ООО «А-Я эксперт»

Библиотека QSimHs для симуляции квантовых вычислений

Руководство программиста

Версия 1.0
(Апрель 2023 г.)

Москва, 2023

1. Введение

QSimHs — это библиотека для симуляции квантовых вычислений, написанная на языке Haskell в компании ООО «А-Я эксперт». Она предназначена для предоставления исследователям, студентам и разработчикам гибкого и эффективного инструмента для моделирования квантовых схем и алгоритмов.

Библиотека состоит из нескольких модулей, включая модуль ядро, которое состоит из набора модулей, в которых определяются основные примитивы, используемые в квантовых вычислениях, такие как кубиты, квантовые гейты, квантовые состояния и др. Имеется несколько модулей для определения известных квантовых алгоритмов, таких как алгоритмы Дойча, Гровера, Йожи, Шора и Саймона, что позволяет легко реализовать эти алгоритмы и изучить их поведение. Модули с описанием проблемно-ориентированного языка программирования Quipper обеспечивают интуитивно понятный способ представления квантовых схем и алгоритмов.

Библиотека разработана так, чтобы быть удобной для пользователя и эффективной. Она оптимизирована для производительности, позволяя пользователям быстро и точно моделировать большие и сложные квантовые схемы. В то же время она проста в использовании, с ясным и интуитивно понятным набором модулей, который упрощает создание, манипулирование и анализ квантовых схем.

Библиотека QSimHs предназначена для студентов, исследователей и разработчиков для изучения методов квантовых вычислений.

Преимущества использования Haskell для моделирования квантовых вычислений:

1. Язык программирования Haskell — это функциональный язык программирования, который предоставляет много преимуществ для моделирования квантовых вычислений.
2. *Строгая типизация*. Строгая система типов языка Haskell гарантирует, что программы будут более надёжными и менее подверженными ошибкам. Это особенно важно в квантовых вычислениях, где даже небольшие ошибки могут привести к значительным неточностям в результатах.
3. *Ленивые вычисления*. Ленивые вычисления в языке Haskell позволяют более эффективно использовать ресурсы. Это особенно важно для квантовых вычислений, где ресурсы, такие как кубиты, ограничены и крайне дороги, а симуляция квантовых вычислений требует экспоненциально большого количества ресурсов в зависимости от числа кубитов.
4. *Выразительный синтаксис*. Выразительный синтаксис языка Haskell позволяет создавать более лаконичный и читабельный код, который легче понять и поддерживать. Это особенно важно для квантовых вычислений, где сложность алгоритмов может быстро стать непосильной.
5. *Высокая производительность*. Высокая производительность языка Haskell делает его хорошо подходящим для моделирования больших и сложных квантовых схем. Библиотека QSimHs использует все преимущества производительности языка Haskell для обеспечения быстроты и точности моделирования.

В целом, язык программирования Haskell обеспечивает мощную и гибкую среду для моделирования квантовых вычислений, позволяя исследователям, студентам и разработчикам уверенно и эффективно изучать модель квантовых вычислений.

Этот документ представляет собой исчерпывающее руководство по библиотеке QSimHs, включая инструкции по установке, описание модулей библиотеки, примеры

использования, советы по устранению неполадок и заключение. В частности, в документе рассматриваются следующие темы:

1. *Введение*: даётся краткий обзор библиотеки QSimHs, её назначения и возможностей.
2. *Установка библиотеки*: описывается, как загрузить и установить библиотеку QSimHs в системе пользователя.
3. *Начало работы*: содержит краткое введение в библиотеку и её основные возможности, включая модули ядра, алгоритмов и взаимодействия с языком Quipper.
4. *Описание модулей библиотеки*: содержит подробное описание модулей библиотеки, включая модуль ядра, алгоритмов и взаимодействия с языком Quipper, а также типы и функции, которые они определяют.
5. *Примеры использования*: содержит несколько примеров использования библиотеки для моделирования квантовых схем и алгоритмов, включая факторизацию с помощью алгоритма Гровера, определение гейта Тоффоли и решение различных задач.
6. *Устранение неполадок*: содержит решения некоторых распространенных проблем, с которыми пользователи могут столкнуться при работе с библиотекой.
7. *Заключение*: суммирует основные моменты, рассмотренные в документе, и дает некоторые заключительные мысли о библиотеке и ее потенциальных приложениях.

Библиотека QSimHs — это мощная и гибкая библиотека для моделирования схем и алгоритмов квантовых вычислений, написанная на языке программирования Haskell. Библиотека состоит из четырёх частей: ядра, модулей с алгоритмами, модулей для взаимодействия с проблемно-ориентированным языком Quipper и модулей с примерами использования.

Ядро содержит базовые определения всех примитивов, таких как кубит, квантовая схема, квантовый вентиль. Этот набор модулей является основой библиотеки, и все остальные модули строятся на его основе.

Модули с алгоритмами определяют известные квантовые алгоритмы, такие как алгоритмы Дойча, Гровера, Йожи, Шора и Саймона. Эти алгоритмы могут быть использованы в качестве строительных блоков для более сложных схем и приложений.

Модули для взаимодействия с языком Quipper содержат краткое введение в проблемно-ориентированный язык Quipper, который можно использовать для определения квантовых схем и алгоритмов в более краткой и удобочитаемой форме.

Наконец, модули с примерами содержат несколько примеров использования библиотеки для моделирования квантовых схем и алгоритмов. Эти примеры варьируются от факторизации с помощью алгоритма Гровера до определения гейта Тоффоли и решения различных задач.

В целом, библиотека QSimHs разработана как простая в использовании и достаточно гибкая, чтобы удовлетворить потребности широкого круга пользователей, от новичков до продвинутых исследователей. Благодаря строгой системе типизации, ленивым вычислениям, выразительному синтаксису и высокой производительности, библиотека QSimHs является мощным инструментом для изучения модели квантовых вычислений.

2. Установка библиотеки

Перед установкой библиотеки QSimHs необходимо убедиться, что система пользователя соответствует следующим требованиям:

- В системе развёрнута платформа Haskell версии 8.6.3 или более поздней.
- Современный процессор с поддержкой инструкций SSE2.
- Не менее 8 Гб оперативной памяти.
- Любая операционная система, на которой может быть запущена платформа Haskell.

Чтобы использовать библиотеку QSimHs, сначала необходимо загрузить и установить её в системе пользователя. В следующих инструкциях описано, как это сделать:

1. Необходимо установить язык программирования Haskell:

Прежде чем устанавливать библиотеку QSimHs, необходимо установить язык программирования Haskell в системе пользователя (если он не установлен). Для этого необходимо использовать инструкции по установке для операционной системы пользователя на официальном сайте Haskell Platform: https://www.haskell.org/platform/.

2. Необходимо загрузить библиотеку QSimHs:

После установки языка Haskell можно загрузить библиотеку QSimHs из её официального репозитория на GitHub: https://github.com/Roman-Dushkin/QSimHs. Можно загрузить исходный код в виде zip-файла или клонировать репозиторий с помощью инструментов Git.

3. Необходимо развернуть библиотеку QSimHs:

Чтобы развернуть библиотеку QSimHs, необходимо скопировать её исходные коды в отдельную папку в операционной системе.

4. Необходимо проверить установку:

Чтобы проверить, что библиотека QSimHs была установлена правильно, можно попробовать импортировать её в модуль на языке Haskell. Следует открыть интерпретатор GHCi, выполнив команду ghci, после чего ввести следующую команду:

```
Prelude> import Qubit
```

Если библиотека импортируется без ошибок, значит, установка прошла успешно. Если возникли какие-либо ошибки, следует обратиться к разделу 6 «Устранение неполадок» настоящего руководства.

В случае необходимости и требования дополнительной информации можно обратиться к документу «Руководство по установке» для библиотеки QSimHs.

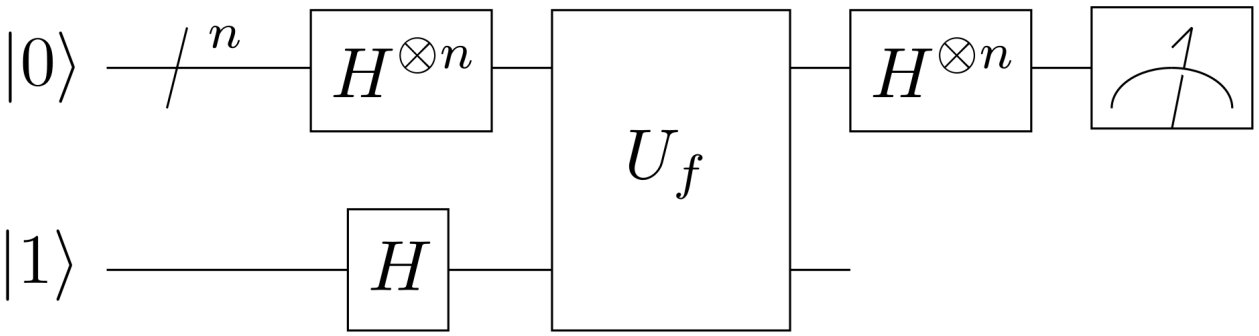
3. Начало работы

В этом разделе даётся краткое введение в библиотеку QSimHs и рассказывается о том, как её использовать для моделирования квантовых схем. Он включает информацию о том, как определить квантовые схемы, гейты и кубиты, и как запустить моделирование с помощью библиотеки. Этот раздел служит отправной точкой для тех, кто только знакомится с библиотекой QSimHs и интересуется моделированием квантовых вычислений.

Квантовый алгоритм Йожи — это квантовый алгоритм, который может быть использован для решения определённого типа проблем, называемых «проблемой оракула». Алгоритм был предложен Ричардом Йожей в 1992 году, и он может определить, является ли функция постоянной или сбалансированной, с помощью одного запроса к функции, в то время как классические алгоритмы требуют нескольких запросов. Алгоритм Йожи обеспечивает сверхлинейное ускорение по сравнению с классическими алгоритмами для

этого типа задач и является важным примером того, как квантовые вычисления могут показать квантовое превосходство над классическими вычислениями.

На следующем рисунке приведена классическая схема последовательности квантовых гейтов, реализующих алгоритм Йожи:



Описанная далее реализация реализована на языке Haskell и использует библиотеку QSimHs для моделирования квантовой схемы алгоритма Йожи.

- Функция `jozsa` реализует квантовый алгоритм Йожи. Она принимает на вход унитарную матрицу, представляющую функцию-оракул, и количество кубитов, используемых для представления входа в эту функцию-оракул. Он применяет гейт Адамара ко всем кубитам, применяет матрицу функции-оракула, а затем применяет ещё один гейт Адамара ко всем кубитам. Наконец, она измеряет кубиты и возвращает полученную строку.
- Функция `makeOracle` — это вспомогательная функция, используемая для создания матрицы функции-оракула. Она принимает на вход классическую функцию, которая отображает списки булевых значений в одно булево значение, и количество кубитов, используемых для представления входа в функцию. Она генерирует матрицу 2^n x 2^n, которая реализует функцию, используя стандартную технику инверсии фазы состояний, соответствующих входам, для которых функция возвращает истину.
- Функция `oracle` — это утилитарная функция, реализующая классические функции-оракулы, на которых работает алгоритм. Она принимает на вход целое число, определяющее, какую функцию-оракул использовать, и список булевых значений, представляющих входные данные для выбранной функции-оракула. Она возвращает булево значение, которое является результатом применения функции-оракула к входным данным.
- Функция `histogram` — специальная функция, которая принимает два аргумента: квантовая схема для запуска и количество раз, которые должна быть запущена квантовая схема. Эта функция возвращает словарь, где каждый ключ — это целочисленное значение, которое наблюдалось в результатах измерений, а соответствующее значение — частота этого результата. По сути, функция создает гистограмму результатов измерений.
- Функция `main` — это функция-драйвер, которая запускает алгоритм Йожи на всех 9 функциях-оракулах и генерирует гистограмму результатов. В качестве входных данных она принимает количество повторений алгоритма для каждой функции-оракула.

Так выглядит таблица функций-оракулов, которые готовятся для исследований алгоритма Йожи при помощи функции `oracle`:

x_1	x_2	x_3	f

			1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	0	0	1	1	1
0	1	1	0	0	0	0	0	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1
1	0	1	0	0	0	1	1	1	1	1	1
1	1	0	0	0	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1

Пример гистограммы для функции-оракула № 4:

О функции-оракуле № 4 можно сказать, что она почти сбалансирована, поэтому опять с более чем 55 % вероятности в результате выходит уже значение |100> — единица в первом кубите показывает, что функция-оракул № 4 сбалансирована именно по первому кубиту (необходимо посмотреть на её определение).

4. Описание модулей библиотеки

В этом разделе будет дано полное описание модулей, составляющих библиотеку QSimHs. Каждый модуль предназначен для предоставления набора инструментов и функций для помощи в реализации и выполнении квантовых алгоритмов, в частности, связанных со

всеми примитивами для квантовых вычислений. Будет рассмотрено назначение каждого модуля и дан обзор функций, которые он содержит.

Список модулей:

- Модули ядра: QuantumState, Gate, Circuit, Qubit.
- Модули алгоритмов: Deutsch, Grover, Jozsa, Shor, Simon.
- Модули для интеграции с языком Quipper: QuipperSimple, QuipperHard.
- Модули с примерами и решениями задач: GF, Toffoli, Tasks.

4.1. Модуль QuantumState

Модуль `QuantumState` предоставляет набор функций и тип данных для представления и манипулирования квантовыми состояниями. Модуль содержит следующее:

- Тип данных `QuantumState`, который представляет одно квантовое состояние. Он содержит два поля: амплитуда квантового состояния и его метка.
- Функция `toPair`, которая принимает значение `QuantumState` и возвращает пару, состоящую из амплитуды и метки.
- Функция `fromPair`, которая принимает пару из амплитуды и метки и возвращает значение `QuantumState`.
- Функция `applyQS`, которая принимает функцию и значение `QuantumState` и применяет функцию к амплитуде квантового состояния.
- Функция `predicateQS`, которая принимает предикат и значение `QuantumState` и возвращает результат применения предиката к амплитуде квантового состояния.
- Функция `conjugateQS`, которая принимает значение `QuantumState` и возвращает его комплексное сопряжение.

Модуль также определяет пользовательский экземпляр класса `Show` для типа `QuantumState`, который обеспечивает человеко-читаемое строковое представление квантового состояния.

4.2. Модуль Gate

Это модуль, который описывает программные объекты для представления и обработки квантовых гейтов. Он содержит функции для основных операций над векторами и матрицами, таких как умножение матрицы на вектор, вычисление смежных матриц и выполнение различных матричных операций.

Модуль также включает функции для преобразования между векторным и матричным представлениями гейтов и соответствующими им представлениями комплексных чисел. Он определяет такие гейты, как квантовые гейты тождества, стандартные гейты X, Y, Z, гейт Адамара и гейт CNOT (управляемое HE), а также функции для построения произвольных n-кубитовых гейтов.

Состав модуля:

- Тип `Vector` — вектор как наименование для списка.
- Тип `Matrix` — матрица как наименование для списка списков.

- Функция `apply` — функция, вычисляющая произведение матрицы на вектор. В итоге получается вектор. Разработчик должен сам следить за корректностью размерностей матрицы и вектора, подаваемых на вход этой функции.
- Функция `adjoint` — функция для получения эрмитово-сопряжённой матрицы для заданной.
- Оператор `<|>` — оператор для получения внутреннего произведения двух векторов.
- Оператор `|><|` — оператор для получения внешнего произведения двух векторов.
- Оператор `<*:>` — оператор для умножения матрицы на число. Первым аргументом получает число, а вторым, соответственно, матрицу.
- Оператор `<**>` — оператор для перемножения (обычного) матриц. Как обычно, разработчик должен сам следить за корректностью размерностей перемножаемых матриц.
- Оператор `<++>` — оператор для тензорного перемножения матриц.
- Оператор `<+>` — оператор для сложения двух матриц. Разработчик самостоятельно должен следить за корректностью размерностей входных матриц, чтобы получать адекватные результаты.
- Оператор `<->` — оператор для вычитания матриц друг из друга. Разработчик самостоятельно должен следить за корректностью размерностей входных матриц, чтобы получать адекватные результаты.
- Функция `vectorToComplex` — сервисная функция для перевода списка счётных чисел в список комплексных чисел. Используется для преобразования кубитов в каноническом представлении с целыми коэффициентами при квантовых состояниях.
- Функция `matrixToComplex` — ещё одна сервисная функция для перевода матрицы счётных чисел в матрицу комплексных чисел. Используется для преобразования унитарных матриц (квантовых операторов) в матричном представлении с целыми коэффициентами.
- Функция `vectorToInt` — сервисная функция для получения простого для восприятия вектора, в котором одни целые числа. Неадекватна к применению для произвольных векторов.
- Функция `matrixToInt` — ещё одна сервисная функция для получения простой для восприятия матрицы, в которой одни целые числа. Неадекватна к применению для произвольных матриц.
- Функция `gateI` — константная функция, возвращающая матричное представление квантового гейта **I** (тождественное преобразование).
- Функция `gateX` — константная функция, возвращающая матричное представление квантового гейта **X** (отрицание).
- Функция `gateY` — константная функция, возвращающая матричное представление квантового гейта **Y** (изменение фазы).
- Функция `gateZ` — константная функция, возвращающая матричное представление квантового гейта **Z** (сдвиг фазы).
- Функция `gateH` — константная функция, возвращающая матричное представление квантового гейта **H** (преобразование Адамара).
- Функция `gateCNOT` — константная функция, возвращающая матричное представление квантового гейта **CNOT** (контролируемое-НЕ).
- Функция `gateN` — функция, реализующая заданный гейт для заданного количества кубитов.

- Функция `gateHn` — функция, реализующая гейт Адамара для заданного количества кубитов.
- Функция `gateIn` — функция, реализующая тождественный гейт для заданного количества кубитов.

В целом, этот модуль обеспечивает основу для реализации квантовых алгоритмов и запуска их симуляций на языке Haskell.

4.3. Модуль Circuit

Модуль `Circuit` — это модуль для организации квантовых вычислений при помощи определения квантовых схем. Он предоставляет функции и операторы для манипулирования квантовыми гейтами, квантовыми состояниями и кубитами. Модуль реэкспортирует модули Gate, QuantumState и Qubit.

Модуль определяет функцию `ylppa` и операторы (`|>`) и (`>>>`). Функция `ylppa` — это специальный синоним функции `apply` из модуля `Gate`, которая изменяет порядок аргументов. Эта функция предназначена для прямого обозначения последовательностей применения гейтов к кубитам. Оператор (`|>`) является синонимом функции `ylppa` и представляет собой более читаемую альтернативу. Оператор (`>>>`) является синонимом оператора (`$`), используемого для передачи потока управления в функцию измерения.

Модуль определяет уровень приоритета оператора `|>` равным 6, а оператора `>>>` равным 5.

В целом, модуль `Circuit` предоставляет основные функции и операторы для организации и выполнения квантовых вычислений.

4.4. Модуль Qubit

Модуль Qubit является важной частью квантовых вычислений, которые опираются на фундаментальную единицу квантовой информации, известную как кубиты. В этом подразделе мы рассмотрим основы кубитов, их свойства и то, как они используются в квантовых схемах для выполнения сложных вычислений. Мы также обсудим различные типы кубитов и проблемы, связанные с их созданием и обслуживанием.

Состав модуля:

- Тип `Qubit` — тип, представляющий кубит. Это просто список квантовых состояний.
- Функция `groups` — сервисная функция для разбиения заданного списка на подсписки заданной длины.
- Функция `changeElement` — служебная функция для замены элемента в заданном списке на заданной позиции (счёт начинается с 1) на заданное значение.
- Функция `quantumStates` — служебная функция для получения из кубита списка его квантовых состояний.
- Функция `toList` — функция для преобразования кубита в вектор (список). На выходе получается просто список амплитуд.
- Функция `toLabelList` — функция для преобразования кубита в список меток его квантовых состояний.

- Функция `fromLists` — функция для создания кубита из двух списков — списка комплексных амплитуд и списка меток. Разработчик сам должен следить за тем, что семантика кубита должна выполняться (например, при полном задействовании всех базисных состояний их количество должно составлять степень двойки).
- Функция `toPairList` — функция для преобразования кубит в список пар.
- Функция `fromPairList` — функция для создания кубита из списка пар — пар комплексных амплитуд и меток. Разработчик сам должен следить за тем, что семантика кубита должна выполняться (например, при полном задействовании всех базисных состояний их количество должно составлять степень двойки).
- Функция `toVector` — функция для преобразования кубита к векторному представлению в стандартном базисе.
- Функция `fromVector` — функция для создания кубита из векторного представления в стандартном базисе.
- Функция `liftQubit` — сервисная функция для «втягивания» заданной функции в кубит и применения её к списку квантовых состояний.
- Функция `entangle` — функция для связывания двух кубитов в одну систему из нескольких кубитов. По сути, производит тензорное умножение кубитов друг на друга.
- Функция `conjugateQubit` — функция для получения комплексно-сопряжённого кубита для заданного.
- Функция `scalarProduct` — функция для вычисления скалярного (внутреннего) произведения двух заданных кубитов.
- Функция `norm` — функция для получения нормы вектора, представляющего собой кубит (то есть его длину).
- Функция `normalize` — функция для нормализации заданного кубита, то есть для получения из кубита нового, норма (длина) которого равна в точности 1. Это значит, что у результирующего кубита сумма квадратов модулей амплитуд равна 1, и выполняется условие нормированности.
- Функция `measure` — функция для осуществления процесса измерения заданного кубита. В зависимости от распределения амплитуд вероятности выбирает одно из квантовых состояний кубита и возвращает его метку.
- Функция `measureP` — функция для осуществления процесса частичного измерения заданного кубита. В зависимости от распределения амплитуд вероятности выбирает одно из частичных квантовых состояний, определяемых набором индексов (второй аргумент). Возвращает пару (измеренная метка, оставшиеся квантовые состояния).
- Функция `getRandomElementWithProbabilities` — служебная функция для получения случайного элемента из заданного списка с учётом распределения вероятностей. Список должен содержать пары, первым элементом которых являются возвращаемые элементы, а вторым — вероятность. Значения вероятности не обязательно должны быть нормированы.
- Функция `qubitZero` — константная функция для представления кубита |0> в стандартном вычислительном базисе.
- Функция `qubitZero'` — константная функция для представления кубита |0> в стандартном вычислительном базисе в виде вектора.
- Функция `qubitOne` — константная функция для представления кубита |1> в стандартном вычислительном базисе.

- Функция `qubitOne'` — константная функция для представления кубита |1> в стандартном вычислительном базисе в виде вектора.
- Функция `qubitPlus` — константная функция для представления кубита |+> в стандартном вычислительном базисе.
- Функция `qubitPlus'` — константная функция для представления кубита |+> в стандартном вычислительном базисе в виде вектора.
- Функция `qubitMinus` — константная функция для представления кубита |-> в стандартном вычислительном базисе.
- Функция `qubitMinus'` — константная функция для представления кубита |-> в стандартном вычислительном базисе в виде вектора.
- Функция `qubitPhiPlus` — константная функция для представления кубита Φ+ (первое состояние Белла) в стандартном вычислительном базисе.
- Функция `qubitPhiPlus'` — константная функция для представления кубита Φ+ (первое состояние Белла) в стандартном вычислительном базисе в виде вектора.
- Функция `qubitPhiMinus` — константная функция для представления кубита Φ- (второе состояние Белла) в стандартном вычислительном базисе.
- Функция `qubitPhiMinus'` — константная функция для представления кубита Φ- (второе состояние Белла) в стандартном вычислительном базисе в виде вектора.
- Функция `qubitPsiPlus` — константная функция для представления кубита Ψ+ (третье состояние Белла) в стандартном вычислительном базисе.
- Функция `qubitPsiPlus'` — константная функция для представления кубита Ψ+ (третье состояние Белла) в стандартном вычислительном базисе в виде вектора.
- Функция `qubitPsiMinus` — константная функция для представления кубита Ψ- (четвёртое состояние Белла) в стандартном вычислительном базисе.
- Функция `qubitPsiMinus'` — константная функция для представления кубита Ψ- (четвёртое состояние Белла) в стандартном вычислительном базисе в виде вектора.

В модуле также определён пользовательский экземпляр класса `Show` для типа `Qubit`, который обеспечивает человеко-читаемое строковое представление для кубита.

4.5. Модуль Deutsch

В этом подразделе описываются программные сущности модуля `Deutsch`, в котором определяются все необходимые функции и примитивы, необходимые для реализации квантового алгоритма Дойча. Алгоритм Дойча — это простейший (и первый в истории) квантовый алгоритм, который демонстрирует потенциальную мощь квантовых вычислений, решая свою задачу быстрее, чем это мог бы сделать любой классический компьютер. Модуль `Deutsch` реализует этот алгоритм, причём его реализация выполнена «с нуля» без использования примитивов библиотеки QSimHs для демонстрации того, как это громоздко (эту реализацию можно сравнить с модулем Jozsa).

Состав модуля:

- Тип `Vector` — синоним типа для представления вектора. Просто список. При использовании разработчик должен самостоятельно следить за размерностью вектора.
- Тип `Matrix` — синоним типа для представления матрицы. Список списков. Опять же, при использовании этого типа разработчик всегда сам должен следить как за размерностью

списка списков, так и за размерностями каждого из списков (предполагается, что они должны быть одинаковыми).

- Функция `f1` — первая функция (0) для проверки при помощи алгоритма Дойча в классическом исполнении.
- Функция `f2` — вторая функция (1) для проверки при помощи алгоритма Дойча в классическом исполнении.
- Функция `f3` — третья функция (id) для проверки при помощи алгоритма Дойча в классическом исполнении.
- Функция `f4` — четвёртая функция (not) для проверки при помощи алгоритма Дойча в классическом исполнении.
- Функция `deutsch` — классическая реализация алгоритма Дойча. Используется два вызова функции, переданной на проверку.
- Функция `testDeutsch` — функция для тестирования классической реализации алгоритма Дойча.
- Функция `vectorToComplex` — сервисная функция для перевода списка счётных чисел в список комплексных чисел. Используется для преобразования кубитов в каноническом представлении с целыми коэффициентами при квантовых состояниях.
- Функция `matrixToComplex` — ещё одна сервисная функция для перевода матрицы счётных чисел в матрицу комплексных чисел. Используется для преобразования унитарных матриц (квантовых операторов) в матричном представлении с целыми коэффициентами.
- Функция `qubitZero` — кубит |0>.
- Функция `qubitOne` — кубит |1>.
- Функция `entangle` — функция для связывания двух кубитов в одну систему из нескольких кубитов. По сути, производит тензорное умножение кубитов друг на друга.
- Функция `gateX` — константная функция, возвращающая матричное представление квантового гейта **X** (отрицание).
- Функция `gateH` — константная функция, возвращающая матричное представление квантового гейта **H** (преобразование Адамара).
- Функция `apply` — функция, вычисляющая произведение матрицы на вектор. В итоге получается вектор. Разработчик должен сам следить за корректностью размерностей матрицы и вектора, подаваемых на вход этой функции.
- Оператор `|>` — специальный синоним для функции `apply`, который меняет порядок аргументов. Этот оператор предназначен для прямой записи последовательности применения гейтов к кубитам.
- Оператор `<*:>` — оператор для умножения матрицы на число. Первым аргументом получает число, а вторым, соответственно, матрицу.
- Оператор `<++>` — оператор для тензорного перемножения матриц.
- Функция `deutsch'` — функция, реализующая квантовый алгоритм Дойча, который за один вычислительный шаг осуществляет проверку, является ли заданная функция, выраженная как унитарное преобразование, константной или сбалансированной. Функция должна быть бинарной и от одного аргумента (всего может быть 4 вида таких функций).
- Функция `testDeutsch'` — функция для тестирования квантовой реализации алгоритма Дойча.
- Функция `f1'` — унитарное преобразование для представления квантового оракула функции f x = 0.

- Функция `f2'` — унитарное преобразование для представления квантового оракула функции f x = 1.
- Функция `f3'` — унитарное преобразование для представления квантового оракула функции f x = x.
- Функция `f4'` — унитарное преобразование для представления квантового оракула функции f x = not x.

4.6. Модуль Grover

Модуль `Grover` — это реализация на языке Haskell в составе библиотеки QSimHs квантового алгоритма Гровера, который представляет собой квантовый алгоритм поиска, позволяющий осуществлять поиск в несортированной базе данных из N элементов за время O(sqrt(N)), что значительно быстрее любого классического алгоритма. Модуль предоставляет необходимые функции и примитивы для реализации алгоритма Гровера, включая оракул, гейт диффузии и главную функцию для тестирования алгоритма Гровера.

Функции модуля:

- Функция-оракул — это матричное (квантовое) представление заданной классической функции, и модуль предоставляет два примера такого оракула: функции `oracle` и `oracle'`. Первая — это специально созданный оракул для демонстрации алгоритма Гровера на трёх кубитах, а функция `oracle'` соответствует функции, которая возвращает 1 для нескольких (трёх) определённых значений.
- Функция `diffusion` реализует оператор диффузии, который представляет собой комбинацию гейта Адамара и гейта для инверсии фазы, используемых для усиления амплитуды помеченного состояния.
- Функция `grover` является основной функцией модуля, которая принимает на вход матрицу функции-оракула и применяет необходимые квантовые гейты для выполнения алгоритма Гровера на трёхкубитной системе. Функция возвращает результат измерения конечного состояния системы, представленный в виде двоичной строки.
- Функция `main` строит гистограмму результатов измерений, выполняя алгоритм Гровера заданное число раз, группируя и подсчитывая результаты.

В целом, модуль `Grover` предоставляет комплексную реализацию алгоритма квантового поиска Гровера на языке Haskell, которая может быть использована в образовательных целях и для понимания практической реализации квантовых алгоритмов.

4.7. Модуль Jozsa

Модуль `Jozsa` реализует квантовый алгоритм Йожи, который является алгоритмом, разработанным для решения проблемы Дойча-Йожи. Проблема может быть сформулирована следующим образом: дана функция-оракул в виде чёрного ящика, которая принимает n-битные вход и возвращает однобитный выход. Необходимо определить, является ли эта функция постоянной (возвращает один и тот же выход для всех входов) или сбалансированной (возвращает разные выходы для двух разных половин возможных входов).

Модуль состоит из нескольких функций, которые работают вместе для реализации алгоритма.

- Функция `makeOracle` принимает функцию-оракул и целое число n в качестве входных данных и возвращает квантовую схему, реализующую оракул. Схема представлена в виде матрицы комплексных чисел.
- Функция `jozsa` является основной функцией модуля. Она принимает на вход матрицу, представляющую функцию-оракул, и целое число n и возвращает результат измерения конечного состояния квантовой схемы. Схема строится путем запуска с n кубитов, инициализированных в нулевое состояние, применения гейта Адамара к каждому кубиту, применения оракула, а затем повторного применения гейта Адамара к каждому кубиту. Результатом является строка, представляющая результат измерения.
- Функция `histogram` — это вспомогательная функция, которая принимает монадическое действие и целое число n в качестве входных данных, выполняет действие n раз, собирает результаты и возвращает список пар, представляющих частоту каждого уникального результата.
- Функция `main` является исполняемой функцией модуля. Она применяет функцию `histogram` к функции `jozsa` для каждого из девяти возможных оракулов и возвращает список гистограмм, представляющих результаты измерений для каждого оракула.
- Наконец, существует функция `oracle`, которая представляет собой набор заранее определённых функций-оракулов, используемых для тестирования алгоритма. Оракулы пронумерованы от 1 до 9, и каждый из них принимает на вход список из трёх булевых значений и возвращает один булевый выход. Оракулы разработаны таким образом, что некоторые из них являются постоянными, а некоторые сбалансированными, что обеспечивает целый ряд тестовых примеров для алгоритма.

4.8. Модуль Shor

Модуль `Shor` — это реализация квантового алгоритма Шора для нахождения простых множителей заданного целого числа. Модуль определяет функции для построения квантовых схем, реализующих алгоритм, а также функции для измерения результатов работы схемы для извлечения простых множителей.

Модуль импортирует несколько функций и типов из модулей `Circuit`, `Gate` и `Qubit`. Он определяет несколько вспомогательных функций, включая функцию для вычисления наибольшего общего делителя двух целых чисел с помощью евклидова алгоритма и функцию для вычисления модульного экспоненцирования числа, возведенного в степень по модулю другого числа. Эти вспомогательные функции используются в квантовой схеме для реализации алгоритма Шора.

Модуль Shor определяет функцию, которая принимает целое число n и возвращает простые множители этого числа. Сначала функция проверяет, является ли n чётным или степенью простого числа, в этом случае она возвращает соответствующую факторизацию. В противном случае она строит квантовую схему, используя комбинацию гейтов Адамара, гейтов модульного экспонирования и гейтов квантового преобразования Фурье. Схема предназначена для выполнения подпрограммы поиска периода в рамках алгоритма Шора.

Подпрограмма поиска периода реализуется с помощью квантового преобразования Фурье, которое выполняется с помощью серии контролируемых вращений кубита в гильбертовом пространстве. Результат квантового преобразования Фурье измеряется для получения периода модульной экспоненциальной функции.

Затем функция использует измеренный период для вычисления простых множителей числа n с помощью алгоритма продолженных дробей. Наконец, функция проверяет, являются ли вычисленные коэффициенты действительно простыми, и возвращает их, если да.

Модуль также определяет функцию `main`, которая выполняет алгоритм Шора. Функция строит квантовую схему, запускает схему несколько раз, чтобы получить несколько образцов периода, и вычисляет простые множители из измеренных периодов.

Состав модуля:

- Изоморфный тип `Chain` — тип для представления цепной дроби, в которую переводится отношение задействованного количества кубитов к найденному в результате выполнения квантовой процедуры числу.
- Функция `nofShorAttempts` — количество попыток запуска алгоритма Шора для факторизации. Чем больше это число, тем больше вероятность найти разложение с первой попытки, однако тем дольше работает функция `main`.
- Функция `nofQFCalling` — количество попыток запуска квантовой подпрограммы в рамках алгоритма Шора. Чем больше это число, тем больше вероятность найти нужный период в первом же запуске полного алгоритма Шора, однако тем медленнее работает функция сбора вероятных периодов.
- Константная функция `numberToFactor` — число, которое будет факторизовываться при помощи квантового алгоритма Шора.
- Константная функция `simpleNumber` — число, при помощи которого будет осуществляться поиск разложения заданного числа.
- Константная функция `nofAncillas` — количество вспомогательных кубитов, которое требуется для факторизации заданного числа.
- Константная функция `nofWorkingQubits` — количество рабочих кубитов, которое требуется для факторизации заданного числа.
- Константная функция `nofQubits` — общее число потребных кубитов, которые требуются для факторизации заданного числа.
- Функция `periodicFunction` — специальная функция, используемая в алгоритме Шора, период которой необходимо найти. Реализована в виде обычной функции, а не в виде квантового оракула.
- Константная функция `oracleList` — предварительно вычисленная константа для построения оракула в виде матрицы. Число обозначает регистр кубитов в векторном представлении (в десятичном выражении; то есть в этом регистре 9 кубитов), в который преобразуется квантовый регистр, векторное представление которого соответствует номеру числа в этом списке, при этом нумерация начинается с 0. То есть кубит \|000000000>\ преобразуется в \|000000001>\ и т. д.
- Функция `makeOracleList` — сервисная функция для построения списка, на основе которого строится оракул для заданной функции. Данный список необходимо использовать в функции `oracle` вместо вызова константы `oracleList`. Если вызвать эту функцию следующим образом: `makeOracleList 4 5 periodicFunction`, то в результате получится список, в точности равный списку `oracleList`.
- Функция `oracle` — подготовленный оракул для функции `periodicFunction` в двоичном представлении.

- Функция `qubitFromInt` — сервисная функция, которая готовит одну строку для создания оракула.
- Функция `qft` — функция для построения гейта, выполняющего квантовое преобразование Фурье.
- Функция `quantumFactoring` — основная функция модуля, которая демонстрирует квантовый алгоритм Шора для факторизации целых чисел.
- Функция `getPeriod` — функция, которая возвращает период по измеренному значению входного (первого) квантового регистра.
- Функция `toChain` — функция, которая переводит дробь, обратную заданной, в цепную. Предполагается, что аргумент меньше единицы.
- Функция `fromChain` — функция, которая переводит цепную дробь в обычную.
- Функция `fromChainLimited` — функция, которая находит приближённое значение отношения заданного в виде цепной дроби числа, полученного в результате измерения выхода квантовой процедуры, к периоду. При этом для найденного значения отношения число бит в числителе и знаменателе получается не более \n\.
- Функция `findRandomPeriod` — функция для получения случайного периода посредством запуска заданное количество раз квантового алгоритма Шора.
- Функция `collectPeriods` — функция для сбора большинства возможных периодов для числа, которое факторизуется при помощи алгоритма Шора.
- Функция `getFactors` — служебная функция, которая получает на вход период, а возвращает пару делителей числа `numberToFactor`.
- Функция `main` — главная функция модуля, которая используется для запуска квантового алгоритма Шора для факторизации числа `numberToFactor` и вывода на экран его простых множителей.
- Функция `investigate` — функция для проведения исследования по поводу вероятности нахождения ответа в зависимости от количества запусков квантовой подпрограммы и самого квантового алгоритма.
- Функция `runInvestigation` — служебная функция для массированного запуска исследований зависимости вероятности нахождения результата по алгоритму Шора `investigate`.
- Функция `classicFactoring` — функция, которая написана специально для того, чтобы показать реализацию алгоритма Шора от первого до последнего шага без использования квантовой подпрограммы. Поиск периода осуществляется при помощи нахождения в списке степеней заданного числа первой единицы.

В целом, модуль `Shor` обеспечивает реализацию квантового алгоритма Шора для факторизации целых чисел, используя мощные свойства квантовой механики для выполнения вычислений намного быстрее, чем классические алгоритмы.

4.9. Модуль Simon

Модуль `Simon` реализует алгоритм Саймона, квантовый алгоритм, цель которого найти скрытый период в функции, представленной в виде чёрного ящика. Алгоритм основан на построении квантовой схемы, которая оценивает функцию в виде чёрного ящика на суперпозиции входов, а затем измеряет выходные кубиты для извлечения информации о скрытом периоде.

В модуле определены следующие функции:

- Функция `targetF` — функция, которая отображает пары булевых чисел в пары булевых чисел, представляющих целевую функцию, период которой необходимо найти.
- Функция `oracle` — матричное представление квантовой схемы, реализующей функцию-оракул для целевой функции.
- Функция `simon` — основная функция модуля, которая реализует алгоритм Саймона, используя функцию-оракул, предоставленную в качестве аргумента. Она возвращает результаты измерений всех кубитов, используемых в алгоритме.
- Функция `main` — главная функция модуля, которая строит гистограмму результатов измерения квантового регистра, запуская алгоритм Саймона заданное количество раз.

4.10. Модуль QuipperSimple

Модуль `QuipperSimple` предоставляет несколько функций для генерации квантовых схем на проблемно-ориентированном языке Quipper.

Модуль импортирует библиотеку `Quipper` и стандартный модель `Control.Monad`. Библиотека `Quipper` — это библиотека языка Haskell для квантового программирования, которая предоставляет набор абстракций и функций для создания квантовых схем.

Модуль `QuipperSimple` предоставляет следующие функции:

- Функция `andGate` — эта функция строит квантовую схему для аналога операции И (конъюнкция) на кубитах. Функция принимает на вход кортеж из двух кубитов, применяет гейн НЕ на третьем кубите, инициализированном |0⟩, и применяет гейт управляемое-НЕ (CNOT) на третьем кубите, контролируемом первыми двумя кубитами. Функция возвращает третий кубит.
- Функция `andList` — эта функция строит квантовую схему, которая выполняет конкатенацию (операцию И) на списке кубитов. Функция использует рекурсивный алгоритм для вычисления конъюнкции И из списка кубитов путём применения функции `andGate`. Если список пуст, функция инициализирует кубиты |1⟩. Если в списке есть только один кубит, функция возвращает его. В противном случае функция применяет функцию `andGate` рекурсивно к первому и второму кубитам списка и результат рекурсивного вызова к оставшимся кубитам списка.
- Функция `andList'` — эта функция является альтернативной реализацией функции `andList`, которая использует монадическую форму функции `fold` для применения функции `andGate` к списку кубитов.
- Функция `showScheme` — эта функция является сервисной функцией, которая отображает схему заданной функции с помощью функции Quipper `print_simple`.
- Функция `showSchemeGeneric` — эта функция является полиморфной версией функции `showScheme`, которая принимает дополнительный аргумент для указания формата вывода принципиальной схемы.
- Функция `main` — эта функция является функцией-заполнителем, которая не определена и предоставляется только для удовлетворения интерпретатора языка Quipper.

В целом, модуль `QuipperSimple` предоставляет несколько функций для генерации квантовых схем на языке Quipper. Эти функции могут быть использованы для реализации квантовых алгоритмов и симуляций на языке Quipper.

4.11. Модуль QuipperHard

Модуль `QuipperHard` предоставляет более сложные и продвинутые функции для генерации квантовых схем на проблемно-ориентированном языке программирования Quipper.

Модуль начинается с импорта модуля `Quipper`. Модуль `Quipper` предоставляет основные строительные блоки для генерации квантовых схем, а описываемый модуль `QuipperHard` предоставляет более сложные функции, построенные поверх модуля `Quipper`.

Модуль определяет несколько функций:

- Функция `plusMinus` принимает булевский аргумент и возвращает квантовую схему, которая преобразует кубиты из вычислительного базиса в базис {|+>, |->}. Она инициализирует кубит значением b и применяет к нему гейт Адамара.
- Функция `entangle` принимает кубит a и возвращает пару кубитов (a, b) в квантовом состоянии a|00> + b|11>. Она инициализирует новый кубит b со значением |0> и применяет к нему гейт контролируемого-НЕ, при этом кубит a является управляющим кубитом.
- Функция `bellPhiPlus` возвращает пару кубитов в состоянии Белла $|\Phi+\rangle = (|00\rangle + |11\rangle)/\sqrt{2}$, сначала применяя функцию `plusMinus` к кубиту, инициализированному значением |0>, а затем применяя функцию `entangle` к полученному кубиту.
- Функция `rotations` является вспомогательной функцией для функции `qft'`. Она принимает управляющий кубит c, список кубитов [q1, q2, ..., qn], целое число n и возвращает список кубитов [q1', q2', ..., qn'], в котором каждый кубит повернут на фазовый коэффициент, определяемый его положением в списке. Вращение контролируется параметром c.
- Функция `qft'` является основной функцией для выполнения квантового преобразования Фурье над списком кубитов. Она принимает список кубитов и возвращает список кубитов в базисе Фурье. Функция рекурсивно применяет к кубитам гейт Адамара, выполняет контролируемые вращения оставшихся кубитов и возвращает результирующий список кубитов.
- Функция `qft` является функцией-обёрткой для функции `qft'`, которая изменяет порядок входных кубитов перед применением квантового преобразования Фурье, а затем возвращает результирующие кубиты в исходном порядке.
- Главная функция `main` остается неопределённой и включена только для того, чтобы интерпретатор Quipper работал правильно.

В целом, модуль QuipperHard предоставляет полезные и сложные функции для генерации квантовых схем на языке Quipper. Эти функции могут быть использованы для построения более сложных квантовых алгоритмов и схем.

5. Примеры использования

В этом разделе будут приведены несколько примеров использования библиотеки QSimHs для решения конкретных задач квантовых вычислений. Эти примеры призваны продемонстрировать достаточную мощь и универсальность библиотеки, а также дать практическое руководство по созданию и запуску квантовых схем с помощью языка Haskell.

Описание начнётся с примера факторизации с использованием алгоритма Гровера, который является известным квантовым алгоритмом, обеспечивающим квадратичное ускорение по сравнению с классическими алгоритмами для определенных задач. Затем будет представлено определение гейта Тоффоли, который является фундаментальным строительным блоком в квантовых вычислениях и может быть использован для построения более сложных схем. Наконец, будут рассмотрены другие задачи, которые могут быть решены с помощью библиотеки QSimHs.

Каждый пример будет включать описание реализованных функций.

5.1. Модуль GF

Модуль `GF` демонстрирует пример использования библиотеки QSimHs для факторизации числа с помощью алгоритма Гровера. Алгоритм Гровера — это известный квантовый алгоритм, который обеспечивает квадратичное ускорение по сравнению с классическими алгоритмами для некоторых задач, включая факторизацию, которую можно реализовать через поиск. Используя библиотеку QSimHs, можно легко реализовать алгоритм Гровера и наблюдать квантовое превосходство в действии.

Модуль реализует следующие функции:

- Функция `pairsOfPrimes` — функция, возвращающая список пар простых чисел, из которых первое строго меньше второго.
- Функция `goodNumbers` — список чисел, которые могут быть разложены на простые множители при помощи алгоритма Шора (они являются произведением ровно двух различных простых чисел, отличающихся от 2).
- Функция `makeOracle` — функция, которая строит оракул для поиска заданного числа в списке правильных чисел `goodNumbers`. Первым аргументом принимает количество кубитов, для которого необходимо построить оракул (2 в степени этого количества должно быть больше, чем индекс числа для факторизации в списке хороших чисел). Число для факторизации должно подаваться вторым аргументом, и это должно быть хорошее число (присутствующее в списке), иначе функция впадёт в бесконечный поиск несуществующего числа в бесконечном списке.
- Функция `isSquare` — сервисная функция, которая возвращает значение `True`, если заданное целое число является квадратом.
- Функция `log2` — сервисная функция для вычисления логарифма по основанию 2, при этом возвращается только целое число (производится округление вверх), поскольку при помощи этой функции получается количество потребных кубитов.
- Функция `nofIterations` — сервисная функция для вычисления верхней оценки количества требуемых для
- Функция `binToNumber` — сервисная функция для преобразования строки, представляющей двоичное число, в число.
- Функция `diffusion` — функция, реализующая гейт диффузии.
- Функция `grover` — функция, реализующая квантовую схему алгоритма Гровера. Первым параметром получает оракул. Вторым параметром получает количество кубитов в схеме. Третьим параметром получает количество итераций.
- Функция `calculatePrimeFactors` — служебная функция для осуществления одного вызова алгоритма Гровера для факторизации заданного числа.

- Функция `calculateComplexity` — служебная функция для расчёта сложности выполнения алгоритма. Первым элементом возвращаемой пары является сложность классического алгоритма. Вторым, соответственно, является сложность квантового алгоритма.
- Функция `repeatUntil` — функция, которая осуществляет монадическое действие до тех пор, пока не будет выполнено условие (предикат). Возвращает результат действия, удовлетворяющий предикат, а заодно и количество повторений действия.
- Функция `showComplexity` — сервисная функция, которая выводит на экран заключение об эффективности квантового алгоритма по сравнению с классическим.
- Функция `main` — главная функция модуля, в которой осуществляется взаимодействие с пользователем, расчёт результата и вывод его на экран.

5.2. Модуль Toffoli

Этот модуль определяет функции для представления элементов Тоффоли и Фредкина, а также различные функции для представления всех интересных логических элементов, выраженных через элементы Тоффоли и Фредкина.

Модуль предоставляет реализацию для следующих функций:

- Функция `getX` — вспомогательная функция для получения первой компоненты результата, возвращаемого элементами Тоффоли и Фредкина.
- Функция `getY` — вспомогательная функция для получения второй компоненты результата, возвращаемого элементами Тоффоли и Фредкина.
- Функция `getZ` — вспомогательная функция для получения третьей компоненты результата, возвращаемого элементами Тоффоли и Фредкина.
- Функция `toffoli` — функция, реализующая элемент Тоффоли.
- Функция `toffoli'` — функция, реализующая элемент Тоффоли с помощью элемента Фредкина.
- Функция `fredkin` — функция, реализующая элемент Фредкина.
- Функция `fredkin'` — функция, реализующая элемент Фредкина с помощью элемента Тоффоли.
- Функции `not` и `not'` — функция, реализующая логическую операцию НЕ с помощью гейта Тоффоли и с помощью гейта Фредкина соответственно.
- Функции `and` и `and'` — функция, реализующая логическую операцию И с помощью гейта Тоффоли и с помощью гейта Фредкина соответственно.
- Функции `nand` и `nand'` — функция, реализующая логическую операцию НЕ-И с использованием гейта Тоффоли и с помощью гейта Фредкина соответственно.
- Функции `or` и `or'` — функция, реализующая логическую операцию ИЛИ с использованием гейта Тоффоли и с помощью гейта Фредкина соответственно.
- Функции `nor` и `nor'` — функция, реализующая логическую операцию НЕ-ИЛИ с использованием гейта Тоффоли и с помощью гейта Фредкина соответственно.
- Функции `xor` и `xor'` — функция, реализующая логическую операцию Исключающее ИЛИ с гейта Тоффоли и с помощью гейта Фредкина соответственно.
- Функции `fanout` и `fanout'` — функция, реализующая элемент FANOUT (дублирование значения бита) с помощью гейта Тоффоли и с помощью гейта Фредкина соответственно.

5.3. Модуль Tasks

Этот модуль обеспечивает реализацию некоторых задач квантовых вычислений. В частности, он предоставляет следующие функции:

- Функция `notHZH` реализует гейт НЕ, используя гейт Адамара и фазовый гейт **Z**.
- Функция `swapQubits` реализует квантовую схему, которая меняет местами состояния двух кубитов.
- Функция `deutsch` реализует алгоритм Дойча, который проверяет, является ли заданная двоичная функция постоянной или сбалансированной. Функция должна быть унарной и может быть одной из четырёх определённых функций. Эта функция уже определена через примитивы, заданные в библиотеке QSimHs, а не так, как в модуле `Deutsch`.
- Функция `gateCNOT'` — это реализация гейта Управляемое-НЕ, где управляющим кубитом является второй из двух кубитов.

В модуле описаны ещё несколько функций, являющихся служебными.

6. Устранение неполадок

Если пользователь столкнулся с какими-либо ошибками или проблемами при компиляции и запуске программ на языке Haskell или на языке Quipper, есть несколько шагов, которые можно предпринять для устранения неполадок.

Во-первых, если при компиляции кода компилятор сообщает, что отсутствуют какие-либо библиотеки, необходимо установить их с помощью утилиты cabal, входящей в состав Haskell Platform. Например, если при компиляции кода появляется сообщение об ошибке типа «Could not find module `Data.Map`», можно использовать следующую команду в терминале для установки недостающего пакета:

```
cabal install containers
```

Эта команда устанавливает пакет `containers`, который включает модуль `Data.Map`.

Если у пользователя нет интерпретатора проблемно-ориентированного языка Quipper, необходимо загрузить и установить его. Это можно сделать с помощью утилиты cabal или загрузив его с официального сайта языка. После установки интерпретатора Quipper можно запускать свой код на языке Quipper, вызывая интерпретатор в терминале:

quipper your-program.qp

Если у пользователя всё ещё возникают проблемы, можно обратиться за дальнейшей помощью к документации на язык Quipper или на форумы сообщества.

7. Заключение

В заключение, в этом документе было дано краткое описание библиотеки для симуляции квантовых вычислений QSimHs, а также её использование в квантовых вычислениях. Было представлено, как настроить библиотеку, а также было приведено несколько простых примеров для начала работы. Было дано несколько советов по устранению неполадок, которые могут возникнуть при установке или компиляции.

Главным дополнительным методическим пособием для работы с библиотекой QSimHs является следующая книга:

Душкин Р. В. *Квантовые вычисления и функциональное программирование*. — М.: ДМК-Пресс, 2014. — 318 с., ил.

Кроме того, на официальном YouTube-канале Р. В. Душкина «Душкин объяснит» имеются плейлисты, охватывающие основные темы, использованные в библиотеке QSimHs:

- Квантовые технологии: https://clck.ru/34HwBK
- Функциональное программирование: https://clck.ru/34HwBx
- Линейная алгебра: https://clck.ru/34HwAj

ООО «А-Я эксперт» надеется, что библиотека QSimHs станет для всех полезным инструментом в исследованиях и экспериментах в области квантовых вычислений. Спасибо, что выбрали это программное обеспечение, и мы будем рады любым вашим отзывам и предложениям, присылаемым на адрес электронной почты info@aia.expert.