Contents

Приветствие

Этот курс позволит вам погрузиться в удивительный мир квантового машинного обучения!

Почему именно этот курс?

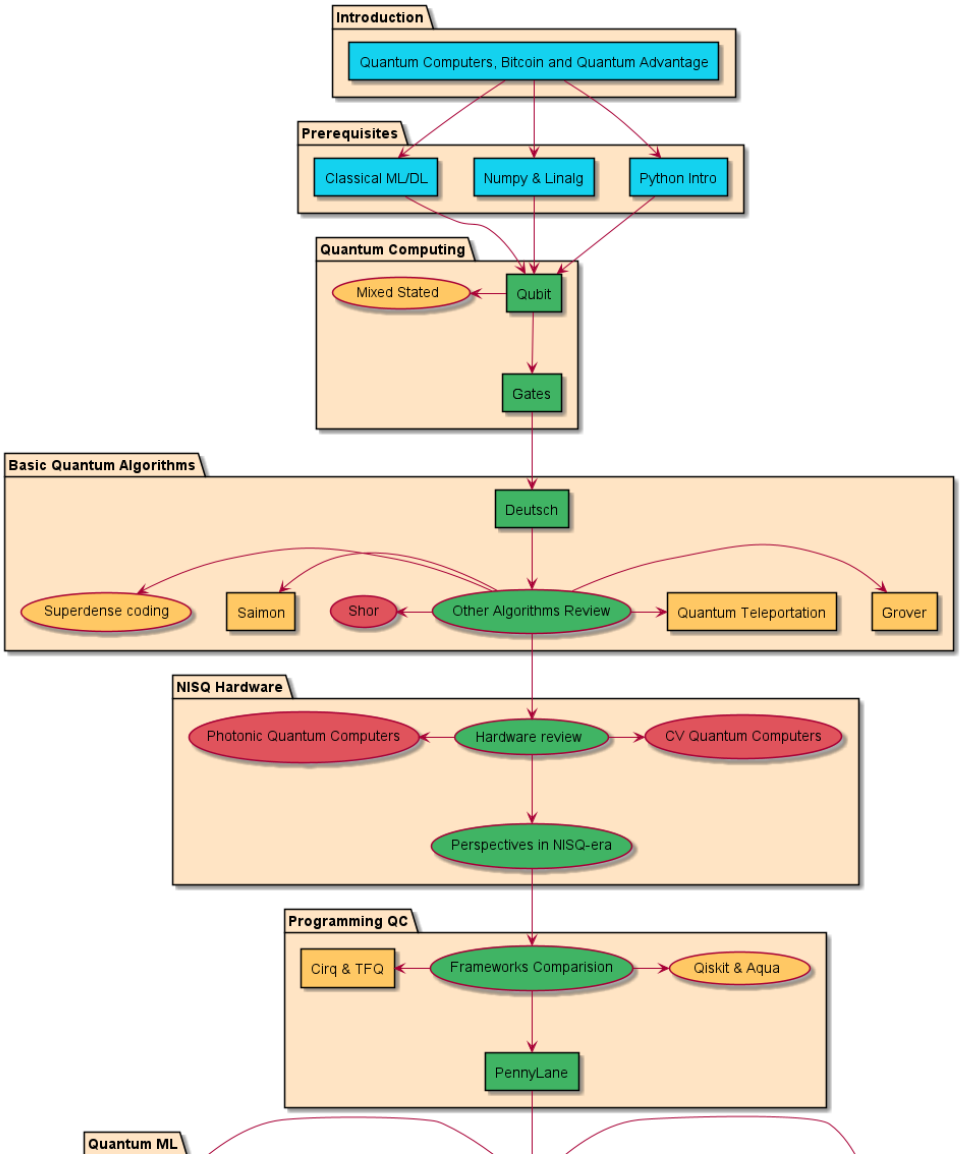
Наш курс отличается от других курсов по квантовым вычислениям:

- он адаптивный и содержит лекции разных уровней сложности и глубины;
- он практический, а все объяснения подкрепляются кодом;
- он про реальные методы, которые будут актуальны ближайшие 10-15 лет.

Как устроен курс?

Наш курс разделен на логические блоки, каждый из которых содержит лекции разных уровней сложности:

- **ГОЛУБОЙ** – вводные лекции;
- **ЗЕЛЕНЫЙ** – лекции “основного” блока курса;
- **ЖЕЛТЫЙ** – лекции, глубже раскрывающие темы блоков;
- **КРАСНЫЙ** – лекции про физику и математику, которая стоит за всем этим;
- **БЕЛЫЙ** – факультативные лекции.



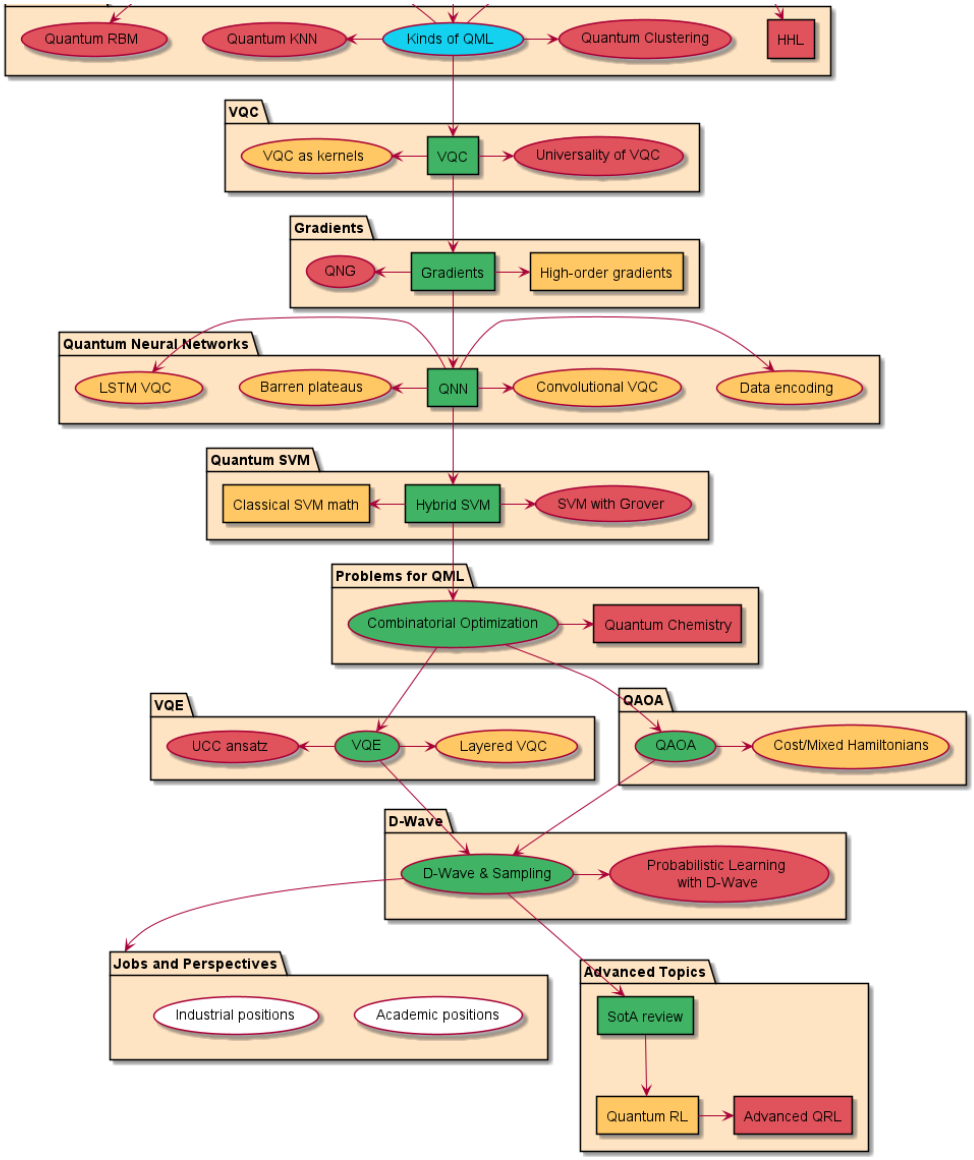


Fig. 1 Программа курса

Как будет проходить этот курс?

Рекомендуем проходить курс в порядке, обозначенном на схеме.

Желаем успехов!

О блоке

Этот блок включает в себя:

- общий рассказ о том, что такое квантовый бит;
- введение в основные квантовые гейты.

Продвинутые темы блока дополнительно рассказывают:

- о квантовой физике, на которой базируется концепция кубита;
- о смешанных состояниях, операторе плотности и энтропии фон Неймана;
- об алгоритмах Шора и Гровера.

Квантовый бит

Описание лекции

Эта лекция расскажет:

- что такое кубит;
- в чем разница между значением и состоянием;
- что такое сфера Блоха;
- какие можно делать операции над кубитами;
- что такое измерение.

Введение

Это первая лекция основного блока нашего курса. Прежде чем мы начнем детально разбирать понятие кубита, давайте взглянем на общий пайплайн квантовых схем.

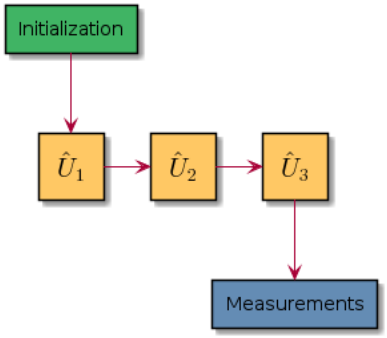


Fig. 2 Схема любого квантового алгоритма

Любая квантовая схема включает в себя:

- кубиты, инициализируемые в начальное состояние, обычно \(\ket{0}\);
- унитарные и обратимые операции над кубитами;
- измерение кубитов.

Эта лекция посвящена разбору операций для одного кубита. Начнем с понятия кубита и его отличий от бита классических компьютеров.

Что такое кубит

Классический компьютер оперирует двоичными числами – нулем и единицей. Минимальный объем информации для классического компьютера называется бит. Квантовый компьютер оперирует квантовыми битами или кубитами, которые тоже имеют два возможных значения – 0 и 1. Так в чем же разница? В чем особенности квантовых компьютеров, которые дают им преимущества над классическими компьютерами?

Разница в том, что для квантовомеханических систем (и кубитов в частности) их *состояния* и *значения* – это не одно и то же.

Состояние vs значение

Состояние классического бита

Обычно мы не отличаем состояние классического бита от его значения и считаем, что если бит имеет значение **1**, то и состояние его описывается числом **1**.

Кот Шредингера

Давайте вспомним мысленный эксперимента Шредингера. Кот, который одновременно и жив, и мертв. Понятно, что *значение* кота точно одно: он либо жив, либо мертв. Но *состояние* его более сложное. Он находится в *суперпозиции* состояний “жив” и “мертв” одновременно.

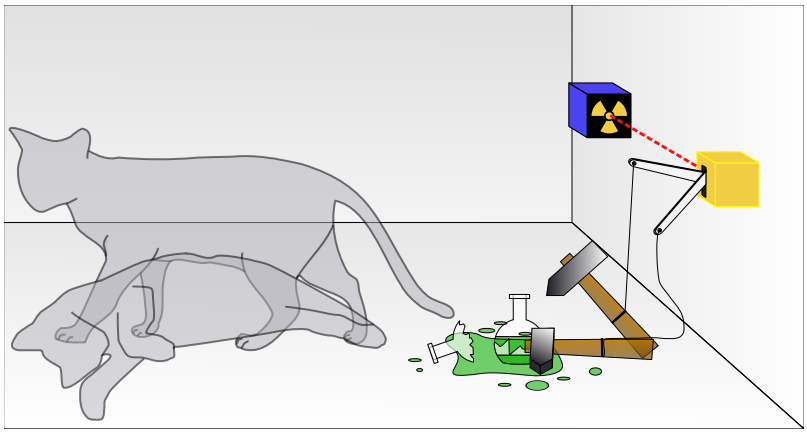


Fig. 3 Код Шредингера

Состояние кубита

Состояние кубита, если можно так сказать, аналогично состоянию кота Шредингера. Оно отличается от *значения* кубита и описывается вектором из двух комплексных чисел. Мы будем обозначать состояния (или вектора) символом \(\ket{\Psi}\) (кет – вектор-столбец) – это широко принятая в квантовой механике и квантовых вычислениях нотация Дирака:
\[\begin{split} \ket{\Psi} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \end{split}\]

Note

Может возникнуть вопрос, а почему комплексные числа? Короткий ответ на этот вопрос дать сложно. Если в двух словах, то использование комплексных чисел связано с удобством представления матричных групп, используемых в квантовой механике.

Все еще звучит сложно??? Тогда нужно вспомнить, что изначально квантовая механика возникла, в том числе из-за того, что физики экпериментально обнаружили у фундаментальных частиц свойство `корпускулярно-волнового дуализма`. Иными словами, электроны, фотоны и другие частицы проявляли как типичные свойства волнового движения (например интерференцию и дифракцию), и свойства частиц – например, всегда есть минимальная порция (`квант`!) света или электричества. Кстати, часто вместо вектора состояния используется понятие `волновой функции`, которая описывает плотность вероятности обнаружить частицу в той или иной точке пространства (обычного или специального). Ко времени создания квантоваой механики для описания волнового движения ученые уже привыкли использовать комплескные числа, которые позволяют упростить описание многих эффектов за счет разделения амплитуды и фазы процесса. Такое удобство справедливо и для многих задач квантовой физики.

Для более детального ответа авторы курса рекомендуют читать книги по истории квантовой физики (и по самой квантовой физике).

Значение чисел \(c_0\) и \(c_1\) мы обсудим чуть позже, а пока запишем наш кубит \(\ket{\Psi}\) в коде Python. Для начала \(c_0 = c_1 = \frac{1}{\sqrt{2}}\).

```
import numpy as np
qubit = np.array([1 / np.sqrt(2) + 0j, 1 / np.sqrt(2) + 0j]).reshape((2, 1))
```

Здесь мы создаем именно вектор-столбец размерности \(2\times1\).

```
print(qubit.shape)
```

```
(2, 1)
```

Связь состояния и значения кубита

Разберем подробнее вектор \(\ket{\Psi}\) и значение цифр \(c_0, c_1\). Посмотрим на состояния кубита, значение которого мы знаем точно. То есть “посмотрим на кота Шредингера”, но который точно жив или точно мертв.

Базисные состояния

Посмотрим, как выглядят состояния кубитов с точно определенными значениями:
\[\begin{split} \ket{0} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{\qquad} \ket{1} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \end{split}\]
Что мы можем сказать об этих состояниях? Как минимум следующее:

- они ортогональны (\(\ket{0} \perp \ket{1}\));
- они имеют единичную норму;
- они образуют базис.

Что это значит для нас? А то, что любое состояние \(\ket{\Psi}\) можно записать как линейную комбинацию векторов \(\ket{0}\) и \(\ket{1}\), причем коэффициентами в этой комбинации будут как раз наши \(c_0, c_1\):

```
basis_0 = np.array([1 + 0j, 0 + 0j]).reshape((2, 1))
basis_1 = np.array([0 + 0j, 1 + 0j]).reshape((2, 1))

c0 = c1 = 1 / np.sqrt(2)

print(np.allclose(qubit, c0 * basis_0 + c1 * basis_1))
```

```
True
```

Амплитуды вероятностей

Квантовая механика устроена таким интересным образом, что если мы будем измерять **значение** кубита, то вероятность каждого из вариантов будет пропорциональна соответствующему коэффициенту в разложении **состояния**. Но так как амплитуды – это в общем случае комплексные числа, а вероятности должны быть строго действительные, нужно домножить амплитуды на комплексно сопряженные значения. В случае наших значений \(c_0 = c_1 = \frac{1}{\sqrt{2}}\) получаем:

```
p0 = np.conj(c0) * c0
p1 = np.conj(c1) * c1

print(np.allclose(p0, p1))
print(np.allclose(p0 + p1, 1.0))
```

```
True
True
```

Видим еще одну важную вещь: сумма вероятностей всех состояний должна быть равна 100%. Это сразу приводит нас к тому, что состояния – это не любые комплексные вектора, а комплексные вектора с единичной нормой:

```
print(np.allclose(np.conj(qubit).T @ qubit, 1.0))
```

```
True
```

Мы будем очень часто пользоваться транспонированием и взятием комплексно сопряженного от векторов. В квантовой механике это имеет специальное обозначение \(\bra{\Psi} = \Psi^{T*} = \Psi^\dagger\) (бра – вектор-строка). Тогда наше правило нормировки из =NumPy= кода может быть записано в нотации Дирака так:
\[\braket{\Psi} = 1 \]

Сфера Блоха

Описанный выше базис \(\ket{0}, \ket{1}\) не является единственно возможным. Вектора \(\ket{0}, \ket{1}\) – это лишь самый часто применимый базис, который называют \(\mathbf{Z}\) базисом. Но есть и другие варианты.

Возможные базисы

Z-базис

Уже описанные нами \(\ket{0}\) и \(\ket{1}\).

X-базис

Базисные состояния \(\ket{+} = \frac{\ket{0} + \ket{1}}{\sqrt{2}}\) и \(\ket{-} = \frac{\ket{0} - \ket{1}}{\sqrt{2}}\):

```
plus = (basis_0 + basis_1) / np.sqrt(2)
minus = (basis_0 - basis_1) / np.sqrt(2)
```

Y-базис

Базисные состояния \(\ket{R} = \frac{\ket{0} + i\ket{1}}{\sqrt{2}}\) и \(\ket{L} = \frac{\ket{0} - i\ket{1}}{\sqrt{2}}\):

```
R = (basis_0 + 1j * basis_1) / np.sqrt(2)
L = (basis_0 - 1j * basis_1) / np.sqrt(2)
```

Легко убедиться, что все вектора каждого из этих базисов ортогональны:

```
print(np.allclose(np.conj(basis_0).T @ basis_1, 0))
print(np.allclose(np.conj(plus).T @ minus, 0))
print(np.allclose(np.conj(R).T @ L, 0))
```

```
True
True
True
```

Заметьте, что в наших векторных пространствах скалярное произведение – это \(\vec{a}\vec{b} = \left\langle a\middle| b\right\rangle\) (бра-кет). Именно поэтому нужно делать транспонирование и комплексное сопряжение первого вектора в паре.

Сфера Блоха

Обозначения \(\ket{0}, \ket{1}, \ket{+}, \ket{-}, \ket{R}, \ket{L}\) выбраны неслучайно: они имеют геометрический смысл.

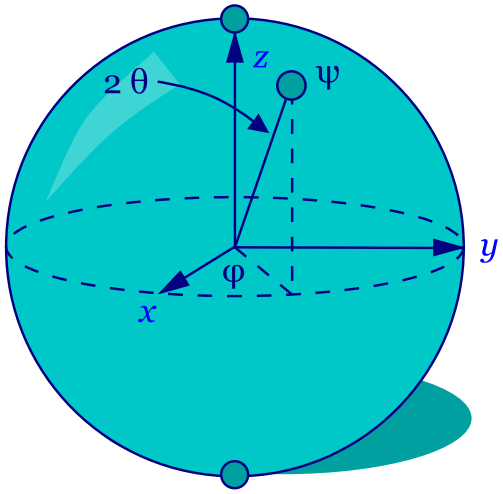


Fig. 4 Сфера Блоха

Принято считать, что ось \(\mathbf{Z}\) – это основная ось, так как физически квантовые компьютеры измеряют именно по ней. Ось \(\mathbf{X}\) “смотрит на нас” и поэтому обозначается \(\ket{+}\) и \(\ket{-}\). А ось \(\mathbf{Y}\) направлена как бы вдоль, поэтому базис обозначают как “право” (\(\ket{R}\)) и “лево” (\(\ket{L}\)).

Вектор состояния кубита еще называют волновой функцией и этот вектор может идти в любую точку сферы Блоха. Сама сфера имеет единичный радиус и это гарантирует нам, что для всех состояний сумма квадратов амплитуд будет равна единице.

Состояние в полярных координатах

Состояние кубита можно выразить через полярные координаты на сфере Блоха:
\[\ket{\Psi} = c_0 \ket{0} + c_1 \ket{1} = \cos\theta\ket{0} + e^{i\phi}\sin\theta \ket{1}, \]
где \(\theta,\phi\) – это угловые координаты на сфере Блоха. В этом смысле сфера Блоха очень удобна для представления состояний одного кубита.

> **Note**
>
> Тут мы воспользовались формулой Эйлера, а также вынесли за скобки локальные фазы множителей \(c_0\) и \(c_1\). Если у вас возникают трудности с подобными операциями над комплексными числами, то рекомендуем еще раз пересмотреть базовую лекцию по линейной алгебре и комплексным числам, там эти моменты освещаются более подробно.

Что можно делать с таким кубитом?

Линейные операторы

Любое действие, которое мы совершаем с кубитом в состоянии \(\ket{\Psi}\), должно переводить его в другое состояние \(\ket{\Phi}\). Что переводит один вектор в другой вектор в том же пространстве? Правильно, матрица. Другими словами, линейный оператор. Мы будем обозначать операторы как \(\hat{U}\).

Унитарность

Как мы уже говорили, квадраты амплитуд – это вероятности. Следовательно, волновая функция должна быть нормирована на единицу. А значит, любой оператор, который переводит одно состояние в другое \(\hat{U}\ket{\Psi} = \ket{\Phi}\), должен сохранять эту нормировку, то есть должен быть *унитарным*. Более того, свойство унитарности приводит к тому, что любой квантовый оператор еще и сохраняет скалярное произведение:
\[\bra{\Psi}\hat{U}^\dagger\hat{U}\ket{\Psi} = \bra{\Psi}\ket{\Psi} \]
Другими словами, унитарный оператор удовлетворяет условию \(\hat{U}^\dagger \hat{U} = \hat{I}\).

Обратимость

Одно из важных следствий унитарности операций над кубитами – это их обратимость. Если вы сделали какую-то последовательность унитарных операций над кубитами $\hat{U}$, то их можно вернуть в начальное состояние, ведь у унитарного оператора всегда есть обратный оператор $\hat{U}^{-1} = \hat{U}^\dagger$.

> **Note**
>
> Квантовый компьютер должен уметь делать несколько не унитарных операций, например, инициализацию кубита в определенное состояние (например, $\ket{0}$) и считывание состояния кубитов. Такие неунитарные операции приводят к потере информации и являются необратимыми.

Пример оператора

В дальнейших лекциях мы разберем много операторов, так как именно операторы (или квантовые **гейты**) являются основой квантовых вычислений. А пока посмотрим простой пример: оператор Адамара (**Hadamard gate**), который переводит $\ket{0} \to \ket{+}$.

Гейт Адамара

Начнем с того, что пока у нас лишь один кубит. Состояние одного кубита – это вектор размерности два. Значит, оператор, который переводит его в другой вектор размерности два – это матрица 2×2. Запишем оператор Адамара в матричном виде, а потом убедимся, что он унитарный и действительно переводит состояние $\ket{0} \to \ket{+}$.
$$\begin{split} \hat{H} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ \end{bmatrix} \end{split}$$
Реализация в Python

```
h = 1 / np.sqrt(2) * np.array([
    [1 + 0j, 1 + 0j],
    [1 + 0j, 0j - 1]
])
```

Унитарность

```
print(np.allclose(np.conj(h) @ h, np.eye(2)))
```

```
True
```

Правильное действие

```
print(np.allclose(h @ basis_0, plus))
```

```
True
```

Измерение

Измерение в квантовых вычислениях выделяется отдельно именно потому, что оно “открывает” коробку с котом Шредингера: мы точно узнаем, жив он или мертв, и уже никогда не сможем это “забыть” обратно. Вся *суперпозиция* его состояния исчезает. То есть *измерение* – это как раз пример одной из не унитарных операций, которые должен уметь делать квантовый компьютер.

> **Note**
>
> Это интересный факт: исчезновение суперпозиции многим кажется парадоксом, именно поэтому и появляются разные интерпретации квантовой механики, например, многомировая интерпретация Эверетта. Действительно, это кажется немного странным, что полностью обратимая квантовая механика и непрерывная динамика волновых функций вдруг “ломаются” и мы получаем такой коллапс, который еще называют редукцией фон Неймана. Доктору Эверетт тоже это не нравилось и он предложил другую интерпретацию этого процесса. Согласно его теории, когда мы производим измерения, мы как бы “расщепляем” нашу вселенную на две ниточки: в одной кот остается жив, а в другой остается мертв.
>
> Такие теории остаются на уровне спекуляций, так как почти невозможно придумать эксперимент, который бы подтверждал или опровергал такую гипотезу. Скорее это вопрос личного понимания и интерпретации процесса, так как математически подобные теории в итоге дают один и тот же наблюдаемый и измеримый результат.

Как мы уже говорили, у кубита может быть несколько разных базисов: $\ket{0}, \ket{1}$, $\ket{+}, \ket{-}$, $\ket{R}, \ket{L}$. *Значение* кубита в каждом из этих базисов может быть измерено. Но что такое измерение с точки зрения математики?

Операторы Паули

На самом деле, любая наблюдаемая величина соответствует какому-то оператору. Например, измерения в разных базисах $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$ соответствуют операторам Паули:
$$\begin{split} \hat{\sigma^x} = \begin{bmatrix} 0 & 1\\ 1 & 0 \end{bmatrix} \qquad \hat{\sigma^y} = \begin{bmatrix} 0 & -i\\ i & 0 \end{bmatrix} \qquad \hat{\sigma^z} = \begin{bmatrix} 1 & 0\\ 0 & -1 \end{bmatrix} \end{split}$$

```
pauli_x = np.array([[0 + 0j, 1 + 0j], [1 + 0j, 0 + 0j]])
pauli_y = np.array([[0 + 0j, 0 - 1j], [0 + 1j, 0 + 0j]])
pauli_z = np.array([[1 + 0j, 0 + 0j], [0 + 0j, 0j - 1]])
```

Эти операторы очень важны, рекомендуется знать их наизусть, так как они встречаются в каждой второй статье по квантовым вычислениям, а также постоянно фигурируют в документации всех основных библиотек для квантового машинного обучения.

Собственные значения

Мы поняли, что есть связь между нашими измерениями и операторами. Но какая именно? Что значит, например, что измерения по оси $\mathbf{Z}$ соответствуют оператору $\hat{\sigma^Z}$?

Здесь мы приходим к собственным значениям операторов. Оказывается (так устроен наш мир), что *измеряя* какую-то величину в квантовой механике, мы всегда будем получать одно из собственных значений соответствующего оператора, а состояние будет коллапсировать в соответствующий собственный вектор этого оператора. Другими словами, *измеряя* кота Шредингера, мы будем получать значения “жив” или “мертв”, а состояние кота будет переходить в состояние, соответствующее одному из этих значений. А еще *измерение* не является обратимой операцией: однажды открыв коробку с котом и поняв, жив он или мертв, мы уже не сможем закрыть ее обратно и вернуть кота в суперпозицию.

Описанное выше – не абстрактные рассуждения из квантовой физики. Оно пригодится, когда мы будем говорить о решении практических комбинаторных задач, таких как задача о выделении сообществ в графе.

Собственные вектора $\hat{\sigma^Z}$

Вернемся к нашему оператору $\hat{\sigma^Z}$. Легко убедиться, что его собственные значения равны 1 и -1, а соответствующие им собственные вектора – это $\begin{bmatrix}1 \\ 0\end{bmatrix}$ и $\begin{bmatrix}0 \\ 1\end{bmatrix}$:

```
print(np.linalg.eig(pauli_z))
```

```
(array([ 1.+0.j, -1.+0.j]), array([[1.+0.j, 0.+0.j],
       [0.+0.j, 1.+0.j]]))
```

Таким образом, измерение по оси $\mathbf{Z}$ всегда будет давать нам одно из этих двух значений и переводить состояние кубита в соответствующий собственный вектор.

Формальная запись

Формально мы можем записать для любого эрмитова оператора $\hat{U}$, что собственные состояния этого оператора являются его собственными векторами, а собственные значения в этом случае являются наблюдаемыми значениями:
$$ \hat{U}\ket{\Psi} = u\ket{\Psi} $$

Другие операторы Паули

Убедимся, что у остальных операторов собственные значения такие же:

```
print(np.linalg.eig(pauli_x))
print(np.linalg.eig(pauli_y))
```

```
(array([ 1.+0.j, -1.+0.j]), array([[ 0.70710678-0.j,  0.70710678+0.j],
       [ 0.70710678+0.j, -0.70710678-0.j]]))
(array([ 1.+0.j, -1.+0.j]), array([[-0.        -0.70710678j,  0.70710678+0.j
],
       [ 0.70710678+0.j        ,  0.        -0.70710678j]]))
```

Note

Можно заметить, что у всех операторов Паули нет ни одного общего собственного вектора. Таким образом, мы приходим к ситуации, когда не можем одновременно точно провести измерения двумя разными операторами, так как наше измерение должно переводить состояние в соответствующий собственный вектор. В квантовой механике это называется **принципом неопределенности**.

Ожидаемое значение при измерении

Мы не будем писать с нуля полный симулятор кубитов, который включает измерения – это требует введения сложного случайного процесса. Но мы можем легко ответить на другой вопрос. А именно: можно ли сказать, какое будет *ожидаемое* значение оператора $\hat{U}$ для состояния Ψ? Другими словами, какое будет математическое ожидание большого числа измерений? Это можно записать следующим образом:
$$ \mathbf{E}(\hat{U}) = \bra{\Psi}\hat{U}\ket{\Psi} $$
Например, оператор $\hat{\sigma^z}$ полностью не определен в состоянии $\ket{+}$, то есть мы будем равновероятно получать значения -1 и 1, а математическое ожидание, соответственно, будет равно нулю:

```
print(plus.conj().T @ pauli_z @ plus)
```

```
[[-2.23711432e-17+0.j]]
```

С другой стороны, измеряя состояние $\ket{+}$ в *X*-базисе мы всегда будем получать 1:

```
print(plus.conj().T @ pauli_x @ plus)
```

```
[[1.+0.j]]
```

Вероятности битовых строк

Последнее, чего мы коснемся в части измерений – это битовые строки и метод Шредингера. Мы много говорили о вероятностной интерпретации волновой функции и аналогиях с классическим битом, но пока этого никак не касались на практике. Как же получить вероятность определенной битовой строки для произвольного состояния? Если взять все битовые строки размерности вектора состояния и отсортировать их в лексикографическом порядке (например, $0 < 1$), ($00 < 01 < 10 < 11$), и т.д.), то вероятность каждой битовой строки получается следующим выражением:
$$ \mathbf{P} = | \bra{\Psi}\ket{\vec{s}} |^2, $$
где $\vec{s}$ – это вектор, каждая компонента которого соответствует порядковой битовой строке или вектор битовых строк. Другими словами, вероятность получить *i*-ю битовую строку равна квадрату *i*-го элемента амплитуды волновой функции. Кажется немного запутанным, но на самом деле $|\ket{\Psi}|^2$ – это идейно и есть плотность вероятности.

Еще пара слов об измерениях

Измерение как проекция на пространство собственных векторов

Мы уже говорили, что при измерении мы как бы “выбираем” один из собственных векторов наблюдаемой. Более строго такой процесс называется проецированием на пространство собственных векторов. Для собственного вектора $\ket{\Phi}$ проекция будет линейным оператором:
$$ \hat{P}\_{\ket{\Phi}} = \ket{\Phi}\bra{\Phi} $$

```
super_position = h @ basis_0
eigenvectors = np.linalg.eig(pauli_z)[1]

proj_0 = eigenvectors[0].reshape((-1, 1)) @ eigenvectors[0].reshape((1, -1))
proj_1 = eigenvectors[1].reshape((-1, 1)) @ eigenvectors[1].reshape((1, -1))
```

Правило Борна

Вероятность наблюдения каждого из собственных значений \(\lambda\) какого-то оператора \(\hat{U}\) определяется как результат измерения оператора проекции на соответствующий собственный вектор:
\[\mathbf{P}(\lambda_i) = \bra{\Psi} \hat{P_i} \ket{\Psi} \]
Считать ожидаемое значение оператора мы уже умеем. Давайте убедимся, что для состояния \(\ket{\Psi} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1\end{bmatrix}\) результаты измерений операторов проекций дадут 0.5 и совпадут с результатом упражнения, которое мы проделали ранее:

```
p_0 = super_position.conj().T @ proj_0 @ super_position
p_1 = super_position.conj().T @ proj_1 @ super_position

print(np.allclose(p_0 + p_1, 1.0))
print(np.allclose(p_0, 0.5))
```

```
True
True
```

Что мы узнали?

- Состояние и значение для кубита – это не одно и то же.
- Состояния представляют собой комплекснозначные вектора.
- Квантовые операторы – унитарные и самосопряженные.
- Измеряемые значения – собственные значения операторов.
- Измерение “ломает” суперпозицию.

Квантовые гейты

Описание лекции

Из этой лекции мы узнаем:

- какие есть основные однокубитные и многокубитные гейты;
- как записывать многокубитные состояния;
- как конструировать многокубитные операторы;
- как работать с библиотекой `PennyLane`.

Введение

Квантовые гейты являются основными *строительными* блоками для любых квантовых схем, в том числе и тех, что применяются для машинного обучения. Можно сказать, что это своеобразный алфавит квантовых вычислений. Он необходим, чтобы сходу понимать, например, что изображено на подобных схемах:

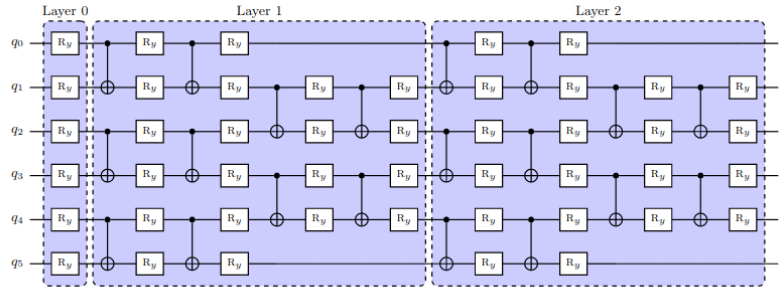


FIG. 2: L-VQE ansatz for a 6-qubit quantum state. R_y denotes rotation around the y-axis defined as $R_y(\theta) \equiv e^{-i\frac{\theta}{2}Y}$. Every R_y contains a parameter that is optimized over in the outer loop.

Fig. 5 Схема Layered-VQE

Основные однокубитные гейты

В прошлый раз мы познакомились с операторами Паули, а также гейтом Адамара. Как для обычных квантовых алгоритмов, так и для QML-алгоритмов нужны и другие гейты, потому что одни только эти гейты не позволяют перейти во все возможные квантовые состояния. Теперь давайте посмотрим, какие еще однокубитные гейты часто применяются в квантовых вычислениях и квантовом машинном обучении.

T-гейт

T-гейт очень популярен в универсальных квантовых вычислениях. Его матрица имеет вид:
$$\begin{split} \hat{T} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{bmatrix} \end{split}$$
Любой однокубитный гейт можно аппроксимировать последовательностью гейтов Адамара и T-гейтов. Чем точнее требуется аппроксимация, тем длиннее будет аппроксимирующая последовательность.

Помимо важной роли в математике квантовых вычислений, гейт Адамара и T-гейт интересны тем, что именно на них построено большинство предложений по реализации квантовых вычислений с топологической защитой или с коррекцией ошибок. На сегодняшний день эти схемы реально пока не очень работают: никаких топологически защищенных кубитов продемонстрировано не было, а коррекция ошибок не выходит за пределы двух логических кубитов.

Гейты поворота вокруг оси

Поворотные гейты играют центральную роль в квантовом машинном обучении. Вспомним на секунду, как выглядят наши однокубитные состояния на сфере Блоха:

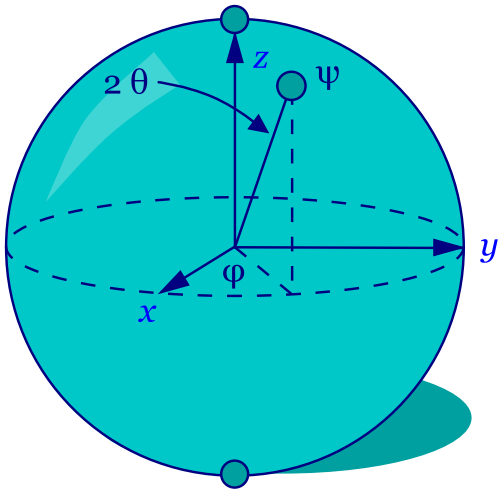


Fig. 6 Сфера Блоха

Любой однокубитный гейт можно представить как вращение вектора состояния Ψ на некоторый угол вокруг некоторой оси, проходящей через центр сферы Блоха.

Гейты $\hat{RX}(\phi), \hat{RY}(\phi), \hat{RZ}(\phi)$ осуществляют поворот на определенный угол ϕ вокруг соответствующей оси на сфере Блоха.

Давайте внимательно рассмотрим это на примере гейта $\hat{RY}$.

Гейт $\hat{RY}$

Сам гейт определяется следующим образом:
$$\begin{split} \hat{RY}(\phi) = \begin{bmatrix} \cos(\frac{\phi}{2}) & -\sin(\frac{\phi}{2}) \\ \sin(\frac{\phi}{2}) & \cos(\frac{\phi}{2}) \end{bmatrix} \end{split}$$

```
import numpy as np

def ry(state, phi):
    return np.array([
        [np.cos(phi / 2), -np.sin(phi / 2)],
        [np.sin(phi / 2),  np.cos(phi / 2)]
    ]) @ state
```

Запишем наше состояние $\ket{0}$:

```
basis = np.array([1 + 0j, 0 + 0j]).reshape((2, 1))
```

Внимательно посмотрим на сферу Блоха. Можно заметить, что если повернуть состояние из \(\ket{0}\) на \(\pi\) и измерить значение \(\hat{\sigma^z}\), то получится 1. А если повернуть на \(-\pi\), то получится 0:

```
def expval(state, op):
    return state.conj().T @ op @ state

pauli_x = np.array([[0 + 0j, 1 + 0j], [1 + 0j, 0 + 0j]])

print(np.allclose(expval(ry(basis, np.pi / 2), pauli_x), 1.0))
print(np.allclose(expval(ry(basis, -np.pi / 2), pauli_x), -1.0))
```

```
True
True
```

Убедимся также, что вращение на угол, пропорциональный \(2\pi\), не меняет результат измерения. Возьмем случайное состояние:
\[\begin{split} \ket{\Psi} = \begin{bmatrix} 0.42 \\ \sqrt{1 - 0.42^2} \end{bmatrix} \end{split}\]

```
random_state = np.array([0.42 + 0j, np.sqrt(1 - 0.42**2) + 0j]).reshape((2, 1))
```

Измерим его по осям \(\mathbf{X}\) и \(\mathbf{Z}\), затем повернем на угол \(2\pi\) и измерим снова:

```
pauli_z = np.array([[1 + 0j, 0 + 0j], [0 + 0j, 0j - 1]])

print("Z:\n\t" + str(expval(random_state, pauli_z)) + "\n")
print("X:\n\t" + str(expval(random_state, pauli_x)) + "\n")

print("Z after RY:\n\t" + str(expval(ry(random_state, 2 * np.pi), pauli_z)) + "\n")
print("X after RY:\n\t" + str(expval(ry(random_state, 2 * np.pi), pauli_x)) + "\n")
```

```
Z:
        [[-0.6472+0.j]]

X:
        [[0.76232025+0.j]]

Z after RY:
        [[-0.6472+0.j]]

X after RY:
        [[0.76232025+0.j]]
```

Другие гейты вращений

Аналогичным образом определяются гейты \(\hat{RX}\) и \(\hat{RZ}\):
\[\begin{split} \hat{RX}(\phi) = \begin{bmatrix} \cos(\frac{\phi}{2}) & -i\sin(\frac{\phi}{2}) \\ -i\sin(\frac{\phi}{2}) & \cos(\frac{\phi}{2}) \end{bmatrix} \qquad \hat{RZ}(\phi) = \begin{bmatrix} e^{-\frac{i\phi}{2}} & 0 \\ 0 & e^{\frac{i\phi}{2}} \end{bmatrix} \end{split}\]

Общая форма записи однокубитных гейтов

В общем случае однокубитные гейты могут быть также записаны следующим образом:
\[\large \hat{R}^\vec{n}(\alpha) = e^{-\frac{i\alpha\hat{\vec{\sigma}}\vec{n}}{2}}, \]
где \(\alpha\) – это угол поворота, \(\vec{n}\) – единичный вектор в направлении оси поворота, а \(\hat{\vec{\sigma}} = \{\hat{\sigma}^x, \hat{\sigma}^y, \hat{\sigma}^z\}\) – это вектор, составленный из операторов Паули. Если использовать покоординатную запись и \(\vec{n} = \{n_x, n_y, n_z\}\) задает ось вращения, то
\[\large \hat{R}^\vec{n}(\alpha) = e^{-i\frac{\alpha}{2}\left(\hat{\sigma}^xn_x+\hat{\sigma}^yn_y+\hat{\sigma}^zn_z\right)}. \]
Забегая вперед, можно сказать, что именно гейты вращений – это основа квантовых вариационных схем, главного инструмента этого курса.

Phase-shift гейт

Другой важный гейт – это так называемый phase-shift гейт или \(\hat{U}_1\) гейт. Его матричная форма имеет следующий вид:
\[\begin{split} \hat{U}_1(\phi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix} \end{split}\]

```
def u1(state, phi):
    return np.array([[1, 0], [0, np.exp(1j * phi)]]) @ state
```

Легко видеть, что с точностью до глобального фазового множителя, который ни на что не влияет, Phase-shift-гейт – это тот же $\hat{RZ}(\phi)$. Он играет важную роль в квантовых ядерных методах.

Гейты $\hat{U}_2$ и $\hat{U}_3$

Более редкие в QML гейты, которые однако все равно встречаются в статьях.
$$\begin{split} \hat{U}_2(\phi, \lambda) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi + \lambda)} \end{bmatrix} = \hat{U}_1(\phi + \lambda)\hat{RZ}(-\lambda)\hat{RY}(\frac{\pi}{2})\hat{RZ}(\lambda) \end{split}$$
Давайте убедимся в справедливости этого выражения:

```
def rz(state, phi):
    return np.array([[np.exp(-1j * phi / 2), 0], [0, np.exp(1j * phi / 2)]]) @ state


def u2_direct(phi, l):
    return (
        1
        / np.sqrt(2)
        * np.array([[1, -np.exp(1j * l)], [np.exp(1j * phi), np.exp(1j * (phi + l))]])
    )


def u2_inferenced(phi, l):
    return (
        u1(np.eye(2), phi + l)
        @ rz(np.eye(2), -l)
        @ ry(np.eye(2), np.pi / 2)
        @ rz(np.eye(2), l)
    )

print(np.allclose(u2_direct(np.pi / 6, np.pi / 3), u2_inferenced(np.pi / 6, np.pi /
3)))
```

```
True
```

Схожим образом определяется $\hat{U}_3(\theta, \phi, \lambda)$:
$$\begin{split} \hat{U}_3(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -e^{1j\lambda}\sin(\frac{\theta}{2}) \\ e^{1j\phi}\sin(\frac{\theta}{2}) & e^{1j(\phi + \lambda)}\cos(\frac{\theta}{2}) \end{bmatrix} = \hat{U}_1(\phi + \lambda)\hat{RZ}(-\lambda)\hat{RY}(\theta)\hat{RZ}(\lambda) \end{split}$$
Читатель может сам легко убедиться, что эти формы записи эквивалентны. Для этого надо написать примерно такой же код, что мы писали раньше для $\hat{U}_2$.

Еще пара слов об однокубитных гейтах

На этом мы завершаем обзор основных однокубитных гейтов. Маленькое замечание: гейты, связанные со сдвигом фазы, никак не меняют состояние кубита, если оно сейчас $\ket{0}$. Так как мы всегда предполагаем, что начальное состояние кубитов – это именно $\ket{0}$, то перед применением, например, $\hat{U}_1$, рекомендуется применить гейт Адамара:

```
print(np.allclose(u1(basis, np.pi / 6), basis))

h = 1 / np.sqrt(2) * np.array([[1 + 0j, 1 + 0j], [1 + 0j, 0j - 1]])
print(np.allclose(u1(h @ basis, np.pi / 6), h @ basis))
```

```
True
False
```

Единичный гейт

Самое последнее об однокубитных гейтах – это единичный гейт $\hat{I}$:
$$\begin{split} \hat{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{split}$$

```
i = np.eye(2, dtype=np.complex128)
print(i)
```

```
[[1.+0.j 0.+0.j]
 [0.+0.j 1.+0.j]]
```

Он не делает с кубитом ровным счетом ничего. Но единичный гейт понадобится нам позже, когда мы будем конструировать многокубитные операторы.

Многокубитные состояния и гейты

Очевидно, что с одним кубитом ничего интересного, кроме разве что генератора истинно-случайных чисел, мы не сделаем. Для начала разберемся, как выглядят состояния для многокубитных систем.

Многокубитные состояния

В классическом компьютере один бит имеет два значения – 0 и 1. Два бита имеют четыре значения – 00, 01, 10, 11. Три бита имеют восемь значений и так далее. Аналогично состояние двух кубитов – это вектор в пространстве $\mathbf{C}^4$, состояние трех кубитов – вектор в пространстве $\mathbf{C}^8$, то есть состояние N кубитов описывается вектором размерности 2^N в комплексном пространстве. Вероятности каждой из возможных битовых строк ($0000...00$, $0000...01$, $0000...10$, и так далее) получаются по методу Шредингера, который мы обсуждали в конце прошлой лекции:
$$ \mathbf{P}(\vec{s}) = | \bra{\Psi}\ket{\vec{s}} |^2 $$
Нужно отсортировать наши битовые строки в лексикографическом порядке – и вероятность *i*-й битовой строки будет равна квадрату *i*-го элемента вектора $\ket{\Psi}$.

Формально многокубитные состояния описываются с помощью математического концепта так называемого тензорного произведения, иначе – произведения Кронекера, обозначаемого значком $\otimes$. Так, если $\ket{\Psi}_A \in \mathrm{H}_A$ и $\ket{\Psi}_B \in \mathrm{H}_B$, то $\ket{\Psi}_{AB} = \ket{\Psi}_A \otimes \ket{\Psi}_B \in \mathrm{H}_{AB} = \mathrm{H}_{A} \otimes \mathrm{H}_{B}$. О том, как элементы вектора $\ket{\Psi}_{AB}$ выражаются через элементы векторов $\ket{\Psi}_{A}$ и $\ket{\Psi}_{B}$, можно прочитать на Википедии в статье "Произведение Кронекера".

Многокубитные операторы

Как мы уже обсуждали ранее, квантовые операторы должны переводить текущее состояние в новое в том же пространстве и сохранять нормировку, а еще должны быть обратимыми. Значит, оператор для состояния из N кубитов – это унитарная комплексная матрица размерности $2^N \times 2^N$.

Конструирование многокубитных операторов

Прежде чем мы начнем обсуждать двухкубитные операторы, рассмотрим ситуацию. Представим, что у нас есть состояние из двух кубитов и мы хотим подействовать на первый кубит оператором Адамара. Как же тогда нам написать такой двухкубитный оператор? Мы знаем, что действуем на первый кубит оператором, а что происходит со вторым кубитом? Ничего не происходит – и это эквивалентно тому, что мы действуем на второй кубит единичным оператором. А финальный оператор $2^2 \times 2^2$ записывается через произведение Кронекера:
$$\begin{split} \hat{H} \otimes \hat{I} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \hat{I} & \hat{I} \\ \hat{I} & -\hat{I} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \end{split}$$
Учитывая, что многокубитные состояния конструируются аналогичным образом через произведение Кронекера, мы можем убедиться в верности нашего вывода:

```
print(np.allclose(np.kron(h @ basis, basis), np.kron(h, i) @ np.kron(basis, basis)))
```

```
True
```

Наблюдаемые для многокубитных гейтов

Аналогичным образом можно сконструировать и наблюдаемые. Например, если мы хотим измерять одновременно два спина по оси $\mathbf{Z}$, то наблюдаемая будет выглядеть так:
$$\begin{split} \mathbf{ZZ} = \hat{\sigma^z} \otimes \hat{\sigma^z} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{split}$$

```
print(np.kron(basis, basis).conj().T @ np.kron(pauli_z, pauli_z) @ np.kron(basis,
basis))
```

```
[[1.+0.j]]
```

Основные двухкубитные гейты

Основные многокубитные гейты, которые предоставляют современные квантовые компьютеры, – это двухкубитные гейты.

CNOT (CX)

Квантовый гейт контролируемого инвертирования – это гейт, который действует на два кубита: *рабочий* и *контрольный*. В зависимости от того, имеет ли контрольный кубит значение 1 или 0, этот гейт инвертирует или не инвертирует рабочий кубит.

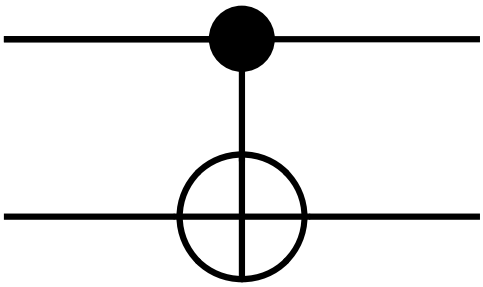


Fig. 7 Гейт CNOT

Иногда этот гейт также называют гейтом CX. В матричном виде этот оператор можно записать так:
\[\begin{split} \hat{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 ,& 0 \end{bmatrix} \end{split}\]

```
cnot = (1 + 0j) * np.array(
    [
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 0, 1],
        [0, 0, 1, 0],
    ]
)

print(np.allclose(cnot @ np.kron(basis, basis), np.kron(basis, basis)))
print(np.allclose(
    cnot @ np.kron(pauli_x @ basis, basis), np.kron(pauli_x @ basis, pauli_x @ basis)
))
```

```
True
True
```

Заметьте, тут мы воспользовались тем, что \(\hat{\sigma^x}\) работает так же, как инвертор кубитов: он превращает \(\ket{0}\) в \(\ket{1}\) и наоборот.

Гейты CY и CZ

Схожие по принципу гейты – это гейты \(\hat{CY}\) и \(\hat{CZ}\). В зависимости от значения управляющего кубита к рабочему кубиту применяют соответствующий оператор Паули:
\[\begin{split} \hat{CY} = \begin{bmatrix} 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & -i\\ 0 & 0 & i & 0 \end{bmatrix} \qquad \hat{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & -1 \end{bmatrix} \end{split}\]

Гейт iSWAP

Гейты \(\hat{CX}\), \(\hat{CY}\) и \(\hat{CZ}\) эквивалентны с точностью до однокубитных гейтов. Это означает, что любой из них можно получить, добавив необходимые однокубитные гейты до и после другого гейта. Например:
\[\hat{CZ} = \left(\hat{I}\times\hat{H}\right)\hat{CX}\left(\hat{I}\times\hat{H}\right). \]

Этим свойством обладают отнюдь не все двухкубитные гейты. Например, таковым является гейт iSWAP:
\[\begin{split} \mathrm{iSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0\\ 0 & 0 & -i & 0\\ 0 & -i & 0 & 0\\ 0 & 0 & 0 & 1 \end{bmatrix} \end{split}\]

Гейт fSim

Для разных архитектур квантовых процессоров “естественный” гейт может выглядеть по-разному. Например, в квантовом процессоре Google Sycamore естественным является так называемый fermionic simulation gate или fSim. Это двухпараметрическое семейство гейтов вида:
\[\begin{split} \mathrm{fSim}(\theta, \phi) = \begin{bmatrix} 1 & 0 & 0 & 0\\ 0 & \cos\theta & -i\sin\theta & 0\\ 0 & -i\sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & e^{-i\phi} \end{bmatrix}. \end{split}\]
Впрочем, и fSim-гейт не является эквивалентным всему множеству двухкубитных гейтов. В общем случае, чем больше кубитов, тем сложнее будет выглядеть декомпозиция произвольного гейта на физически реализуемые в “железе”.

Первое знакомство с PennyLane

На сегодняшний день существует достаточно много фреймворков для программирования квантовых компьютеров. Для целей этого курса мы будем использовать PennyLane. Он предоставляет высокоуровневый `Python API` и создан специально для решения задач квантового машинного обучения.

```
import pennylane as qml
```

Device

Для объявления квантового устройства используется класс `Device`. `PennyLane` поддерживает работу с большинством существующих квантовых компьютеров, но для целей курса мы будем запускать все наши программы лишь на самом простом симуляторе идеального квантового компьютера:

```
device = qml.device("default.qubit", 2)
```

Первый аргумент тут – указание устройства, а второй – число кубитов.

QNode

Основной *строительный блок* в `PennyLane` – это `qnode`. Это функция, которая отмечена специальным декоратором и включает в себя несколько операций с кубитами. Результатом такой функции всегда является измерение. Напишем функцию, которая поворачивает первый кубит на \(45^o\), после чего измеряет оба кубита по оси \(\mathbf{Z}\).

Сначала на NumPy

```
state = np.kron(basis, basis)
op = np.kron(ry(np.eye(2), np.deg2rad(45)), np.eye(2, dtype=np.complex128))
measure = np.kron(pauli_z, pauli_z)

print((op @ state).conj().T @ measure @ (op @ state))
```

```
[[0.70710678+0.j]]
```

Теперь через QNode

```
@qml.qnode(device)
def test(angle):
    qml.RY(angle, wires=0)
    return qml.expval(qml.PauliZ(0) @ qml.PauliZ(1))


print(test(np.deg2rad(45)))
```

```
0.7071067811865475
```

Заключение

Это последняя вводная лекция, где мы сами писали операторы и операции на чистом `NumPy`: это должно помочь лучше понять ту математику, которая лежит “под капотом” у квантовых вычислений. Дальше мы будем пользоваться только `PennyLane` и в отдельной лекции расскажем, как работать с этим фреймворком.

Итого:

- мы знаем, что такое кубит;
- понимаем линейную алгебру, которая описывает квантовые вычисления;
- понимаем, как можно сконструировать нужный нам оператор и как его применить;
- знаем, что такое измерение и наблюдаемые.

Теперь мы готовы к тому, чтобы знакомиться с квантовыми вариационными схемами и переходить непосредственно к построению моделей квантового машинного обучения.

Задачи

- Как связаны ось и угол вращения на сфере Блоха с собственными значениями и собственными векторами матрицы однокубитного гейта? Для этого найдите собственные векторы и собственные значения гейта \(R^\vec{n}\left(\alpha\right)\).
- Вокруг какой оси и на какой угол вращает состояние гейт Адамара?
- Гейт SWAP меняет кубиты местами. Его унитарная матрица имеет вид:

\[\begin{split} \mathrm{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \end{split}\]
Попробуйте составить последовательность гейтов, реализующую \(\mathrm{SWAP}\), из гейтов \(\mathrm{iSWAP}\), \(\hat{CZ}\) и \(\hat{RZ}(\phi)\).

Алгоритм Дойча

Алгоритм Дойча (в английском варианте - **Deutsch’s algorithm**) - это один из первых алгоритмов, показавших, что квантовый компьютер может решать задачи особым способом, отличающимся как от алгоритмов классического компьютера, так и от интуиции и здравого смысла человека. При этом такое решение может занимать меньшее количество шагов.

Нужно прежде всего сказать, что алгоритм Дойча не имеет практического применения в силу своей предельной простоты, зато является простейшим примером, с помощью которого можно понять, в чем состоит отличие квантовых алгоритмов от классических. Данный алгоритм был предложен в 1985 году, когда квантовых компьютеров еще не было, а практически он был реализован в 1998 году на 2-кубитном квантовом компьютере, работавшем на принципах ядерно-магнитного резонанса.

> **Дэвид Дойч**
>
> Помимо занятий теоретической физикой в Оксфордском университете, Дэвид Дойч - автор книг “Структура реальности” и “Начало бесконечности”, в которых он популярно излагает идеи квантовых вычислений с точки зрения многомировой интерпретеции (сторонником которой является) и философствует о будущем науки и человечества. Так что можно сказать, что работа алгоритма, согласно замыслу создателя, производится в параллельных вселенных. Так это или нет, пока проверить невозможно, но вычисления работают, и это главное.

Итак, в чем состоит задача, которую решает алгоритм? Представьте, что у вас есть функция, которая представляет собой “черный ящик”, принимающий на вход число из множества \(\{0, 1\}\). Функция неким образом обрабатывает входное значение и возвращает число из этого же множества, то есть либо \(0\), либо \(1\). Нам известно, что эта функция принадлежит либо к классу сбалансированных функций, либо к классу константных функций (которые мы также можем называть несбалансированными). Задача алгоритма - установить, к какому классу принадлежит функция.

Рассмотрим все варианты этих двух классов. Всего их четыре, то есть по две функции в каждом классе. Начнем с несбалансированных:

1). \(f_1(x) = 0\)
Это функция, всегда возвращающая \(0\), независимо от входного значения.

Для нее справедливы выражения:
$f_1(0) = 0$
$f_1(1) = 0$

2). $f_2(x) = 1$
Такая функция всегда возвращает 1, то есть верно следующее:
$f_2(0) = 1$
$f_2(1) = 1$

Ну а теперь посмотрим на сбалансированные функции.
Для них характерно то, что они могут возвращать как 0, так и 1. В этом и заключается “баланс”.

3). $f_3(x) = x$
Это тождественная функция, которая ничего не делает с входным значением.
Для нее справедливо следующее:
$f_3(0) = 0$
$f_3(1) = 1$

4). $f_4(x) = \overline{x}$
А вот эта функция инвертирует входное значение, то есть возвращает не то число,
которое было подано на вход, а другое:
$f_4(0) = 1$
$f_4(1) = 0$

Классический компьютер справляется с задачей за два шага. Например, нам дана некоторая функция-“черный ящик”, и мы должны установить, сбалансирована ли она. На первом шаге мы отправляем в функцию входное значение 0. Допустим, мы получили на выходе также 0. Мы можем сказать, что данная функция - либо f_1 (константная функция, всегда возвращающая 0), либо f_3 (сбалансированная функция, не меняющая входное значение). Для окончательного решения мы должны сделать еще один шаг - отправить в функцию значение 1. Если при этом мы получим опять 0, то это функция f_1, а если получили на выходе 1, то искомая функция - f_3.

Способа, с помощью которого на классическом компьютере можно за одно действие установить, сбалансирована функция или нет, не существует. И здесь свое преимущество показывает квантовый компьютер: он может установить класс функции за одно действие.

Для начала рассмотрим простейшую схему, с помощью которой можно отправлять число на вход и получать ответ от черного ящика:

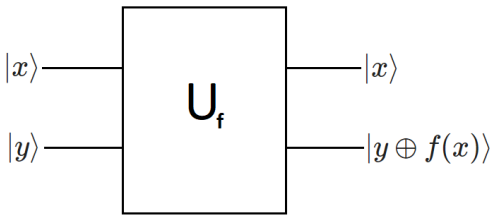


Fig. 8 Схема 1.

U_f на данной схеме - это неизвестная функция (ее также часто называют “оракул”), являющаяся унитарным оператором. Обратите внимание, что в квантовой схеме используются два кубита. Это нужно для того, чтобы информация, с которой работает квантовый компьютер, не стиралась. В квантовом компьютере важно, чтобы все действия с кубитами (кроме операции измерения) были обратимыми, а для этого информация должна сохраняться. В верхнем кубите будет записано входное значение, а в нижнем - выходное значение функции. Таким образом, входное значение не будет перезаписано значением, которое вернет функция.

Но нам важно будет не только сохранить значение $|x\rangle$, но также и не разрушить $|y\rangle$. Так как кубит y очевидно имеет некоторое изначальное значение, мы не можем его просто перезаписать тем числом, которое выдаст функция $f(x)$. Здесь на помощь приходит операция исключающее ИЛИ - XOR (также ее можно называть сложением по модулю 2), обозначенная на схеме как $\oplus$. В процессе работы черный ящик U_f не только находит значение $f(x)$, но и применяет исключающее ИЛИ к значениям y и $f(x)$.
Операции XOR соответствует такая таблица истинности:

a b a XOR b

0 0 0

0 1 1

1 0 1

1 1 0

Операция \(XOR\) хороша для нас тем, что она не разрушает значение \(|y\rangle\), так как является обратимой. Убедиться в этом можно, проверив тождество:
\((a \oplus b) \oplus b = a\)

Схема 1 пока что не дает преимущества по сравнению с классическим компьютером, но мы можем ее немного усовершенствовать:

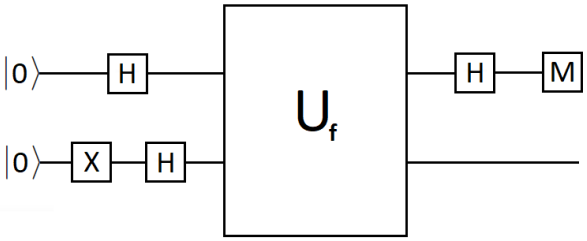


Fig. 9 Схема 2.

В новой схеме оба кубита вначале будут находиться в состоянии \(|0\rangle\). Затем мы применим к верхнему кубиту оператор Адамара, а к нижнему - гейт \(X\), а затем так же, как и к верхнему, оператор Адамара. Тем самым мы приведем оба кубита в состояние суперпозиции перед тем, как они попадут на вход функции \(U_f\). Верхний кубит будет находиться в такой суперпозиции:
\(|x\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle,\)
а нижний - в такой:
\(|y\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle.\)

Если рассмотреть это как систему \(|\psi\rangle\), состоящую из двух кубитов, то она будет выглядеть так:

\(|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)\)

Сразу после \(U_f\) система будет находиться в состоянии:

\(|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle)\)

После того, как \(U_f\) отрабатывает, нижний кубит, как это ни странно, уже нас не интересует, так что к нему операции больше не применяются, и его измерение также не производится.

Дело в том, что ответ на вопрос о том, сбалансирована функция \(f(x)\) или нет, будет нами получен из верхнего кубита после того, как на него подействует оператор Адамара и будет произведено измерение.

В том случае, если функция сбалансирована, результат измерения верхнего кубита будет равен \(1\), а если несбалансированна - \(0\).

Разберемся подробнее, почему это происходит.

Рассмотрим все возможные \(f(x)\), которые могут находиться в черном ящике:

1). \(f(x) = f_1\)

В этом случае \(f(x)\) всегда принимает значение \(0\), и система кубитов будет выглядеть так:

\(|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle) = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)\)

2). \(f(x) = f_2\)

\(f(x)\) будет равно \(1\) независимо от аргумента. Используя таблицу истинности XOR, легко убедиться, что во второй скобке \(|0\rangle\) и \(|1\rangle\) поменяются местами, но если вынести минус за скобку, то мы можем его не учитывать, так общий фазовый множитель (-1 в данном случае) для системы не имеет значения:

\(|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|1\rangle - |0\rangle) = -\frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle) = -\frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)\)

Видно, что при применении функций \(f_1\) и \(f_2\), являющихся несбалансированными, мы получаем фактически одно и тоже состояние. Если после этого применить к первому кубиту оператор Адамара, то после измерения мы получим значение \(0\).

Рассмотрим теперь сбалансированные функции \(f_3\) и \(f_4\).

3). \(f(x) = f_3\)

Здесь ситуация сложнее, так как \(f(x)\) будет зависеть от состояния первого кубита. Поэтому мы раскроем скобки, а значения функции подставим позже:

\(|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) = \frac{1}{2}(|0\rangle |0 \oplus f(x)\rangle - |0\rangle |1 \oplus f(x)\rangle + |1\rangle |0 \oplus f(x)\rangle - |1\rangle |1 \oplus f(x)\rangle)=\)

\(=\frac{1}{2}(|0\rangle |0 \oplus 0\rangle - |0\rangle |1 \oplus 0\rangle + |1\rangle |0 \oplus 1\rangle - |1\rangle |1 \oplus 1\rangle) = \frac{1}{2}(|00\rangle - |01\rangle + |11\rangle - |10\rangle) = \)

\(=\frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) = \frac{1}{2}(|0\rangle - |1\rangle)(|0\rangle - |1\rangle)\)

Видно, что первый кубит поменял свое состояние - теперь он в суперпозиции \(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\), так что далее к нему можно применить оператор Адамара, после которого он перейдет в состояние \(|1\rangle\).

4). \(f(x) = f_4\)

Здесь будет похожая ситуация:

\(|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) = \frac{1}{2}(|0\rangle |0 \oplus f(x)\rangle - |0\rangle |1 \oplus f(x)\rangle + |1\rangle |0 \oplus f(x)\rangle - |1\rangle |1 \oplus f(x)\rangle)=\)

\(=\frac{1}{2}(|0\rangle |0 \oplus 1\rangle - |0\rangle |1 \oplus 1\rangle + |1\rangle |0 \oplus 0\rangle - |1\rangle |1 \oplus 0\rangle) = \frac{1}{2}(|01\rangle - |00\rangle + |10\rangle - |11\rangle) = \)

\(=-\frac{1}{2}(|00\rangle - |01\rangle + |11\rangle - |10\rangle) =-\frac{1}{2}(|0\rangle - |1\rangle)(|0\rangle - |1\rangle)\)

Получили то же состояние \(|\psi\rangle\), что и для \(f_3\), с точностью до фазового множителя. Соответственно, здесь первый кубит после применения оператора Адамара также будет измерен с результатом \(1\).

Теперь можно получить более компактную формулу, которая подходит для всех четырех функций:

\(|\psi\rangle = \frac{1}{2}((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle)(|0\rangle - |1\rangle)\)

Задание: с помощью квантовых операторов попробуйте создать \(U_f\) для всех четырех \(f(x)\). (Задание рекомендуется сделать до прочтения программистской части по алгоритму Дойча, так как там содержится ответ).

Алгоритм Дойча в коде

Запрограммируем алгоритм с помощью библиотеки PennyLane. Предполагается, что функция, находящаяся в черном ящике, изначально присутствует, но для учебного примера создадим также и её, точнее, все её четыре варианта. Для того, чтобы нам не сразу было известно, какая из этих функций анализируется алгоритмом (иначе будет неинтересно), будем использовать случайный выбор функции.

Импортируем все необходимые библиотеки и модули, а также создадим квантовое устройство-симулятор, рассчитанное на схему из двух кубитов:

```
import pennylane as qml
from pennylane import numpy as np

dev = qml.device('default.qubit', shots=1, wires=2)
```

Теперь создадим функции для черного ящика. Обратите внимание, что здесь уже учтено сложение по модулю \(2\) результата функции с состоянием второго кубита:

```
def f1():
    pass

def f2():
    qml.PauliX(wires=[1])

def f3():
    qml.CNOT(wires=[0, 1])

def f4():
    qml.PauliX(wires=0)
    qml.CNOT(wires=[0, 1])
    qml.PauliX(wires=0)
```

Создадим словарь с функциями и их названиями:

```
black_boxes_dict = {'f1': f1, 'f2': f2, 'f3': f3, 'f4': f4}
```

А вот таким образом мы будем случайно выбирать название функции для черного ящика:

```
def random_black_box():
    black_boxes = ['f1', 'f2', 'f3', 'f4']
    n = np.random.randint(0, 4)
    return black_boxes[n]
```

А теперь самое важное - сам алгоритм Дойча:

```
@qml.qnode(dev)
def circuit(black_boxe_name):
    qml.Hadamard(wires=0)
    qml.PauliX(wires=1)
    qml.Hadamard(wires=1)
    black_boxes_dict[black_box_name]()
    qml.Hadamard(wires=0)
    return qml.sample(qml.PauliZ([0]))
```

Итак, подготовительные действия завершены, можно приступать к демонстрации работы алгоритма. Для начала выберем случайным образом функцию:

```
black_box_name = random_black_box()
```

А затем запустим алгоритм Дойча и выведем результат его работы. Собственное значение \(1\) оператора \(Z\) будет соответствовать состоянию \(|0\rangle\) (функция несбалансированна), а собственное значение \(-1\) - состоянию \(|1\rangle\) (функция сбалансирована):

```
result = circuit(black_box_name)
print(result)
```

```
1
```

Проверим, насколько правильно сработал алгоритм. Для этого посмотрим на функцию из черного ящика:

```
print(black_box_name)
```

```
f1
```

Также посмотрим, как выглядит квантовая схема:

```
print(circuit.draw())
```

```
 0: ──H──H─┤ Sample[Z]
 1: ──X──H─┤
```

На примере алгоритма Дойча мы видим, что уже двухкубитная схема дает прирост скорости в два раза. Если же увеличивать количество входных параметров (как в аналогичном алгоритме Дойча-Йожи), то ускорение будет экспоненциальным.

Для специалистов, занимающихся искусственным интеллектом, алгоритм может быть интересен тем, что не просто решает задачу нахождения некоторого значения, действуя как калькулятор, а дает возможность определить скрытую функцию. Это похоже на задачи машинного обучения, когда дата сайентист, производя математические манипуляции с данными, в итоге получает модель (фактически - функцию), описывающую связь признаков с целевой переменной. Таким образом, интерес специалистов ИИ к квантовым вычислениям, вполне понятен, как и перспективы квантовых вычислений в этой области.

Алгоритм Гровера

Одно из самых востребованных действий в работе с данными – поиск по базе данных. При использовании классического компьютера такой поиск в худшем случае требует N операций, где N – количество строк в таблице. В среднем найти нужный элемент можно за $N/2$ операций.

Фактически, это означает, что если мы не знаем, где расположен нужный элемент в таблице, то придется перебирать все элементы, пока не найдем то, что нужно. Для классических вычислений это нормально, но что, если у нас есть квантовый компьютер?

Если наша база данных работает на основе квантовых вычислений, то мы можем применить алгоритм Гровера, и тогда такой поиск потребует всего порядка $\sqrt{N}$ действий. Конечно же, такое ускорение не будет экспоненциальным, как при использовании некоторых других квантовых алгоритмов, но оно будет квадратичным, что также довольно неплохо.

Fig. 10 Лов Гровер

> **Лов Гровер**
>
> Индо-американский ученый в сфере Computer Science Лов Кумар Гровер предложил квантовый алгоритм поиска по базе данных в 1996 году. Этот алгоритм считается вторым по значимости для квантовых вычислений после алгоритма Шора. Впервые он был реализован на простейшем квантовом компьютере в 1998 году, а в 2017 году алгоритм Гровера был впервые запущен для трехкубитной базы данных.

Итак, наша задача состоит в том, что мы должны найти идентификационный номер (Id) записи, которая удовлетворяет определенным условиям. Функция-оракул находит такую запись (для простоты будем сначала считать, что такая запись одна) и помечает соответствующий ей Id. Отметка делается достаточно оригинальным способом: Id умножается на -1.

Для полной ясности соотнесем количество Id с числом кубитов в квантовой схеме. Здесь все очень просто: имея n кубитов, можно закодировать $N = 2^n$ идентификаторов. К примеру, если в таблице базы данных 1024 записей, то закодировать все Id можно с помощью десяти кубитов.

Для того, чтобы не запутаться в квантовых операциях, рассмотрим пример поменьше: с помощью двух кубитов закодируем четыре идентификационных номера, один из которых будет помечен функцией-оракулом как искомый – он будет домножен на -1. Все эти четыре числа могут существовать в квантовой схеме одновременно, если кубиты приведены в состояние суперпозиции.

Пусть искомый Id равен 11 (будем пользоваться двоичной системой и вести счет с нуля), тогда после работы функции-оракула мы будем иметь 4 состояния: $|00\rangle$, $|01\rangle$, $|10\rangle$, $-|11\rangle$. Проблема в том, что если измерить эту схему, то с равной вероятностью будет обнаружено одно из этих четырех значений, но узнать, какое из них функция-оракул пометила минусом, будет невозможно.

Получается, что одной функции-оракула недостаточно, нужно что-то дополнительное. На помощь приходит алгоритм Гровера. Правда, у него есть такая особенность – он является итерационным, то есть определенные операции (в том числе и применение функции-оракула) нужно повторить несколько раз (порядка $\sqrt{N}$). Причем, с количеством итераций нельзя ошибиться, иначе алгоритм даст неправильный ответ.

В идеале после всех итераций квантовую схему можно будет измерить и получить значение Id искомой записи в таблице базы данных.

Разберем операции, которые включает в себя каждая итерация, но перед этим добавим в схему еще один кубит, который мы будем называть вспомогательным. Он нужен для хранения метки искомого индекса. Звучит не совсем понятно, но ничего сложного в этом нет, все станет ясным после разбора работы функции-оракула. Итак, наша база данных двухкубитная, но сама схема состоит из трех кубитов.

Квантовая схема выглядит так:

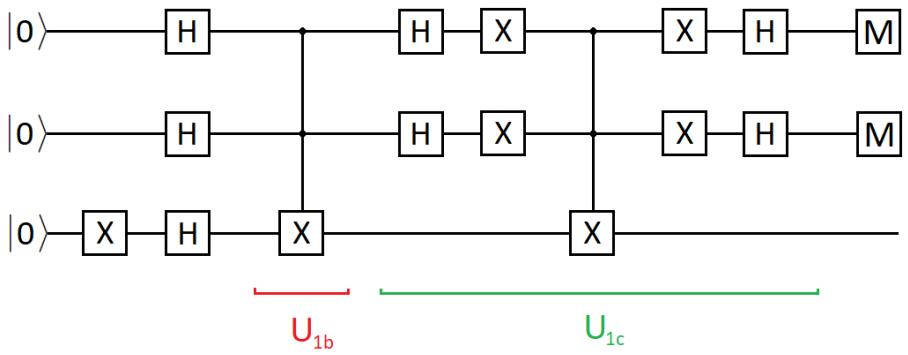


Fig. 11 Алгоритм Гровера для n=2 (искомый индекс 11).

В самом начале, еще до всех итераций, все кубиты (включая вспомогательный) должны быть приведены в состояние суперпозиции с помощью оператора Адамара. Причем начальное состояние всех кубитов должно быть равно 0, кроме вспомогательного кубита – до действия оператора Адамара он должен быть приведен в состояние 1.

Таким образом, вспомогательный кубит после применения оператора Адамара будет находиться в состоянии $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, тогда как остальные кубиты примут состояние $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

Далее начинаем итерации. Каждая итерация состоит из двух частей. Первая часть – это действие функции-оракула. Это некоторая функция, умеющая эффективно определять, какой индекс соответствует искомому объекту. Эта функция не может сообщить нам этот индекс напрямую, зато она может пометить его минусом.

Для разбора внутренней работы алгоритма нам потребуется задать функцию-оракул вручную и сделать ее достаточно простой, поэтому нужно знать, что в рабочих условиях она будет действовать похоже, но будет устроена, скорее всего, по-другому, так как предназначена для конкретной задачи выбора искомых данных. Мы не будем касаться вопроса конкретной реализации функции-оракула для выбора определенных данных, так как это уже другой вопрос, не влияющий на принцип алгоритма Гровера.

Для того чтобы понять алгоритм Гровера, мы должны будем понять, какие изменения происходят с состояниями кубитов до того момента, когда производится измерение, выдающее искомый индекс.

Мы договорились, что в нашей учебной задаче искомый Id равен 11, так что в результате измерения мы должны получить именно это значение. Смоделируем оракул, который будет помечать этот индекс. В качестве такого оракула подойдет гейт Тоффоли ($CCNOT$). При подаче на оба его управляющих входа значений 1, он будет применять к управляемому кубиту (это как раз будет вспомогательный кубит) гейт X.

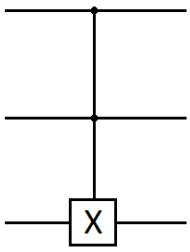


Fig. 12 Гейт Тоффоли.

На состояния верхних кубитов, кодирующих индексы \(00\), \(01\) и \(10\) гейт Тоффоли не будет реагировать, и вспомогательный кубит будет находиться в состоянии \(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\).

Но при срабатывании гейта на индексе \(11\) к вспомогательному кубиту применится оператор \(X\), так что вспомогательный кубит примет состояние \(\frac{1}{\sqrt{2}}(|1\rangle - |0\rangle)\), или, если это состояние записать по-другому, за скобками появится минус: \(-\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\).

Этот минус относится не только к вспомогательному кубиту, но и ко всему состоянию, соответствующему индексу \(11\), так что можно считать, что вспомогательный кубит остался неизменным, и отнести минус к состоянию \(11\) верхних кубитов. Таким образом, индекс \(11\) помечен минусом как искомый. Другими словами, функция-оракул перевела состояние \(|11\rangle |q_{helper}\rangle\) в состояние \(-|11\rangle |q_{helper}\rangle\), где \(|q_{helper}\rangle\) – вспомогательный кубит.

Запишем состояние квантовой схемы после применения оракула (состояние вспомогательного кубита – скобка справа):

\(|\psi\rangle = \frac{1}{2\sqrt{2}}(|00\rangle + |01\rangle + |10\rangle - |11\rangle)(|0\rangle - |1\rangle)\)

Итак, первая часть первой итерации завершена. Искомый индекс помечен, но если измерить кубиты прямо сейчас, то это ничего не даст – минус не проявит себя при измерении. Да и сам индекс \(11\) будет получен только с вероятностью \(0.25\) – такой же, как и у других индексов.

Для того, чтобы лучше понять дальнейшие действия, представим первую половину работы алгоритма в виде рисунка, показывающего вектор текущего состояния. В качестве единичного вектора горизонтальной оси мы будем использовать все состояния из суперпозиции кроме того, который соответствует искомому индексу, а вертикальным единичным вектором будет искомый вектор.

Вектор \(c\) – состояние системы перед первой итерацией – является линейной комбинацией векторов, соответствующим горизонтальной и вертикальной осям.

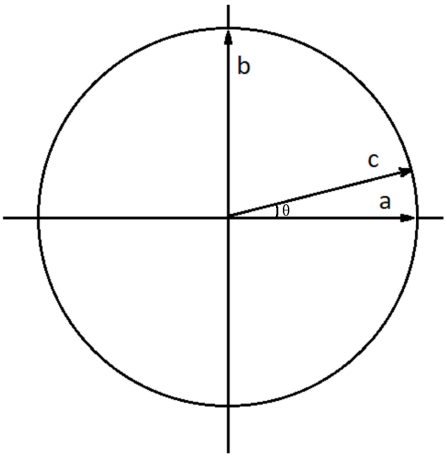


Fig. 13 Состояние системы перед первой итерацией.

Можно выразить вектор \(c\) для нашего случая (системы из двух кубитов с искомым индексом \(11\)), обозначив его координаты за \(x\) и \(y\):

\(\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = x \frac{1}{\sqrt{3}}(|00\rangle + |01\rangle + |10\rangle) + y |11\rangle\)

Из данного уравнения находим \(x = \frac{\sqrt{3}}{2}\) и \(y = \frac{1}{2}\).

По этим координатам можно понять, что угол между вектором \(c\) и горизонтальной осью (обозначим его как \(\theta\)) равен \(\frac{\pi}{6}\). Если забежать немного вперед, то можно сказать, что наша цель – добиться, чтобы текущее состояние дошло до \(\frac{\pi}{2}\) (или хотя бы приблизительно), то есть почти или полностью равнялось искомому состоянию, так что после измерения можно было его и получить с высокой вероятностью.

Координаты текущего вектора состояния можно записать через угол \(\theta\):

\(x = \cos{\theta}\)

\(y = \sin{\theta}\)

На всякий случай нужно уточнить, что вспомогательный кубит не отражается на рисунке с окружностью, так как он не предназначен для обозначения индекса, а только хранит в себе его метку.

После применения функции-оракула текущий вектор отразится относительно горизонтальной оси. Объясняется это очень легко – его вертикальная компонента (вектор \(|11\rangle\)) становится отрицательной.

Вектор \(c_{1b}\) – это отражение вектора \(c\) на угол \(\theta\) вниз относительно горизонтальной оси:

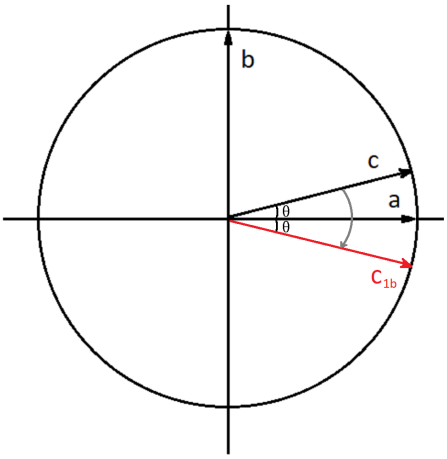


Fig. 14 Состояние системы после первой части первой итерации.

Такое отражение в нашем примере производится с помощью операции \(CCNOT\), но в общем случае операция выглядит так:

\(U_{1b} = I - 2|b\rangle \langle b|\)

Функцию-оракул мы здесь обозначили как \(U_{1b}\). Она меняет знак только для вертикальной составляющей вектора состояния, поэтому и происходит отражение.

Проверим формулу в действии, применив ее для нашего примера:

\(U_{1b} |c\rangle = (I - 2|11\rangle \langle 11|) \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle - 2|11\rangle) = \frac{1}{2} (|00\rangle + |01\rangle + |10\rangle - |11\rangle)\)

И наконец приступаем к разбору второй части первой итерации. В ней будет происходить еще одно отражение вектора, но уже не относительно горизонтальной оси, а относительно вектора \(c\). Нетрудно заметить, что при этом текущий вектор состояния станет равен \(\cos{3 \theta}|a\rangle + \sin{3 \theta}|b\rangle\).

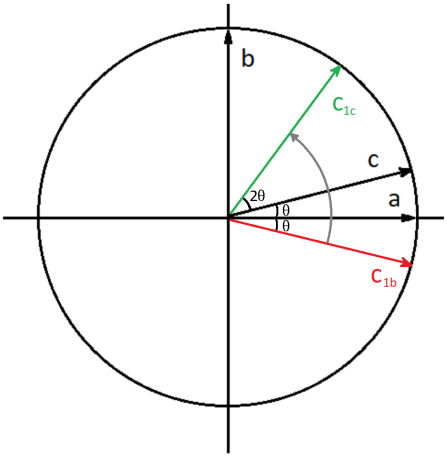


Fig. 15 Состояние системы после второй части первой итерации.

Операция для получения вектора \(c_{1c}\) будет выглядеть так:

\(U_{1c} = 2|c\rangle \langle c| - I\)

Посчитаем, чему равен вектор \(c_{1c}\) для нашего примера:

\(U_{1c} |c_{1b}\rangle = (2|c\rangle \langle c| - I) \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) = |11\rangle\)

Произошло отражение текущего вектора состояния \(|c_{1b}\rangle\) относительно вектора \(|c\rangle\). Если представить \(|c_{1b}\rangle\) как \(k_1 |c\rangle + k_2 |c_{\bot}\rangle\), где \(|c_{\bot}\rangle\) – вектор, перпендикулярный \(|c\rangle\), а \(k_1\) и \(k_2\) – действительные коэффициенты, то тогда отраженный вектор будет равен \(k_1 |c\rangle - k_2 |c_{\bot}\rangle\).

В нашей квантовой схеме эта часть итерации реализована таким образом:

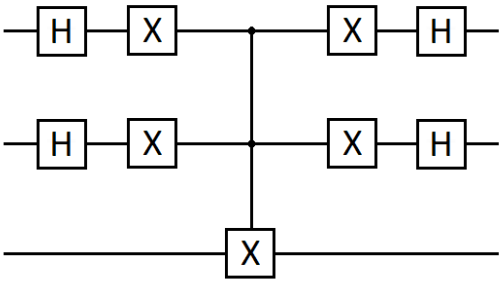


Fig. 16 Вторая часть итерации.

Вначале применяется оператор Адамара для первого и второго кубитов. Это упрощает нашу задачу, так как теперь отразить вектор состояния нужно не относительно состояния суперпозиции \(\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)\), а относительно состояния \(|00\rangle\).

Далее требуется отразить вектор, то есть всем состояниям кроме нулевого присвоить минус, но мы сделаем проще: присвоим минус нулевому состоянию \(|00\rangle\), а остальные состояния, составляющие суперпозицию, оставим как есть. Сделаем это, последовательно применив гейты \(X\) и \(CCNOT\). После этого вернемся в исходную “систему координат”, применив операции в обратном порядке: сначала \(X\), а потом \(H\).

Из-за применения такого лайфхака (присвоения минуса нулевому состоянию) мы в нашем двухкубитном примере получим результат с точностью до общей фазы: не \(|11\rangle\), а \(-|11\rangle\). Но это не страшно, так как после измерения мы все равно увидим искомое значение индекса.

По рисунку, изображающему на окружности состояние системы после второй части первой итерации, видно, что в общем случае каждая последующая итерация будет приближать текущий вектор к вертикальному. Но в нашем случае угол между вектором состояния и горизонтальной осью после окончания первой итерации равен \(3 \theta\), то есть это уже и есть желаемый угол \(\frac{\pi}{2}\).

В общем случае этот угол равен $(2t + 1) \theta \approx \frac{\pi}{2}$, где t – номер произведенной итерации. Отсюда можно вывести число итераций, необходимое для работы алгоритма. При большом количестве итераций t угол θ будет стремиться к 0, так что его можно заменить на $\sin{\theta}$, который, в свою очередь, равен $\frac{1}{\sqrt{N}}$:

$$(2t + 1) \theta \rightarrow (2t + 1) \frac{1}{\sqrt{N}} \approx \frac{\pi}{2}$$

Если пренебречь единицей в скобках на основании того, что t – большое число, можно найти, что t приблизительно равно $\frac{\pi \sqrt{N}}{4}$.

Мы уже разобрались, что каждая итерация состоит из двух частей. Первая часть – отражение вниз относительно горизонтальной оси. Вторая часть – отражение вверх относительно изначального состояния, то есть вектора c. Вектор состояния всегда будет отражаться вверх на больший угол, чем в первой части итерации. Этим и будет обеспечиваться его постепенное приближение в вертикальной оси.

Мы разобрали случай, когда требуется найти один объект в таблице. Если же потребуется найти несколько объектов, то тогда, обозначив их количество за K, мы должны будем проделать около $\frac {\pi}{4} \sqrt {\frac{N}{K}}$ итераций. Таким образом, для успешной работы алгоритма Гровера необходимо знать число K, чтобы можно было найти через него угол θ, а затем число итераций.

Реализация алгоритма Гровера

Итак, мы разобрали общие принципы алгоритма Гровера, а также рассмотрели конкретный пример. Настало время написать для этого примера программу.

Для начала импортируем все необходимые библиотеки и создадим схему из трех кубитов:

```
import pennylane as qml
from pennylane import numpy as np

dev = qml.device('default.qubit', shots=1, wires=3)
```

Начальная функция, создающая суперпозицию для каждого кубита:

```
def U_start():
    qml.PauliX(wires=2)
    for i in range(3):
        qml.Hadamard(wires=i)
```

Создадим функцию, действующую аналогично оракулу (первая часть итерации). Эта функция помечает значение индекса 11:

```
 def U_b():
    qml.Toffoli(wires=[0, 1, 2])
```

Вторая часть итерации:

```
 def U_c():
    for i in range(2):
        qml.Hadamard(wires=i)
        qml.PauliX(wires=i)

    qml.Toffoli(wires=[0, 1, 2])
    for i in range(2):
        qml.PauliX(wires=i)
        qml.Hadamard(wires=i)
```

Объединим первую и вторую часть итерации в одну функцию:

```
def U_iteration():
    U_b()
    U_c()
```

Переходим к итоговой функции, содержащей все шаги, а также производящей измерение кубитов в конце. В аргументе N мы должны будем указать количество итераций:

```
@qml.qnode(dev)
def circuit(N: int):
    U_start()
    for t in range(N):
        U_iteration()
    return qml.probs(wires=[0, 1])
```

Запускаем функцию и выведем ее результат:

```
print(circuit(N=1))
```

```
[0. 0. 0. 1.]
```

Так как в качестве искомого индекса выступало значение \(11\), то в результате запуска функции мы должны получить массив, состоящий из вероятностей каждого индекса, в котором искомый индекс (в нашем примере он будет последним в массиве) должен иметь наибольшую вероятность. Параметр устройства `shots` при необходимости можно увеличивать, не забывая о том, что его увеличение будет кратно замедлять алгоритм. Таким образом, мы нашли с помощью алгоритма Гровера искомый индекс.

Алгоритм Гровера может применяться не только для задач простого поиска в базе данных, но и как дополнительное средство ускорения для поиска экстремума целочисленной функции, а также для поиска совпадающих строк в базе данных, так что этот алгоритм, как и его модификации, сможет быть полезным в разнообразных задачах Data Science.

Задание

1. Распишите операторы \(U_{1b}\) и \(U_{1c}\) из примера в виде матриц \(4\)x\(4\) и проведите расчеты для получения \(c_{1b}\) и \(c_{1c}\) в виде векторов-столбцов.
2. Модифицируйте приведенный выше код алгоритма Гровера для двухкубитной базы данных так, чтобы искомый индекс соответствовал состоянию \(|00\rangle\).
