis(2, 0)
>>> target_state = H * qt.basis(2, 1)
>>> register_state = qt.tensor(control_state, target_state)
>>> X = qt.sigmax()
>>> oracle = qt.tensor(I, I)
>>> oracle
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
➥ isherm = True
Qobj data =
[[1. 0. 0. 0.]                                          ❶
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
>>> register_state = oracle * register_state
>>> register_state                                      ❷
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]
 [-0.5]]
```

1. **Ничегонеделание на контрольном кубите и ничегонеделание на целевом кубите можно просимулировать, ничего не делая на всем реестре.**
2. **Выходное состояние отличается от оракула id более чем на глобальную фазу. Нет такого скаляра, который мы могли бы умножить на глобальную фазу, чтобы превратить выходное значение id в выходное значение zero.**

Здесь мы видим первое отличие от предыдущего листинга: знаки минуса находятся в разных местах на векторе состояния. До этого мы использовали оракул `id` и получили, что выходное состояние было (|00⟩ – |01⟩ – |10⟩ + |11 ⟩)/2. Используя оракул `zero`, выходное состояние равно (|00⟩ – |01⟩ + |10⟩ – |11⟩)/2. Нет такого числа, на которое мы могли бы умножить весь вектор, чтобы превратить его либо в `[[0.5],[-0.5],[-0.5],[0.5]]`, либо в `[[-0.5],[0.5],[0.5],[-0.5]]`. В целях понимания того, как эта разница приводит к точному установлению, какого оракула, сбалансированного либо постоянного, мы имеем, давайте перейдем к следующему шагу алгоритма Дойча–Йожи.

Упражнение D.1: попробуйте оракул one

Попробуйте, сможете ли вы применить приемы Python, которые мы использовали ранее, чтобы выяснить, как изменяются состояния целевого и контрольного кубитов при применении оракула one.

Решения упражнений

Все решения упражнений в этой книге можно найти в прилагаемом репозитории исходного кода: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Просто перейдите в папку соответствующей главы, в которой вы находитесь, и откройте блокнот Jupyter с именем, в котором упоминаются решения упражнений.

D.4 Шаги 3 и 4: откатить подготовку на целевом кубите и измерить

В этом месте гораздо легче разобраться в выходном значении, если откатить шаги, которые мы использовали для подготовки |+−⟩, чтобы все вернулось в вычислительный базис (|00⟩ … |11⟩). В целях ревизии в табл. D.4 приведены векторы состояний для всех четырех оракулов (три из которых мы разработали ранее).

Теперь мы хотим откатить наши подготовительные шаги на целевом кубите.

Таблица D.4 Состояние реестра после применения оракулов

Имя функции	Состояние реестра после применения оракула
`zero`	[[0.5], [−0.5], [0.5], [−0.5]]
`one`	[[−0.5], [0.5], [−0.5], [0.5]]
`id`	[[0.5], [−0.5], [−0.5], [0.5]]
`not`	[[−0.5], [0.5], [0.5], [−0.5]]

Почему мы выполняем «откат подготовки» целевого кубита?

В главе 7 мы видим, что нам нужно сбрасывать кубиты в состояние |0⟩, прежде чем возвращать их на целевую машину. В этом месте наш целевой кубит всегда находится в состоянии |−⟩, *независимо от того, какого оракула мы используем*. Это означает, что после применения оракула мы точно знаем, как поставить его обратно в |0⟩. Как и в главе 7, это помогает нам избежать дополнительных измерений, которые на некоторых квантовых устройствах бывают дорогостоящими.

Обратите внимание, что мы можем безопасно вернуть целевой кубит в |−⟩, не влияя на результаты при измерении контрольного кубита, поскольку вызов оракула является единственной двухкубитовой операцией в алгоритме Дойча–Йожи. Как мы видим в главе 5, выполнение однокубитовых операций на одном кубите не может повлиять на результаты на другом кубите; в противном случае мы могли бы отправлять информацию быстрее света!

Давайте попробуем сделать это, откатив подготовку на реестре из оракула, представляющего функцию `id`.

Листинг D.5 Выходное значение оракула id в вычислительном базисе

```
>>> I = qt.qeye(2)                                                    ❶
>>> register_state_id = qt.cnot(2,0,1) *                              ❷
... (qt.tensor(H * qt.basis(2, 0), H * (qt.sigmax() * qt.basis(2, 0))))
...
>>> register_state_id =
➥ qt.tensor(I, H) * register_state_id                                ❸
>>> register_state_id
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1),
➥ type = ket                                                         ❹
Qobj data =
[[ 0. ]
 [ 0.70710678]
 [ 0. ]
 [-0.70710678]]
>>> register_state_id =
➥ qt.tensor(I, qt.sigmax()) *
➥ register_state_id                                                  ❺
>>> register_state_id
```

```
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1),
➥ type = ket                                                    ❻
Qobj data =
[[ 0.70710678]
 [ 0.        ]
 [-0.70710678]
 [ 0.        ]]
>>> qt.tensor(H * qt.basis(2, 1), qt.basis(2, 0))
➥ == register_state_id                                          ❼
True
>>> register_state_id = qt.tensor(H, I) *
➥ register_state_id                                             ❽
>>> register_state_id
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1),
➥ type = ket                                                    ❾
Qobj data =
[[0.]
 [0.]
 [1.]
 [0.]]
```

❶ Полезно определить укороченную нотацию для матрицы тождественности $\mathbb{1}$, которую мы используем для представления того, что происходит с кубитом, когда мы не применяем к нему никакой инструкции.

❷ Воспроизводит реестр для оракула, представляющего функцию id, сразу после применения оракула.

❸ Поскольку мы преобразовываем двухкубитовое состояние, нам, чтобы получить нашу матрицу, нужно сказать, что происходит с каждым кубитом. Мы делаем это, снова используя функцию tensor.

❹ Выходное значение читается гораздо легче: реестр находится в состоянии $(|01\rangle - |11\rangle)/\sqrt{2}$.

❺ В нашей программе Q# мы использовали инструкцию X для возвращения целевого кубита в $|0\rangle$ перед его выпуском, просимулировав функцией sigmax() пакета QuTiP.

❻ Поскольку инструкция X переворачивает свой аргумент, применение матрицы *X* дает нам состояние $(|00\rangle - |10\rangle)/\sqrt{2}$.

❼ Мы можем использовать пакет QuTiP, чтобы подтвердить, что еще одним вариантом написать $(|00\rangle - |10\rangle)/\sqrt{2}$ является $(H|1\rangle) \otimes |0\rangle = |-0\rangle$.

❽ Мы можем просимулировать операцию MResetX, применив H, а затем измерив в *Z*-базисе.

❾ Состояние реестра непосредственно перед измерением имеет 1 в строке 2 (10 в двоичном формате), поэтому состояние реестра равно $|10\rangle$, и мы с уверенностью получим результат измерения One.

ДЛЯ СПРАВКИ Мы используем операцию `MResetX` из стандартных библиотек Q# для измерения в *X*-базисе. *X*-базисное измерение возвращает результат `Zero`, когда его аргумент находится в $|+\rangle$, и возвращает результат `One`, когда его аргумент находится в $|-\rangle$. Следовательно, мы можем просимулировать операцию `MResetX`, применив *H*, а затем измерив в *Z*-базисе.

Глядя на окончательный вектор в листинге D.5, мы видим, что он представляет состояние $|10\rangle$. Если бы мы измеряли контрольный кубит из этого состояния, то мы бы получали классический бит в 100 % случаев.

В файле algorithm.qs, который мы пишем в главе 8, мы возвращаем пользователю, что оракул сбалансирован, если мы измерили его на контрольном кубите, поэтому мы правильно заключаем, что `id` является сбалансированным оракулом! Тот факт, что мы всякий раз будем измерять One на контрольном бите, действительно выглядит круто.

ПРИМЕЧАНИЕ Хотя некоторые квантовые алгоритмы являются рандомными, как пример с QRNG из главы 2 или игра Морганы и Ланселота из главы 7, они не обязательно должны быть такими. На самом деле алгоритм Дойча–Йожи является *детерминированным*: всякий раз, когда мы его выполняем, мы получаем один и тот же ответ.

Из этих примеров мы вынесли одно важное наблюдение (см. табл. D.5): применение оракула к контрольному и целевому кубитам может повлиять на состояние контрольного кубита.

Таблица D.5 Векторы, представляющие состояние реестра после применения различных оракулов

Имя функции	Состояние реестра непосредственно перед измерением	Результат измерения контрольного кубита по *Z*
`zero`	[[1], [0], [0], [0]]	`Zero`
`one`	[[–1], [0], [0], [0]]	`Zero`
`id`	[[0], [0], [1], [0]]	`One`
`not`	[[0], [0], [–1], [0]]	`One`

Предметный указатель

P

Q

R

S

T

U

W

X

Y

Z

A

Б

В

Г

Д

З

И

К

Р

С

Т

У

Ф

Х

Ч

Ш

Э

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Сара Кайзер и Кристофер Гранад

Изучаем квантовые вычисления на Python и Q#

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 34,94. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Изучаем квантовые вычисления на Python и Q#

Квантовые компьютеры обеспечивают радикальный скачок в скорости и вычислительной мощи. Совсем скоро могут появиться усовершенствованные научные модели и новые рубежи в криптографии, которые были немыслимы при использовании классических вычислений. Комплект инструментов для квантовой разработки от компании Microsoft и язык Q# предоставляют вам возможность поупражняться в квантовых вычислениях, даже если вы не знаете математику или физику на продвинутом уровне.

В этой книге квантовые технологии обсуждаются с практической точки зрения. Используя Python, вы сможете создать собственный квантовый симулятор, а задействуя инструменты с открытым исходным кодом от Microsoft – тонко настроить квантовые алгоритмы. Авторы покажут, как применять квантовые методы для решения практических задач, в том числе для отправки секретных сообщений.

«Квантовые вычисления – это предвестие прорыва, который уже не за горами. Данная книга вам понадобится, чтобы получить фору в игре».
Клайв Харбер, Distorted Thinking Ltd.

«Отличный вступительный курс по квантовым вычислениям».
Дмитрий Денисенко, Fyld

«Это практическое руководство по квантовым вычислениям подготовит вас к работе в кратчайшие сроки».
Томас Хейман, TechnoGems

«Замечательное введение в квантовые вычисления с отличными приложениями!»
Уильям Э. Уилер, TekSystems

Издание предназначено для разработчиков программного обеспечения. Предварительного опыта работы с квантовыми вычислениями не требуется.

Рассматриваемые темы:

- механика, лежащая в основе квантовых компьютеров;
- симулирование кубитов на языке Python;
- разведывательный анализ квантовых алгоритмов с помощью Q#;
- решение химических и арифметических задач;
- работа с данными при помощи квантовых вычислений.

Доктор Сара Кайзер работает в некоммерческой организации «Унитарный фонд» (Unitary Fund), поддерживающей квантовую экосистему с открытым исходным кодом, и является экспертом в создании квантовых технологий в лабораторных условиях. Доктор Кристофер Гранад работает в группе квантовых систем в Microsoft и специализируется на описании характеристик квантовых устройств.

ISBN 978-5-97060-935-4

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС издательство
www.дмк.рф