

Министерство науки и высшего образования РФ  
ФГБОУ ВО «Тверской государственный университет»  
Математический факультет  
Направление 02.04.01 Математика и компьютерные науки  
Профиль «Математическое и компьютерное моделирование»

## МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Вариационный квантовый алгоритм с оптимизацией  
методом отжига

Автор:  
Алешин Д.А.  
Подпись:

Научный руководитель:  
д. ф.-м. н. Цирулёв А.Н.  
Подпись:

Допущен к защите:  
Руководитель ООП: Цветков В.П.

---

*(подпись, дата)*

Тверь 2025

# Оглавление

Введение	3
<b>1 Общая схема квантовых вариационных алгоритмов</b>	<b>5</b>
1.1 Базис Паули . . . . .	5
1.1.1 Связь стандартного базиса и базиса Паули . . . . .	5
1.1.2 Коммутационное и антикоммутационное соотношение	8
1.1.3 Коммутирующие элементы в алгебре Ли группы $SU(2^n)$ . . . . .	9
1.2 Вариационная квантовая оптимизация . . . . .	11
1.3 Анзац и общая схема алгоритма . . . . .	13
1.4 Пример, иллюстрирующий особенности алгоритма . . . . .	15
<b>2 Вариационный квантовый алгоритм на основе метода отжига</b>	<b>21</b>
2.1 Метод отжига . . . . .	21
2.2 Алгоритм . . . . .	23
2.3 Сравнительные результаты тестирования . . . . .	30
Заключение	32
Литература	33
Приложения	36

# Введение

В последние годы вариационные квантовые алгоритмы приобретают всё большее значение в современных исследованиях по математическому моделированию и квантовым вычислениям. Особый интерес представляют гибридные квантово-классические методы, в которых оптимизация параметров квантовых схем сочетается с классическими алгоритмами поиска минимума. Такие подходы позволяют эффективно моделировать и исследовать малоразмерные квантовые системы, актуальные для описания новых состояний вещества, включая топологические материалы [3].

Важной прикладной задачей в области вариационных квантовых алгоритмов является поиск основного состояния гамильтониана, что эквивалентно задаче минимизации средней энергии в пространстве квантовых состояний, генерируемых параметризованным анзацем. На практике решение такой задачи требует эффективных методов оптимизации, устойчивых к наличию большого числа локальных минимумов и не требующих вычисления производных целевой функции. Среди таких методов особое место занимает метод имитации отжига, который широко используется в задачах глобальной оптимизации и хорошо адаптируется к вариационным квантовым схемам.

Целью данной работы является построение и исследование вариационного квантового алгоритма с классической оптимизацией методом имитации отжига для поиска основного состояния гамильтониана квантовой системы.

В рамках исследования были поставлены следующие основные задачи:

1. Изучить теоретические основы вариационных квантовых алгоритмов, включая формализм базиса Паули и построение анзаца для

многокубитных систем;

2. Разработать и реализовать алгоритм вариационной квантовой оптимизации с использованием метода имитации отжига;
3. Провести сравнительный анализ эффективности различных параметров отжига и схем построения анзаца;
4. Реализовать программную модель алгоритма на языке Python для практического моделирования процесса вариационной оптимизации.

Объектом исследования является задача минимизации энергии гамильтониана в пространстве состояний, порождённых вариационным анзацем в базисе Паули. Предметом исследования выступает разработка и программная реализация гибридного квантового алгоритма с классическим оптимизатором на методе имитации отжига.

Структура работы включает две основные главы. В первой главе изложены теоретические основы вариационных квантовых алгоритмов, формализм базиса Паули и вопросы построения параметризованного анзаца. Вторая глава посвящена описанию метода имитации отжига, его адаптации к задачам квантовой оптимизации и программной реализации алгоритма, а также анализу результатов тестирования. В заключении приведены основные выводы по проделанной работе. Список литературы включает современные публикации российских и зарубежных авторов, посвящённые тематике квантовых вычислений и оптимизации.

В работе везде принята система единиц, в которой постоянная Планка  $\hbar = 1$  и скорость света  $c = 1$ , что соответствует общепринятому подходу в теоретической физике.

# Глава 1

## Общая схема квантовых вариационных алгоритмов

### 1.1 Базис Паули

#### 1.1.1 Связь стандартного базиса и базиса Паули

Рассмотрим квантовую систему из  $n$  кубитов, где каждый кубит связан с двумерным гильбертовым пространством  $\mathcal{H}$  и его эрмитово сопряжённым пространством  $\mathcal{H}^\dagger$ . Обозначим через  $\mathcal{H}_n = \mathcal{H}^{\otimes n}$  и  $\mathcal{H}_n^\dagger = (\mathcal{H}^\dagger)^{\otimes n}$  гильбертово пространство системы и его эрмитово сопряжение соответственно. Пространство линейных операторов, действующих на  $\mathcal{H}$  и  $\mathcal{H}^\dagger$  левым и правым умножением, задаётся как  $L(\mathcal{H}_n) = \mathcal{H}_n \otimes \mathcal{H}_n^\dagger$ . Тогда

$$\dim_{\mathbb{C}} \mathcal{H}_n = \dim_{\mathbb{C}} \mathcal{H}_n^\dagger = 2^n, \quad \dim_{\mathbb{C}} L(\mathcal{H}_n) = 2^{2n}.$$

Пространство  $L(\mathcal{H}_n)$  наделено скалярным произведением Гильберта-Шмидта:

$$\langle \hat{A}, \hat{B} \rangle = \text{tr}(\hat{A}^\dagger \hat{B}), \quad \hat{A}, \hat{B} \in L(\mathcal{H}_n), \quad (1.1)$$

которое естественно продолжает скалярное произведение в  $\mathcal{H}_n$ . Вещественное линейное пространство эрмитовых операторов далее обозначим как  $H(\mathcal{H}_n)$ .

Пусть  $\{|0\rangle, |1\rangle\}$  образуют ортонормированный базис в однокубитном пространстве  $\mathcal{H}$ . Единичная матрица и матрицы Паули задаются как:

$$\sigma_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

а соответствующие операторы Паули представляются в виде:

$$\begin{aligned} \hat{\sigma}_0 &= |0\rangle\langle 0| + |1\rangle\langle 1|, \quad \hat{\sigma}_1 = |0\rangle\langle 1| + |1\rangle\langle 0|, \\ \hat{\sigma}_2 &= -i|0\rangle\langle 1| + i|1\rangle\langle 0|, \quad \hat{\sigma}_3 = |0\rangle\langle 0| - |1\rangle\langle 1|. \end{aligned}$$

Эти операторы одновременно эрмитовы и унитарны, а также образуют базис в  $\mathcal{H}$ . Обратное преобразование выражается следующим образом:

$$|0\rangle\langle 0| = \frac{\hat{\sigma}_0 + \hat{\sigma}_3}{2}, \quad |0\rangle\langle 1| = \frac{\hat{\sigma}_1 + i\hat{\sigma}_2}{2}, \quad |1\rangle\langle 0| = \frac{\hat{\sigma}_1 - i\hat{\sigma}_2}{2}, \quad |1\rangle\langle 1| = \frac{\hat{\sigma}_0 - \hat{\sigma}_3}{2}.$$

Для  $k, l, m \in \{1, 2, 3\}$  выполняются свойства:  $\text{tr} \hat{\sigma}_k = 0$ ,  $\hat{\sigma}_k^2 = \hat{\sigma}_0$ , а также

$$\hat{\sigma}_k \hat{\sigma}_l = -\hat{\sigma}_l \hat{\sigma}_k, \quad \hat{\sigma}_k \hat{\sigma}_l = i \text{sign}(\pi) \hat{\sigma}_m, \quad (klm) = \pi(123), \quad (1.2)$$

где  $\pi(123)$  — произвольная перестановка множества  $\{1, 2, 3\}$ .

Рассмотрим стандартный<sup>1</sup> бинарный базис в  $\mathcal{H}_n$ , образованный ортонормированными базисами  $\{|0\rangle, |1\rangle\}$  в однокубитных пространствах. Позиция в тензорном произведении позволяет различать кубиты. Для фиксированного  $n$  элементы этого базиса и соответствующие им элементы двумерного базиса удобно записывать как:

$$|k\rangle = |k_1 \dots k_n\rangle = |k_1\rangle \otimes \dots \otimes |k_n\rangle, \quad \langle k| = \langle k_1 \dots k_n| = \langle k_1| \otimes \dots \otimes \langle k_n|,$$

где строки  $k_1 \dots k_n$  ( $k_1, \dots, k_n \in \{0, 1\}$ ) интерпретируются как двоичные числа с десятичным представлением  $k$ . Например,  $|101\rangle = |5\rangle$  и  $|001110\rangle = |6\rangle$ .

---

<sup>1</sup>Мы избегаем термина «вычислительный», так как он может приводить к неоднозначности. И базис Паули, и стандартный базис являются вычислительными в одинаковом контексте.

В стандартном базисе:

$$|u\rangle = \sum_{k=0}^{2^n-1} u_k |k\rangle, \quad \hat{A} = \sum_{k,l=0}^{2^n-1} a_{kl} |k\rangle\langle l|,$$

где  $|u\rangle \in \mathcal{H}_n$  и  $\hat{A} \in L(\mathcal{H}_n)$ .

Базис Паули  $P(\mathcal{H}_n)$  в  $L(\mathcal{H}_n)$  определяется как:

$$\{\hat{\sigma}_{k_1\dots k_n}\}_{k_1,\dots,k_n \in \{0,1,2,3\}}, \quad \hat{\sigma}_{k_1\dots k_n} = \hat{\sigma}_{k_1} \otimes \dots \otimes \hat{\sigma}_{k_n}, \quad (1.3)$$

где  $\hat{\sigma}_{0\dots 0}$  — тождественный оператор. Базис  $P(\mathcal{H}_n)$  содержит  $4^n$  элементов. Для краткости будем использовать обозначение:

$$\hat{\sigma}_K = \hat{\sigma}_{k_1\dots k_n},$$

где строка Паули  $k_1 \dots k_n$  ( $k_1, \dots, k_n \in \{0, 1, 2, 3\}$ ) соответствует числу  $K$  в десятичной системе ( $0 \leq K \leq 4^n - 1$ ). Строка Паули  $K$  и элемент  $\hat{\sigma}_K$  взаимно однозначно соответствуют друг другу.

Сравним  $P(\mathcal{H}_n)$  со стандартным базисом. Для элементов базиса Паули выполняются:

$$\hat{\sigma}_{k_1\dots k_n} \hat{\sigma}_{k_1\dots k_n} = \hat{\sigma}_{0\dots 0}, \quad \text{tr } \hat{\sigma}_{0\dots 0} = 2^n, \quad \text{tr } \hat{\sigma}_{k_1\dots k_n} \Big|_{k_1\dots k_n \neq 0\dots 0} = 0. \quad (1.4)$$

Базис Паули является эрмитовым, унитарным и ортогональным относительно скалярного произведения (1.1). Отметим, что оператор  $|k\rangle\langle l|$  из стандартного базиса не является унитарным или эрмитовым при  $k \neq l$ . Стандартный базис не включает тождественный оператор, который в этом базисе записывается как:

$$\sum_{k=0}^{2^n-1} |k\rangle\langle k|.$$

В базисе Паули любой оператор  $\hat{U}$  из унитарной группы  $U(\mathcal{H}_n)$  (где  $\hat{U}^\dagger \hat{U} = \hat{\sigma}_{0\dots 0}$ ) раскладывается в виде:

$$\hat{U} = \sum_{i_1,\dots,i_n \in \{0,1,2,3\}} U_{i_1\dots i_n} \hat{\sigma}_{i_1\dots i_n}, \quad \hat{U}^\dagger = \sum_{i_1,\dots,i_n \in \{0,1,2,3\}} \overline{U}_{i_1\dots i_n} \hat{\sigma}_{i_1\dots i_n},$$

где коэффициенты удовлетворяют условиям:

$$\sum_{i_1, \dots, i_n \in \{0,1,2,3\}} \bar{U}_{i_1 \dots i_n} U_{i_1 \dots i_n} = 1, \quad \sum_{\substack{i_1, \dots, i_n, j_1, \dots, j_n \in \{0,1,2,3\} \\ (i_1, \dots, i_n) \neq (j_1, \dots, j_n)}} \bar{U}_{i_1 \dots i_n} U_{j_1 \dots j_n} = 0.$$

Последнее условие эквивалентно  $2^{2n-1}(2^n - 1)$  независимым соотношениям.

Эрмитовы операторы в базисе Паули разлагаются с вещественными коэффициентами.

### 1.1.2 Коммутационное и антикоммутационное соотношение

Коммутатор определяет взаимодействие операторов при их перестановке. Для операторов  $A$  и  $B$  он задаётся как:

$$[A, B] = AB - BA.$$

Если  $[A, B] = 0$ , операторы коммутируют; в противном случае — нет. В базисе Паули коммутаторы выражаются через символ Леви-Чивиты  $\varepsilon_{ijk}$ :

$$[\sigma_i, \sigma_j] = 2i\varepsilon_{ijk}\sigma_k,$$

где  $\varepsilon_{ijk}$  равен 1 при чётной перестановке индексов  $(i, j, k)$ ,  $-1$  при нечётной и 0 в остальных случаях. Например:

$$[\sigma_1, \sigma_2] = 2i\sigma_3, \quad [\sigma_2, \sigma_3] = 2i\sigma_1.$$

Антикоммутатор характеризует симметричное произведение операторов:

$$\{A, B\} = AB + BA.$$

Если  $\{A, B\} = 0$ , операторы антикоммутируют. Для операторов Паули:

$$\{\sigma_i, \sigma_j\} = 2\delta_{ij},$$

где  $\delta_{ij}$  — символ Кронекера (1 при  $i = j$ , 0 иначе). Примеры:

$$\{\sigma_1, \sigma_2\} = 0, \quad \{\sigma_2, \sigma_3\} = 0.$$



Коммутаторы и антикоммутаторы применяются в квантовой механике для анализа свойств систем (спин электрона, кубиты), в квантовой теории поля (взаимодействия частиц) и квантовых вычислениях (алгоритмы, коррекция ошибок). Коммутаторы помогают определить совместную измеримость наблюдаемых, а антикоммутаторы — описать фермионные системы.

### 1.1.3 Коммутирующие элементы в алгебре Ли группы $SU(2^n)$

Рассмотрим базис Паули и множество строк Паули длины  $n$ :

$$\text{Str}_n = \{K = k_1 \dots k_n\}_{k_1, \dots, k_n \in \{0, 1, 2, 3\}}.$$

1. Множество  $\mathbb{F}_4 = \{0, 1, 2, 3\}$  образует квадгруппу Клейна с умножением:

$$0 \cdot k = k, \quad k \cdot k = 0, \quad k \cdot l = m,$$

где  $k, l, m \in \{1, 2, 3\}$  и  $(klm)$  — произвольная перестановка  $(1\ 2\ 3)$ .

2. Функция  $s : \mathbb{F}_4 \times \mathbb{F}_4 \rightarrow \{1, i, -i\}$  задаётся значениями:

$$\begin{aligned} s(0, 0) = s(0, k) = s(k, 0) = s(k, k) &= 1, \quad k = 1, 2, 3, \\ s(1, 2) = s(2, 3) = s(3, 1) &= i, \quad s(2, 1) = s(3, 2) = s(1, 3) = -i. \end{aligned}$$

3. Функция  $S : \text{Str}_n \times \text{Str}_n \rightarrow \{1, -1, i, -i\}$  определяется как:

$$S_{KL} = s(k_1, l_1) \cdot s(k_2, l_2) \cdot \dots \cdot s(k_n, l_n),$$

где  $K = k_1 k_2 \dots k_n$  и  $L = l_1 l_2 \dots l_n$ .

Симметрия функции  $S$  зависит от числа пар  $(k_r, l_r)$  (на позициях  $r$  в строках  $K$  и  $L$ ), где  $k_r, l_r \in \{1, 2, 3\}$  и  $k_r \neq l_r$ , а также от их взаимного порядка. Пусть  $\omega_{KL}^+$  и  $\omega_{KL}^-$  — количество пар вида  $(1, 2), (2, 3), (3, 1)$  и  $(2, 1), (3, 2), (1, 3)$  соответственно, и  $\omega_{KL} = \omega_{KL}^+ + \omega_{KL}^-$ . Тогда

$$S_{(KL)} = \frac{S_{KL}}{2} (1 + (-1)^{\omega_{KL}}), \quad S_{[KL]} = \frac{S_{KL}}{2} (1 - (-1)^{\omega_{KL}}), \quad (1.5)$$

где

$$S_{KL} = i^{\omega_{KL}} (-1)^{\omega_{KL}^-}.$$

$\omega_{KL} \bmod 4$	0	2	0	2	1	3	1	3
$\omega_{KL}^- \bmod 4$	0	1	1	0	0	1	1	0
$S_{KL}$		1	-1	-1	$i$	$i$	$-i$	$-i$
$S_{(KL)}$	1	1	-1	-1	0	0	0	0
$S_{[KL]}$	0	0	0	0	$i$	$i$	$-i$	$-i$

Таблица 1: Множитель до  $\hat{\sigma}_M$  в (1.6) для  $\hat{\sigma}_K \hat{\sigma}_L$ ,  $\{\hat{\sigma}_K, \hat{\sigma}_L\}$ , и  $[i\hat{\sigma}_K, i\hat{\sigma}_L]$ .

Здесь  $S_{(KL)}$  и  $S_{[KL]}$  — симметричная и антисимметричная части  $S_{KL}$ . Значения  $S_{KL}$ ,  $S_{(KL)}$  и  $S_{[KL]}$  приведены в таблице 2.

Композицию элементов базиса Паули, их антикоммутаторов и коммутаторов можно компактно выразить в виде, удобном для программной реализации:

$$\hat{\sigma}_K \hat{\sigma}_L = S_{KL} \hat{\sigma}_M, \quad \{\hat{\sigma}_K, \hat{\sigma}_L\} = S_{(KL)} \hat{\sigma}_M, \quad [i\hat{\sigma}_K, i\hat{\sigma}_L] = -S_{[KL]} \hat{\sigma}_M, \quad (1.6)$$

где

$$\hat{\sigma}_M = \hat{\sigma}_{m_1 \dots m_n}, \quad m_r = k_r \cdot l_r \quad (r = 1, \dots, n). \quad (1.7)$$

Две строки Паули длины  $n$  могут коммутировать, даже имея ненулевые элементы в одних и тех же позициях. Например, операторы  $\hat{\sigma}_{11}$ ,  $\hat{\sigma}_{22}$  и  $\hat{\sigma}_{33}$  коммутируют друг с другом. Унитарная матрица перехода из стандартного базиса  $\{|i_1 \dots i_n\rangle \langle j_1 \dots j_n|\}$  в базис Паули содержит только элементы 0,  $\pm 1$  и  $\pm i$ . Например:

$$|00 \dots 0\rangle \langle 00 \dots 0| \rightarrow \frac{1}{2^n} \sum_{i_1, \dots, i_n \in \{0,3\}} \hat{\sigma}_{i_1 \dots i_n}.$$

Общее выражение для стандартных ортогональных проекторов имеет вид:

$$|i_1 \dots i_n\rangle \langle i_1 \dots i_n| = \frac{1}{2^n} \sum_{k_1, \dots, k_n \in \{0,3\}} \mathcal{X}_{k_1}^{i_1} \dots \mathcal{X}_{k_n}^{i_n} \hat{\sigma}_{k_1 \dots k_n},$$

где

$$\mathcal{X}_0^0 = \mathcal{X}_3^0 = \mathcal{X}_0^1 = 1, \quad \mathcal{X}_3^1 = -1.$$

Из выражения (1.6) следует, что: 1. Множество  $\{i\hat{\sigma}_K\}_{K=0}^{4^n-1}$  образует ортонормированный базис в  $\mathfrak{su}(2^n)$ . 2. Множество

$$\tilde{P}(\mathcal{H}_n) = \{\epsilon \hat{\sigma}_K \mid K \in \text{Str}_n, \epsilon \in \{\pm 1, \pm i\}\},$$

содержащее  $4^{n+1}$  элементов, образует группу — т.н.  $n$ -кубитную группу Паули.

Нормализатор группы Паули в унитарной группе:

$$\mathcal{C}(\mathcal{H}_n) = \left\{ \hat{U} \in U(\mathcal{H}_n) \mid \hat{U} \hat{\sigma}_K \hat{U}^\dagger \in \tilde{P}(\mathcal{H}_n), \quad \forall \hat{\sigma}_K \in \tilde{P}(\mathcal{H}_n) \right\},$$

называется группой Клиффорда. Исходя из (1.2), (1.4) и (1.7) получаем следующее утверждение

**Утверждение 1.** *Взаимные унитарные преобразования базисных операторов Паули подчиняются соотношениям  $\hat{\sigma}_{i_1 \dots i_n} \hat{\sigma}_{k_1 \dots k_n} \hat{\sigma}_{i_1 \dots i_n} = \pm \hat{\sigma}_{i_1 \dots i_n}$ , где знак плюс стоит тогда и только тогда, когда количество троек  $(i_m k_m i_m)_{m \in \{1, \dots, n\}}$ , удовлетворяют условиям  $i_m \neq k_m$ ,  $i_m \neq 0$ , и  $k_m \neq 0$  четности.*

$l$	0	1	2	3	4	5	6	7
$l_2 l_1 l_0$	000	001	010	011	100	101	110	111
$k_2 k_1 k_0$	011	011	011	011	011	011	011	011
$\bar{l} \wedge k$	011	010	001	000	011	010	001	000
$l \wedge k$	000	001	010	011	000	001	010	011
$l \wedge \bar{k}$	000	000	000	000	100	100	100	100
$\hat{\sigma}_I$	$\hat{\sigma}_{011}$	$\hat{\sigma}_{012}$	$\hat{\sigma}_{021}$	$\hat{\sigma}_{022}$	$\hat{\sigma}_{311}$	$\hat{\sigma}_{312}$	$\hat{\sigma}_{121}$	$\hat{\sigma}_{322}$

Таблица 2: Элементы базиса Паули, возникающие для  $k = 011$ .

## 1.2 Вариационная квантовая оптимизация

Пусть  $\mathcal{H}_n$  — гильбертово пространство квантовой системы, состоящей из  $n$  кубитов,  $\mathcal{S} \subset \mathcal{H}_n$  — пространство векторов состояния (т.е. векторов, нормированных на единицу),  $L(\mathcal{H}_n)$  — алгебра операторов на  $\mathcal{H}_n$  и  $\hat{H} \in L(\mathcal{H}_n)$  — эрмитов оператор. Определим функцию

$$E(|u\rangle) = \langle u | \hat{H} | u \rangle, \quad |u\rangle \in \mathcal{S}. \quad (1.8)$$

В простейшей постановке **квантовой задачи оптимизации** требуется найти вектор состояния, на котором целевая функция (функция стоимости)  $E$  принимает минимальное значение, т.е., в формальной

записи, решить задачу

$$E(|u\rangle) \xrightarrow{|u\rangle \in \mathcal{S}} \min. \quad (1.9)$$

Ниже, для определенности и краткости, будем называть  $\hat{H}$  гамильтонианом системы, а целевую функцию  $E$  — энергией.

Сложность алгоритмов прямого вычисления собственных значений гамильтониана  $\hat{H}$  растет экспоненциально с ростом числа кубитов, поэтому для больших систем используются вариационные методы решения задачи оптимизации (1.9).

Вариационными квантовыми алгоритмами обычно называют такие гибридные квантово-классические алгоритмы, нацеленные на решение квантовых задач оптимизации посредством квантовых вычислений или их классической имитации, которые проводят вариационную настройку параметров квантовой схемы. Параметрически управляемое квантовое устройство, обычно представленное квантовой цепью, реализует анзац, т.е. унитарное преобразование стандартного начального состояния  $|0\rangle^{\otimes n}$  или, как вариант, предудущего полученного состояния. На каждом шаге регулирующие параметры подбираются так, чтобы минимизировать энергию (целевую функцию). Обычно это выполняется путём измерения энергии состояний, предоставляемых вариационной схемой, и обновления параметров для минимизации целевой функции.

В точной математической формулировке сказанное означает, что в функции (1.8) вектор состояния  $|u\rangle$  зависит от набора  $m$  параметров  $\boldsymbol{\theta} = (\theta_1, \dots, \theta_m)$ , которые принимают значения в некоторой связной и односвязной области  $\Omega \in \mathbb{R}^m$ . Вариационная формулировка квантовой задачи оптимизации (1.9) имеет вид

$$E(|u(\boldsymbol{\theta})\rangle) \xrightarrow{\boldsymbol{\theta} \in \Omega} \min, \quad E(|u(\boldsymbol{\theta})\rangle) = \langle u(\boldsymbol{\theta}) | \hat{H} | u(\boldsymbol{\theta}) \rangle. \quad (1.10)$$

Итак, цель вариационного квантового алгоритма — найти такой набор параметров, на котором энергия достигает минимума. Число параметров  $m$  в наборе  $\boldsymbol{\theta} = (\theta_1, \dots, \theta_m)$  зависит от конкретной задачи, в частности, от числа кубитов в квантовом устройстве. Для  $n$  кубитов размерность пространства состояний,  $N = 2^n$ , растет экспоненциально с ростом числа кубитов. Поэтому вариационный квантовый алгоритм должен быть

организован и выполнен так, чтобы выполнялось условие  $m \ll N$ , поскольку в противном случае высокий класс сложности алгоритма сделает его неэффективным с практической точки зрения.

Но наиболее важным вопросом является выбор зависимости вектора состояния  $|u(\boldsymbol{\theta})\rangle$  от параметров. В вариационных квантовых алгоритмах используется *анзац* (унитарное преобразование) вида

$$|u(\boldsymbol{\theta})\rangle = \hat{U}(\boldsymbol{\theta})|u_0\rangle. \quad (1.11)$$

В общем случае форма анзаца определяет, какими будут параметры  $\boldsymbol{\theta}$  и как их можно настроить для минимизации энергии (целевой функции). Структура анзаца, как правило, будет зависеть от поставленной задачи, так как во многих случаях можно использовать информацию о проблеме, чтобы подобрать анзац: это "анзац, подсказанный задачей". Однако можно построить анзацы достаточно общего вида, которые пригодны для использования в некоторых классах задач даже тогда, когда интуиция и известная информация о задаче не позволят его уточнить. Две наиболее распространенные формы анзаца рассмотрены в следующем разделе.

### 1.3 Анзац и общая схема алгоритма

В вариационных квантовых алгоритмах анзац (1.11) стандартно выбирается в виде композиции  $m$  последовательно примененных унитарных преобразований

$$\hat{U}(\boldsymbol{\theta}) = \hat{U}_m(\theta_m) \cdots \hat{U}_1(\theta_1). \quad (1.12)$$

В композиции (1.12) выбор операторов определяется типом задачи и технической возможностью их реализации на конкретном квантовом устройстве. Например, можно выбрать

$$\hat{U}_K(\theta_K) = \hat{W}_K \exp(i\theta_K \hat{\sigma}_K) = \hat{W}_K (\cos\theta_K \hat{\sigma}_{0\dots 0} + i \sin\theta_K \hat{\sigma}_K), \quad (1.13)$$

где  $1 \leq K \leq m$ ,  $\hat{\sigma}_K = \hat{\sigma}_{k_1} \otimes \dots \otimes \hat{\sigma}_{k_n}$ ,  $k_1, \dots, k_n \in \{0, 1, 2, 3\}$ ,  $K = k_1 \dots k_n$  (т.е.  $K$  — десятичное представление строки  $k_1 \dots k_n$ , рассматриваемой как число по основанию 4),  $n$  — число кубитов, а  $\hat{W}_K$  — независимый от параметров унитарный оператор. Как правило, в строке  $k_1 \dots k_n$  только отдельные числа отличны от нуля, так что в тензорном произведении  $\hat{\sigma}_{k_1} \otimes \dots \otimes \hat{\sigma}_{k_n}$  часть операторов являются тождественными.

В другом распространенном варианте операторы в композиции (1.12) имеют вид

$$\hat{U}_K(\theta_K) = \hat{W}_K(e^{i\theta_{k_1}\hat{\sigma}_{k_1}} \otimes \dots \otimes e^{i\theta_{k_n}\hat{\sigma}_{k_n}}), \quad (1.14)$$

где по-прежнему  $1 \leq K \leq m$  и  $K = k_1 \dots k_n$ . Если в (1.13) все операторы  $\hat{W}_K$  могут быть тождественными, то в (1.14), по крайней мере некоторые операторы  $\hat{W}_K$  должны быть запутывающими и, следовательно, как минимум двухкубитными.

Таким образом, анзацы (1.12), (1.13) и (1.14) конкретизируют вариационную квантовую задачу оптимизации (1.10) и (1.11) в отношении параметрической зависимости вектора состояния,

$$|u(\boldsymbol{\theta})\rangle = \hat{U}(\boldsymbol{\theta})|0 \dots 0\rangle,$$

где начальное состояние имеет вид  $|0 \dots 0\rangle = |0\rangle^{\otimes n}$ . Если предположить далее, что мы в состоянии уверенно приготовить начальное состояние, реализовать анзац на физическом устройстве и вычислить значение энергии  $E(|u(\boldsymbol{\theta})\rangle) = \langle u(\boldsymbol{\theta}) | \hat{H} | u(\boldsymbol{\theta}) \rangle$  посредством измерений (с привлечением классического компьютера), то следующий — основной — вопрос можно сформулировать так: как искать параметры, которые обеспечивают глобальный минимум энергии. Этот этап выполняется с помощью классического компьютера, так что вариационный квантовый алгоритм — гибридный квантово-классический алгоритм: параметризованная квантовая схема и измерительный прибор представляют квантовую часть, а алгоритм настройки параметров — классическую.

Опишем кратко схему алгоритма псевдокодом (см. также Рис. 1.1).

### Алгоритм вариационной квантовой оптимизации

1. Ввод начального вектора параметров  $\boldsymbol{\theta}$ . Полагаем  $\boldsymbol{\theta}_0 = \boldsymbol{\theta}$ .
2. Генерация состояния  $|u(\boldsymbol{\theta})\rangle = \hat{U}(\boldsymbol{\theta})|0\rangle^{\otimes n}$  и случайного вектора  $\delta\boldsymbol{\theta}$ .
3. Вычисление энергии  $E(\boldsymbol{\theta}) = \langle u(\boldsymbol{\theta}) | \hat{H} | u(\boldsymbol{\theta}) \rangle$  с текущим вектором  $\boldsymbol{\theta}$ .
4. Алгоритмический переход к вектору параметров  $\boldsymbol{\theta}_1 = \boldsymbol{\theta} + \delta\boldsymbol{\theta}$ .  
Если  $E(\boldsymbol{\theta}_1) < E(\boldsymbol{\theta})$ , то положим  $\boldsymbol{\theta} = \boldsymbol{\theta}_1$  и запомним  $\boldsymbol{\theta}_0 = \boldsymbol{\theta}_1$ .
5. Условие завершения работы. Если условие выполнено, то выходим с результатом  $|u(\boldsymbol{\theta}_0)\rangle$  и  $E(\boldsymbol{\theta}_0)$ . Иначе переходим к метке 2.

Переход к новому вектору параметров  $\theta_1$  является вторым ключевым пунктом (после выбора анзаца), который характеризует различные типы вариационных квантовых алгоритмов. Условие завершения работы может быть выбрано очень многими способами. Простейшим и одновременно универсальным является условие отсутствия понижения энергии в течение определенного числа циклов работы алгоритма.

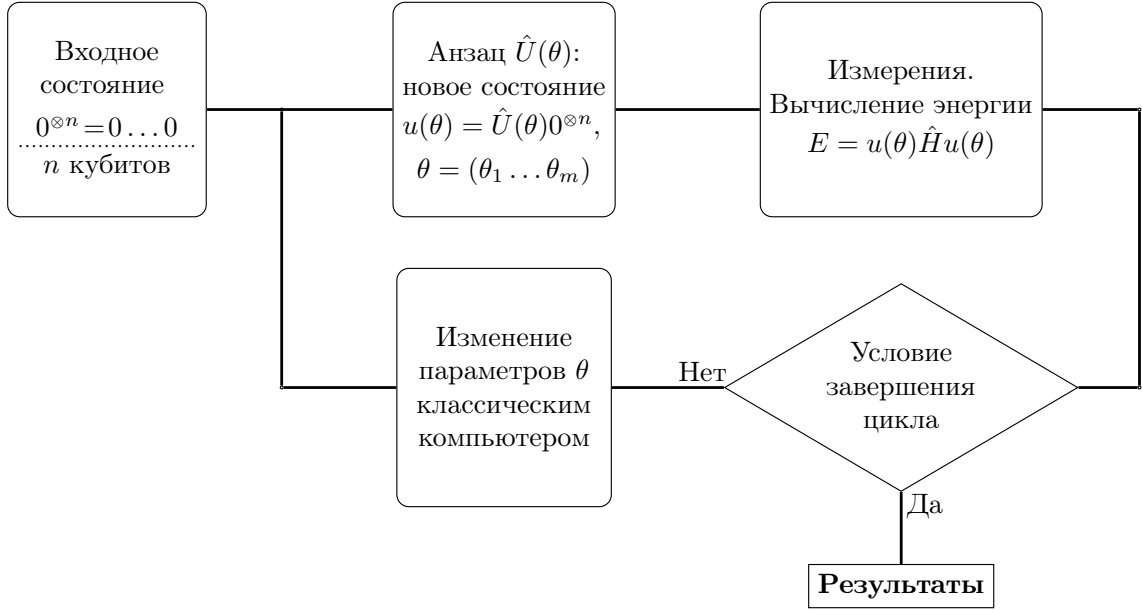


Рис. 1.1: Общая схема квантового вариационного алгоритма. Более подробную схему можно изучить на Рис. 2.1

## 1.4 Пример, иллюстрирующий особенности алгоритма

Для иллюстрации алгоритма рассмотрим гамильтониан

$$\hat{H} = 2\hat{\sigma}_{03} + \hat{\sigma}_{30} - 4\hat{\sigma}_{11}, \quad (1.15)$$

который в стандартном базисе  $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$  имеет матрицу

$$H = \begin{pmatrix} 3 & 0 & 0 & -4 \\ 0 & -1 & -4 & 0 \\ 0 & -4 & 1 & 0 \\ -4 & 0 & 0 & -3 \end{pmatrix}.$$

Используя систему Maple находим собственные значения и собственные состояния в порядке возрастания собственных значений, начиная с основного состояния  $|u_0\rangle$  с собственным значением  $E_0$ :

$$E_0 = -5, \quad |u_0\rangle = \frac{1}{\sqrt{5}} |00\rangle + \frac{2}{\sqrt{5}} |11\rangle, \quad (1.16)$$

$$E_1 = -\sqrt{17}, \quad |u_1\rangle = \frac{\sqrt{17}+1}{\sqrt{34+2\sqrt{17}}} |01\rangle + \frac{\sqrt{8}}{\sqrt{17+\sqrt{17}}} |10\rangle, \quad (1.17)$$

$$E_2 = \sqrt{17}, \quad |u_1\rangle = -\frac{\sqrt{17}-1}{\sqrt{34-2\sqrt{17}}} |01\rangle + \frac{\sqrt{8}}{\sqrt{17-\sqrt{17}}} |10\rangle, \quad (1.18)$$

$$E_3 = 5, \quad |u_3\rangle = -\frac{2}{\sqrt{5}} |00\rangle + \frac{1}{\sqrt{5}} |11\rangle. \quad (1.19)$$

Рассмотрим далее пошаговое выполнение вариационного квантового алгоритма, который позволяет найти состояние, близкое к основному.

*Первый шаг — выбор анзаца*, т.е. унитарного преобразования  $\hat{U}(\boldsymbol{\theta})$ . В гамильтониан (1.15) не входят операторы вида  $\hat{\sigma}_{k2}$  и  $\hat{\sigma}_{2k}$  с  $k \neq 2$ , поэтому имеет смысл сразу выбирать анзац так, чтобы при действии на  $k \neq 2$  он давал вектор состояния с вещественными коэффициентами. Других наводящих соображений относительно формы анзаца не видно, поэтому следует рассмотреть разные варианты. В общем случае вектор параметров  $\boldsymbol{\theta}$  четырехмерен. В простейшем варианте анзац с четырехмерным вектором параметров  $\boldsymbol{\theta} = (\xi, \lambda, \mu, \nu)$  можно выбирать как композицию экспонент

$$\hat{U}(\boldsymbol{\theta}) = e^{i\xi\hat{\sigma}_{02}} e^{i\lambda\hat{\sigma}_{03}} e^{i\mu\hat{\sigma}_{30}} e^{i\nu\hat{\sigma}_{11}} \quad (1.20)$$

операторов Паули, присутствующих в гамильтониане (1.15). Вычислим вначале

$$\begin{aligned} e^{i\mu\hat{\sigma}_{30}} e^{i\nu\hat{\sigma}_{11}} |00\rangle &= (\cos\mu \hat{\sigma}_{00} + i \sin\mu \hat{\sigma}_{30}) (\cos\nu |00\rangle + i \sin\nu |11\rangle) \\ &= \cos\mu \cos\nu |00\rangle - \sin\mu \sin\nu |11\rangle + i \sin\mu \cos\nu |00\rangle + i \cos\mu \sin\nu |11\rangle \\ &= e^{i\mu} \cos\nu |00\rangle + i e^{i\mu} \sin\nu |11\rangle = e^{i\mu} \cos\nu |00\rangle + e^{i(\mu+\pi/2)} \sin\nu |11\rangle. \end{aligned}$$

Действуя на результат оператором  $e^{i\lambda\hat{\sigma}_{03}}$ , получим следующий промежу-



точный вектор состояния:

$$\begin{aligned}
e^{i\lambda\hat{\sigma}_{03}}e^{i\mu\hat{\sigma}_{30}}e^{i\nu\hat{\sigma}_{11}}|00\rangle &= e^{i\mu}\cos\nu\left(\cos\lambda\hat{\sigma}_{00}+i\sin\lambda\hat{\sigma}_{03}\right)|00\rangle \\
&\quad + e^{i(\mu+\pi/2)}\sin\nu\left(\cos\lambda\hat{\sigma}_{00}+i\sin\lambda\hat{\sigma}_{03}\right)|11\rangle \\
&= e^{i\mu}\cos\nu\left(\cos\lambda+i\sin\lambda\right)|00\rangle + e^{i(\mu+\pi/2)}\sin\nu\left(\cos\lambda+i\sin\lambda\right)|11\rangle \\
&= e^{i(\mu+\lambda)}\cos\nu|00\rangle + e^{i(\mu+\pi/2-\lambda)}\sin\nu|11\rangle. \quad (1.21)
\end{aligned}$$

Очевидно, что этот анзац не является универсальным.

Варьируя параметры  $\lambda, \mu, \nu$ , можно получить основное состояние (1.16) с точностью до несущественного множителя  $e^{i(\mu+\pi/4)}$ , например, при

$$\lambda = \pi/4, \quad \mu \in \mathbb{R}, \quad \cos\nu = 1/\sqrt{5}, \quad \sin\nu = 2/\sqrt{5}. \quad (1.22)$$

Здесь  $\mu$  — любое, поэтому к нужному результату приводит более простой анзац (при  $\mu = 0$ )  $\hat{U}(\boldsymbol{\theta}) = e^{i\lambda\hat{\sigma}_{03}}e^{i\nu\hat{\sigma}_{11}}$ , однако заранее это нам не известно. Более того основное состояние (1.16) можно достигнуть (что заранее также неизвестно и неочевидно) даже однопараметрическим анзацем

$$e^{i\nu\hat{\sigma}_{12}}|00\rangle = \left(\cos\nu\hat{\sigma}_{00}+i\sin\nu\hat{\sigma}_{12}\right)|00\rangle = \cos\nu|00\rangle + \sin\nu|11\rangle,$$

с теми же значениями  $\cos\nu$  и  $\sin\nu$ , что и в (1.22).

Действуя на (1.21) оператором  $e^{i\xi\hat{\sigma}_{02}}$ , получим вектор состояния

$$\begin{aligned}
|\Phi\rangle &= \hat{U}(\boldsymbol{\theta})|00\rangle \\
&= \left(\cos\xi\hat{\sigma}_{00}+i\sin\xi\hat{\sigma}_{02}\right)\left(e^{i(\mu+\lambda)}\cos\nu|00\rangle + e^{i(\mu+\pi/2-\lambda)}\sin\nu|11\rangle\right) \\
&= e^{i(\mu+\lambda)}\sin\xi\cos\nu|01\rangle - e^{i(\mu+\pi/2-\lambda)}\sin\xi\sin\nu|10\rangle \\
&\quad + e^{i(\mu+\lambda)}\cos\xi\cos\nu|00\rangle + e^{i(\mu+\pi/2-\lambda)}\cos\xi\sin\nu|11\rangle, \quad (1.23)
\end{aligned}$$

который зависит от четырех параметров. Из формы данного вектора видно, что анзац (1.20) универсален (с учетом замечания о вещественности коэффициентов, сделанного выше). Основное состояние достигается при произвольном  $\mu \in \mathbb{R}$  и

$$\xi = 0, \quad \lambda = \{\pi/4, 7\pi/4\}, \quad \cos\nu = 1/\sqrt{5}, \quad \sin\nu = \{2/\sqrt{5}, -2/\sqrt{5}\} \quad (1.24)$$

или

$$\xi = \pi, \lambda = \{\pi/4, 7\pi/4\}, \cos\nu = -1/\sqrt{5}, \sin\nu = \{-2/\sqrt{5}, 2/\sqrt{5}\}. \quad (1.25)$$

Для сокращения записи имеет смысл освободиться в (1.23) от фазового множителя и записать вектор состояния в виде

$$\begin{aligned} |\Phi\rangle = \sin\xi \big( \cos\nu |01\rangle - e^{i(\pi/2-2\lambda)} \sin\nu |10\rangle \big) \\ + \cos\xi \big( \cos\nu |00\rangle + e^{i(\pi/2-2\lambda)} \sin\nu |11\rangle \big) \end{aligned} \quad (1.26)$$

Второй шаг — вычисление энергии состояния, т.е. среднего значения  $\langle\Phi|\hat{H}|\Phi\rangle$ . Заметим, что первый и второй шаги должны выполняться на квантовых устройствах, а при классической симуляции алгоритма необходимо проводить явные вычисления. Из (1.15) и (1.26) находим

$$\begin{aligned} \hat{H}|\Phi\rangle = \sin\xi \big( 2\hat{\sigma}_{03} + \hat{\sigma}_{30} - 4\hat{\sigma}_{11} \big) \big( \cos\nu |01\rangle - e^{i(\pi/2-2\lambda)} \sin\nu |10\rangle \big) \\ + \cos\xi \big( 2\hat{\sigma}_{03} + \hat{\sigma}_{30} - 4\hat{\sigma}_{11} \big) \big( \cos\nu |00\rangle + e^{i(\pi/2-2\lambda)} \sin\nu |11\rangle \big) \\ = \sin\xi \left\{ \left( 4e^{i(\pi/2-2\lambda)} \sin\nu - \cos\nu \right) |01\rangle \right. \\ \left. - \left( e^{i(\pi/2-2\lambda)} \sin\nu + 4\cos\nu \right) |10\rangle \right\} \\ + \cos\xi \left\{ \left( 3\cos\nu - 4e^{i(\pi/2-2\lambda)} \sin\nu \right) |00\rangle \right. \\ \left. - \left( 4\cos\nu + 3e^{i(\pi/2-2\lambda)} \sin\nu \right) |11\rangle \right\}. \end{aligned}$$

Поскольку

$$\begin{aligned} \langle\Phi| = \sin\xi \big( \cos\nu \langle 01| - e^{-i(\pi/2-2\lambda)} \sin\nu \langle 10| \big) \\ + \cos\xi \big( \cos\nu \langle 00| + e^{-i(\pi/2-2\lambda)} \sin\nu \langle 11| \big), \end{aligned}$$

то

$$\begin{aligned}
E_{\Phi} &= \langle \Phi | \hat{H} | \Phi \rangle \\
&= \sin^2 \xi \left\{ \cos \nu (4e^{i(\pi/2-2\lambda)} \sin \nu - \cos \nu) \right. \\
&\quad \left. + \sin \nu (\sin \nu + 4e^{-i(\pi/2-2\lambda)} \cos \nu) \right\} \\
&\quad + \cos^2 \xi \left\{ \cos \nu (3 \cos \nu - 4e^{i(\pi/2-2\lambda)} \sin \nu) \right. \\
&\quad \left. - \sin \nu (4e^{-i(\pi/2-2\lambda)} \cos \nu + 3 \sin \nu) \right\} \\
&= \sin^2 \xi (4 \sin 2\lambda \sin 2\nu - \cos 2\nu) + \cos^2 \xi (3 \cos 2\nu - 4 \sin 2\lambda \sin 2\nu). \quad (1.27)
\end{aligned}$$

Разумеется, если взять значения  $\xi$ ,  $\lambda$ ,  $\cos \nu$ ,  $\sin \nu$  как в (1.24) или в (1.25), то мы получим энергию основного состояния (1.16), т.е.  $E_{\Phi} = -5$ .

*Третий шаг — изменение значений параметров  $\lambda, \mu, \nu$  (с целью минимизации  $E_{\Phi}$ ) и возвращение к первому шагу; предполагается, что в начале выполнения алгоритма начальные значения параметров заданы.* Из (1.27) видно, что на значение  $E_{\Phi}$  параметр  $\mu$  не влияет, а параметры  $\lambda, \nu$  должны варьироваться в области  $[0, \pi] \times [0, \pi]$ . Однако изначально это неизвестно, поэтому все четыре параметра должны варьироваться в области  $[0, 2\pi] \times [0, 2\pi] \times [0, 2\pi] \times [0, 2\pi]$ . Имеет смысл установить независимость энергии от параметра  $\mu$  (и ее зависимость от остальных параметров) в начале работы алгоритма.

Мы уже знаем, что глобальный минимум энергии достигается для четырех наборов параметров (1.24) и (1.25). Соответствующие собственные векторы, вычисленные по выражению (1.26) отличаются от (1.16) только фазовыми множителями. Теперь необходимо выяснить, имеются ли у функции (трех переменных) (1.27) другие локальные минимумы.

Используя систему Maple, вычислим производные

$$\begin{aligned}
&\partial_{\xi} E_{\Phi}, \quad \partial_{\lambda} E_{\Phi}, \quad \partial_{\nu} E_{\Phi}, \\
&A = \partial_{\xi}^2 E_{\Phi}, \quad B = \partial_{\lambda}^2 E_{\Phi}, \quad C = \partial_{\nu}^2 E_{\Phi}, \\
&K = \partial_{\xi\lambda} E_{\Phi}, \quad L = \partial_{\xi\nu} E_{\Phi}, \quad M = \partial_{\lambda\nu} E_{\Phi}.
\end{aligned}$$

Находим

$$\begin{aligned}
\partial_\xi E_\Phi &= 4 \sin 2\xi (2 \sin 2\lambda \sin 2\nu - \cos 2\nu), \\
\partial_\lambda E_\Phi &= -8 \cos 2\xi \cos 2\lambda \sin 2\nu, \\
\partial_\nu E_\Phi &= \sin^2 \xi (8 \sin 2\lambda \cos 2\nu + 2 \sin 2\nu) - \cos^2 \xi (6 \sin 2\nu + 8 \sin 2\lambda \cos 2\nu), \\
A &= 8 \cos 2\xi (2 \sin 2\lambda \sin 2\nu - \cos 2\nu), \\
B &= 16 \cos 2\xi \sin 2\lambda \sin 2\nu, \\
C &= 4 \cos^2 \xi (4 \sin 2\lambda \sin 2\nu - 3 \cos 2\nu) - 4 \sin^2 \xi (4 \sin 2\lambda \sin 2\nu - \cos 2\nu), \\
K &= 16 \sin 2\xi \cos 2\lambda \sin 2\nu, \\
L &= 4 \sin 2\xi (4 \sin 2\lambda \cos 2\nu + 2 \sin 2\nu), \\
M &= -16 \cos 2\xi \cos 2\lambda \cos 2\nu.
\end{aligned}$$

Необходимые и достаточные условия минимума имеют вид

$$\begin{aligned}
&\partial_\xi E_\Phi = 0, \quad \partial_\lambda E_\Phi = 0, \quad \partial_\nu E_\Phi = 0, \\
&A > 0, \quad \det \begin{pmatrix} A & K \\ K & B \end{pmatrix} > 0, \quad \det \begin{pmatrix} A & K & L \\ K & B & M \\ L & M & C \end{pmatrix} > 0.
\end{aligned}$$

Снова проводя вычисления с помощью системы Maple, обнаруживаем четыре точки локального минимума с энергией  $E_\Phi = -\sqrt{17}$ :

$$\begin{aligned}
\xi &= \frac{\pi}{2}, & \lambda &= \frac{\pi}{4}, & \nu &= \pi - \frac{1}{2} \arctan(4), \\
\xi &= \frac{\pi}{2}, & \lambda &= \frac{\pi}{4}, & \nu &= 2\pi - \frac{1}{2} \arctan(4), \\
\xi &= \frac{\pi}{2}, & \lambda &= \frac{3\pi}{4}, & \nu &= \frac{1}{2} \arctan(4), \\
\xi &= \frac{\pi}{2}, & \lambda &= \frac{3\pi}{4}, & \nu &= \pi + \frac{1}{2} \arctan(4).
\end{aligned}$$

Таким образом, в процессе оптимизации целевой функции должны использоваться методы, которые позволяют избежать попадания в точку локального минимума, например, метод отжига.

## Глава 2

# Вариационный квантовый алгоритм на основе метода отжига

### 2.1 Метод отжига

Метод отжига или, подробнее, метод имитации отжига относится к семейству методов Монте-Карло, разработанных для поиска *глобального* минимума целевой функции. Этот метод обладает одним существенным преимуществом по сравнению с аналитическими методами оптимизации: он применим к целевым функциям произвольной природы. Особенно важно то, что метод отжига полностью сохраняет свою эффективность в задачах, где целевая функция не дифференцируема и при этом имеет большое число локальных минимумов. По этой причине, метод отжига, как фундаментальная концепция в теории глобальной оптимизации, широко используется в квантовых вычислениях, особенно в вариационных квантовых алгоритмах (в их квантовой реализации, а не классической эмуляции), где вычисление целевой функции не обладает достаточной точностью. Основная идея метода заключается в постепенном снижении "температуры" системы, чтобы достичь состояния с минимальным значением целевой функции (далее, для краткости, энергии). В этом разделе подробно рассматривается классический вариант отжига и детали его практического применения.

Классический метод отжига основывается на аналогии с физическим процессом термического отжига, при котором материал медленно охлаждается, чтобы избежать образования дефектов и достичь состояния минимальной энергии. Математическое обоснование метода связано с распределением Больцмана, которое описывает вероятность состояния системы при заданной температуре  $T$  формулой

$$p(x) = \frac{1}{Z(T)} \exp \left( -\frac{E(x)}{k_B T} \right),$$

в которой  $E(x)$  — энергия состояния  $x$ ,  $k_B$  — постоянная Больцмана, а  $Z(T)$  — статистическая сумма. Процесс отжига является дискретным и моделирует систему, которая на каждом шаге может переходить между состояниями  $x$  и  $y$  с вероятностью, которая зависит от разности энергий  $\Delta E = E(y) - E(x)$ :

$$p(x \rightarrow y) = \min \left( 1, \exp \left( -\frac{\Delta E}{k_B T} \right) \right). \quad (2.1)$$

С вероятностью  $1 - p(x \rightarrow y)$  состояние системы не изменяется.

В самом общем случае в абстрактную математическую постановку задачи оптимизации методом отжига входят следующие данные.

### Список данных

1. Множество состояний  $C$ .
2. Энергия (целевая функция)  $E : C \rightarrow \mathbf{R}$ .
3. Семейство  $Rnd = \{R_s\}_{s \in J}$  отображений  $R_s : C \rightarrow C$ , где множество индексов  $J$ , вообще говоря, не счетно.
4. Начальная температура  $T$ , минимальная температура  $T_{min}$  и закон ее понижения  $T_k \rightarrow T_{k+1} = F(T_k, k)$ , где  $k$  — шаг процесса отжига.

Практическая реализация имитации отжига обычно осуществляется посредством последовательности конечного числа шагов — случайных испытаний. Каждое испытание состоит, во-первых, из случайного выбора некоторого отображения  $R_s \in Rnd$ , задающего переход  $x \rightarrow y = R_s(x)$ , т.е. этот переход выбирается случайным образом из некоторого семейства

возможностей. В зависимости от типа задачи вероятностное распределение (вероятностная мера) на  $Rnd$  выбирается в достаточно широком диапазоне (нормальное распределение, распределение Коши, равномерное распределение и т.д.). Такое распределение, в свою очередь, может зависеть от температуры.

Во-вторых, с помощью генератора псевдослучайных чисел выбирается значение  $\varepsilon$  равномерно распределенной на интервале  $(0, 1)$  случайной величины. Если выполнено условие

$$\varepsilon < \exp \frac{E(x) - E(y)}{k_B T},$$

то переход  $x \rightarrow y$  принимается даже если энергия нового состояния,  $E(y)$ , больше, чем энергия  $E(x)$  предыдущего состояния. В противном случае, когда  $E(y) \leq E(x)$ , данное условие выполняется. Такой способ перехода предохраняет от попадания в локальный минимум, в особенности, если процедуру отжига повторять несколько раз. Наконец, в третьих, на каждом шаге производится понижение температуры в соответствии с некоторым правилом. Постепенное уменьшение температуры приводит к уменьшению вероятности перехода в состояния с более высокой энергией, в то время как система стремится к состоянию глобального минимума энергии.

## 2.2 Алгоритм

### Краткое описание алгоритма

В задачах квантовой оптимизации в системе из  $n$  кубитов множество состояний  $C$ , входящее в *Список данных* из предыдущего раздела 2.1, является подмножеством в пространстве состояний системы  $\mathcal{H}_n$ . Оно возникает в результате применения параметризованного анзаца  $\hat{U}(\boldsymbol{\theta})$  к начальному состоянию  $|0 \dots 0\rangle = |0\rangle^{\otimes n}$ , так что  $|u(\boldsymbol{\theta})\rangle = \hat{U}(\boldsymbol{\theta})|0\rangle^{\otimes n}$ , где

$$\boldsymbol{\theta} = (\theta_1, \dots, \theta_m) \in \Omega = [0, 2\pi]^m.$$

Таким образом,  $|u(\boldsymbol{\theta})\rangle \in C$  — это функция на пространстве параметров  $\Omega$  со значениями в  $\mathcal{H}_n$ ; при этом  $C$  не является подпространством в  $\mathcal{H}_n$ . Энергия (целевая функция на  $\Omega$ ) из *Списка данных*, п. 2, определяется выражением  $E(\boldsymbol{\theta}) = \langle u(\boldsymbol{\theta}) | \hat{H} | u(\boldsymbol{\theta}) \rangle$ .

В предлагаемом алгоритме выбор отображения из семейства  $Rnd$  (*Список данных, п. 3*) равновероятен в отношении любого отображения в следующем смысле. С помощью генератора псевдослучайных чисел выбирается  $m$  значений  $\varepsilon_i \in (0, 1)$ ,  $i = 1, \dots, m$  случайной величины, равномерно распределенной на данном интервале, а затем формируется случайный вектор

$$\delta\theta = (\delta\theta_1, \dots, \delta\theta_m) \in \Omega, \quad \delta\theta_i = \varepsilon_i \pi.$$

Отображение  $R_{\delta\theta}: C \rightarrow C$  определяется формулой  $|u(\theta)\rangle \mapsto |u(\theta + \delta\theta)\rangle$ , где вектор  $\theta + \delta\theta$  берется по модулю  $2\pi$ . Далее производим понижение температуры (*Список данных, п. 4*) по простейшей схеме равномерного уменьшения ее значения на каждом шаге процесса отжига  $T_{k+1} = T_k - \tau$ , где выбор значения  $\tau$  производится на основе численных экспериментов с конкретной задачей.

Краткая формулировка алгоритма на уровне псевдокода выглядит следующим образом.

### **Вариационная квантовая оптимизация методом отжига**

1. Вводим начальный вектор  $\theta \in \Omega$ , минимальную температуру  $T_{min}$ , текущую температуру  $T > T_{min}$  и значение  $\tau$ . Положим  $\theta_0 = \theta$ .
- 2.** Генерируем состояние  $|u(\theta)\rangle$  и вычисляем энергию  $E(\theta)$  при текущей температуре  $T$ .
3. Генерируем случайный вектор  $\delta\theta \in \Omega$ ,  $\theta_1 = \theta + \delta\theta \pmod{2\pi}$ . Вычисляем энергию  $E(\theta_1)$ .
4. Если  $E(\theta_1) \leq E(\theta)$ , то полагаем  $\theta = \theta_1$  и  $\theta_0 = \theta_1$ . Иначе генерируем  $\varepsilon \in (0, 1)$  и если  $\varepsilon < \exp\{[E(\theta) - E(\theta_1)]/T\}$ , то полагаем  $\theta = \theta_1$ .
5. Понижаем температуру:  $T \mapsto T - \tau$ .
6. Если  $T > T_{min}$ , то переходим к метке **2**. Иначе выходим с результатом  $|u(\theta_0)\rangle$  и  $E(\theta_0)$ .

Теперь рассмотрим программную реализацию вариационного квантового алгоритма с оптимизацией методом отжига.

### **Полное описание алгоритма. Общая структура программы**

Разработанная программа реализует вариационный квантовый алгоритм с классическим оптимизатором на основе метода имитации отжи-



га (simulated annealing). На вход подаётся гамильтониан в базисе Паули, после чего осуществляется поиск минимума энергии в пространстве параметризованных квантовых состояний, генерируемых вариационным анзацем. Программа построена модульно и позволяет последовательно:

1. Прочитать описание гамильтониана из текстового файла;
2. Построить вариационный анзац в виде произведения экспонент Паули-операторов;
3. Численно вычислить энергию состояния  $E(\boldsymbol{\theta})$ ;
4. Найти минимум энергии методом отжига;
5. Вывести подробную информацию о найденном состоянии и параметрах анзаца.

## Описание гамильтониана

Гамильтониан  $\hat{H}$  вводится в виде линейной комбинации операторов Паули:

$$\hat{H} = \sum_{k=1}^K c_k \bigotimes_{j=1}^n \sigma_{k_j}^{(j)} \quad (2.2)$$

где  $c_k \in \mathbb{C}$  — коэффициенты,  $\sigma_{k_j} \in \{I, X, Y, Z\}$  — операторы Паули для  $j$ -го кубита.

В файле `hamiltonian_operators.txt` каждая строка имеет вид:

[действительная часть] [мнимая часть] [строка Паули]

Например, строка

```
1 2.0 0.0 03
```

задаёт слагаемое  $2.0 \cdot \sigma_0 \otimes \sigma_3 = 2I \otimes Z$ .

## Загрузка гамильтониана

Для загрузки и парсинга гамильтониана используется функция:

```

1 def read_hamiltonian_data(file_path):
2     lines = read_file_lines(file_path, ignore_comments=False)
3     pauli_operators = []
4     pauli_strings = []
5     for line in lines:
6         parts = line.strip().split()
7         if len(parts) == 3:
8             real_part, imag_part, index_str = (
9                 float(parts[0]),
10                float(parts[1]),
11                str(parts[2]),
12            )
13            coefficient = np.complex128(real_part + imag_part * 1j)
14            index_list = [int(c) for c in index_str]
15            if coefficient != 0:
16                pauli_operators.append((coefficient, index_list))
17                pauli_strings.append(index_list)
18    return pauli_operators, pauli_strings

```

## Вариационный анзац в базисе Паули

Анзац реализован в виде произведения операторов Паули, параметризованных углами  $\theta$ :

$$\hat{U}(\theta) = \prod_{j=1}^m \exp \left( i\theta_j \cdot \|c_j\| \cdot \hat{\sigma}_{K_j} \right) \quad (2.3)$$

где  $\hat{\sigma}_{K_j}$  — базисные операторы Паули, входящие в гамильтониан, а  $c_j$  — соответствующие коэффициенты.

Для разложения экспоненты используется формула Эйлера:

$$\exp(i\alpha\hat{\sigma}) = \cos \alpha \cdot I + i \sin \alpha \cdot \hat{\sigma} \quad (2.4)$$

Реализация разложения анзаца по базису Паули:

```

1 def calculate_ansatz(
2     theta: np.ndarray, pauli_operators: List[Tuple[complex, List[int]]]
3 ) -> Tuple[Dict[Tuple[int, ...], complex], str, str]:
4     operator_length = len(pauli_operators[0][1])
5     result = {tuple([0] * operator_length): 1.0}
6     for t, (coeff, op) in zip(theta, pauli_operators):
7         angle = t * abs(coeff)
8         cos_t = np.cos(angle)
9         sin_t = np.sin(angle)
10        new_result = {}

```

```

11     op_tuple = tuple(op)
12     for existing_op, existing_coeff in result.items():
13         new_result[existing_op] = (
14             new_result.get(existing_op, 0) + existing_coeff * cos_t
15         )
16         compose_coeff, compose_op = pauli_compose(existing_op,
17             op_tuple)
18         final_coeff = existing_coeff * 1j * sin_t * compose_coeff
19         new_result[compose_op] = new_result.get(compose_op, 0) +
20             final_coeff
21     result = new_result
22     symbolic_str, numeric_str = format_ansatz(pauli_operators, result)
23     return result, symbolic_str, numeric_str

```

## Алгебра Паули и композиция операторов

Композиция двух операторов Паули  $\hat{\sigma}_K, \hat{\sigma}_L$  реализуется покубитно согласно таблице умножения:

$$\hat{\sigma}_i \hat{\sigma}_j = \begin{cases} \hat{\sigma}_0 & \text{если } i = j \\ \hat{\sigma}_j & \text{если } i = 0 \\ \hat{\sigma}_i & \text{если } j = 0 \\ \pm i \hat{\sigma}_k & \text{иначе} \end{cases} \quad (2.5)$$

где  $i, j, k \in \{0, 1, 2, 3\}$ ,  $\hat{\sigma}_0 = I$ .

Программная реализация:

```

1 def multiply_pauli(i: int, j: int) -> Tuple[complex, int]:
2     if i == j:
3         return (1, 0)
4     if i == 0:
5         return (1, j)
6     if j == 0:
7         return (1, i)
8     return PAULI_MAP.get((i, j), (1, 0))
9
10 def pauli_compose(s1: tuple, s2: tuple) -> Tuple[complex, tuple]:
11     coefficient = 1.0
12     result = []
13     for a, b in zip(s1, s2):
14         coeff, idx = multiply_pauli(a, b)
15         coefficient *= coeff
16         result.append(idx)
17     return coefficient, tuple(result)

```

## Вычисление энергии состояния

Энергия состояния для текущего набора параметров  $\theta$  вычисляется как математическое ожидание гамильтониана в состоянии  $|u(\theta)\rangle$ :

$$E(\theta) = \langle 0 | \hat{U}^\dagger(\theta) \hat{H} \hat{U}(\theta) | 0 \rangle \quad (2.6)$$

Реализация последовательной композиции операторов  $U^\dagger H U$  и вычисления среднего значения:

```
1 def compute_uhu(  
2     u_dict: Dict[Tuple[int, ...], complex], h_terms:  
3     List[Tuple[complex, List[int]]]  
4 ) -> Dict[Tuple[int, ...], complex]:  
5     uhu_dict = {}  
6     for coeff_h, op_h in h_terms:  
7         op_h_tuple = tuple(op_h)  
8         for j_op, j_coeff in u_dict.items():  
9             conj_j_coeff = np.conj(j_coeff)  
10            c1, op_uh = pauli_compose(j_op, op_h_tuple)  
11            for k_op, k_coeff in u_dict.items():  
12                c2, op_uhu = pauli_compose(op_uh, k_op)  
13                total_coeff = conj_j_coeff * k_coeff * coeff_h * c1 * c2  
14                uhu_dict[op_uhu] = uhu_dict.get(op_uhu, 0) + total_coeff  
15    return uhu_dict  
16  
17 def calculate_expectation(uhu_dict: Dict[Tuple[int, ...], complex]) ->  
18 float:  
19     expectation = 0.0  
20     for op, coeff in uhu_dict.items():  
21         if all(p in (0, 3) for p in op):  
22             expectation += coeff.real  
23     return expectation
```

Вклад в среднее значение дают только те операторы Паули, которые не изменяют состояние  $|0 \dots 0\rangle$ , то есть состоящие только из  $I$  и  $Z$  на каждом кубите.

## Метод имитации отжига (Simulated Annealing)

Поиск глобального минимума осуществляется методом имитации отжига, который повторяет физический процесс медленного охлаждения системы. На каждом шаге генерируется новое состояние  $\theta' = \theta + \delta\theta$  с малым случайным возмущением, и принимается по правилу Метрополиса:

$$P_{\text{accept}} = \begin{cases} 1, & \text{если } E(\boldsymbol{\theta}') < E(\boldsymbol{\theta}) \\ \exp\left(-\frac{E(\boldsymbol{\theta}') - E(\boldsymbol{\theta})}{T}\right), & \text{иначе} \end{cases} \quad (2.7)$$

Температура  $T$  постепенно понижается по закону  $T_{k+1} = \alpha T_k$ , где  $0 < \alpha < 1$ .

$$T_{k+1} = \alpha T_k \quad (2.8)$$

Реализация основного цикла отжига:

```

1 def simulated_annealing(
2     initial_theta: np.ndarray,
3     pauli_operators: List[Any],
4     progress: Any,
5     task: Any,
6     initial_temp: float = 1000.0,
7     cooling_rate: float = 0.99,
8     min_temp: float = 1e-5,
9     num_iterations_per_temp: int = 500,
10    step_size: float = 0.5,
11 ) -> Tuple[np.ndarray, float]:
12     current_theta = initial_theta.copy()
13     best_theta = current_theta.copy()
14     best_energy = float("inf")
15     rng = np.random.default_rng()
16     temp = initial_temp
17     thermalization_steps = int(num_iterations_per_temp * 0.2)
18     while temp > min_temp:
19         # Термализация
20         for _ in range(thermalization_steps):
21             neighbor_theta = generate_neighbor_theta(current_theta,
22                                                         step_size)
23             ansatz_dict, _, _ = calculate_ansatz(neighbor_theta,
24                                                  pauli_operators)
25             uhu_dict = compute_uhu(ansatz_dict, pauli_operators)
26             current_energy = calculate_expectation(uhu_dict)
27             if current_energy < best_energy:
28                 best_theta = neighbor_theta.copy()
29                 best_energy = current_energy
30                 current_theta = neighbor_theta.copy()
31             if progress is not None:
32                 progress.update(task, advance=1)
33         # Основной цикл
34         for _ in range(num_iterations_per_temp):
35             perturbation = rng.normal(0, step_size * (temp /
36                                                         initial_temp), current_theta.shape)

```

```

34     neighbor_theta = (current_theta + perturbation) % (2 *
35         np.pi)
36     ansatz_dict, _, _ = calculate_ansatz(neighbor_theta,
37         pauli_operators)
38     uhu_dict = compute_uhu(ansatz_dict, pauli_operators)
39     current_energy = calculate_expectation(uhu_dict)
40     energy_diff = current_energy - best_energy
41     if energy_diff < 0 or rng.random() < np.exp(-energy_diff /
42         temp):
43         current_theta = neighbor_theta.copy()
44         if current_energy < best_energy:
45             best_theta = current_theta.copy()
46             best_energy = current_energy
47         if progress is not None:
48             progress.update(task, advance=1)
49     temp *= cooling_rate
50     return best_theta, best_energy

```

## Визуализация и логирование результата

Для удобства анализа и отладки программа реализует вывод промежуточных и итоговых результатов в консоль в виде таблиц, панелей и прогресс-баров с помощью библиотеки `rich`, а также ведёт лог-файл всего вывода. В частности, реализованы функции:

```

1 def print_hamiltonian(console, pauli_operators): ...
2 def print_pauli_table(console, pauli_operators): ...
3 def print_composition_table(console, pauli_compose, pauli_strings): ...

```

Итоговое значение энергии и параметры оптимального анзаца выводятся в символьном и численном виде.

## 2.3 Сравнительные результаты тестирования

Для тестирования алгоритма на системе из четырех кубитов будет использоваться гамильтониан

$$\hat{H}_{OH} = 0.501 \hat{\sigma}_{1230} - 0.501 \hat{\sigma}_{2103} - 1.252 \hat{\sigma}_{0330} - 1.453 \hat{\sigma}_{2323} + 1.700 \hat{\sigma}_{1010} + 0.223 \hat{\sigma}_{1313}, \quad (2.9)$$

взятый из работы [6], в которой изучается спектроскопия основного состояния гидроксильной группы  $\text{OH}^-$ ; гамильтониан переписан в базисе

Паули и редуцирован по нулевому уровню энергии (т.е. учтена только бесследовая часть оператора). Сравнительное тестирование включает в себя две задачи: во-первых, сравнительное тестирование двух методов Монте-Карло выбора нового набора параметров, а именно, метода отжига и метода случайного поиска; во-вторых, сравнительное тестирование анзацев — универсального анзаца, построенного на основе базисных операторов Паули из  $\hat{H}_{OH}$ , и нескольких "угаданных" пробных анзацев.

Универсальный анзац имеет вид

$$\hat{U}(\boldsymbol{\theta}) = e^{i\theta_4 \hat{\sigma}_{0330}} e^{i\theta_3 \hat{\sigma}_{1313}} e^{i\theta_2 \hat{\sigma}_{2103}} e^{i\theta_1 \hat{\sigma}_{1230}}, \quad (2.10)$$

где  $\theta_k \in [0, 2\pi)$ ,  $k = 1, 2, 3, 4$ . Он записан на основе операторов Паули из гамильтониана (2.9) за исключением операторов  $\hat{\sigma}_{1100}$   $\hat{\sigma}_{2323}$ , которые дублируются оператором  $\hat{\sigma}_{1313}$ . Последний выбран только потому, что он входит в гамильтониан с минимальным коэффициентом.

### Сравнительный анализ параметров отжига

Для оценки эффективности вариационного квантового алгоритма с оптимизацией методом имитационного отжига были проведены серии тестов с различными наборами параметров отжига. В качестве эталонного значения использовалась энергия основного состояния, равная  $-3.601$ . В каждом тесте варьировались следующие параметры: начальная температура (`initial_temp`), множитель охлаждения (`cooling_rate`), минимальная температура (`min_temp`), число итераций на каждой температуре (`num_iterations_per_temp`), а также стандартное отклонение для шума параметров  $\theta$  (`step_size`).

Полученные результаты позволяют выделить основные закономерности:

- При фиксированных начальной температуре и числе итераций на температуре уменьшение шага `step_size` приводит к ухудшению результата: алгоритм хуже исследует пространство параметров и чаще застревает в локальных минимумах. Наилучшие значения энергии наблюдались при `step_size` = 0.1.
- Увеличение числа итераций на каждой температуре не приводит к принципиальному улучшению результата, хотя небольшое положительное влияние имеет место.

- Повышение начальной температуры (например, до 1000) также не даёт существенного выигрыша, указывая на то, что роль этого параметра в пределах выбранных диапазонов невелика.
- Во всех тестах достигнутое значение энергии оставалось выше эталонного примерно на 0.20 – 0.24 единицы, что может быть связано с ограничениями самого метода или особенностями выбора анзаца.

Таким образом, можно сделать вывод, что параметры *step\_size* и *num\_iterations\_per\_temp* оказывают наибольшее влияние на итоговую энергию, однако даже при их оптимизации достичь эталонного значения не удалось. Это свидетельствует о необходимости дальнейшей доработки схемы оптимизации: возможно, стоит рассмотреть более сложные схемы охлаждения, адаптивное изменение шага или комбинированные алгоритмы поиска. Полученные результаты подчеркивают важность тонкой настройки параметров имитационного отжига для задач вариационной квантовой оптимизации.



# Заключение

В работе получены следующие основные результаты:

1. Изучены теоретические основы вариационных квантовых алгоритмов, включая формализм базиса Паули и построение анзаца для многокубитных систем.
2. Реализован и протестирован алгоритм вариационной квантовой оптимизации с использованием метода имитации отжига для поиска глобального минимума энергии гамильтониана.
3. Проведён сравнительный анализ эффективности различных параметров отжига и показано влияние выбора анзаца и схемы оптимизации на точность и сходимость алгоритма.
4. Разработана и описана программная реализация алгоритма на языке Python, позволяющая на практике моделировать процесс вариационной оптимизации для квантовых систем.

# Литература

- [1] V. V. Nikonov, A. N. Tsirulev. *Pauli basis formalism in quantum computations*. Volume 8, No 3, pp. 1 – 14, 2020.  
([doi:10.26456/mmg/2020-831](https://doi.org/10.26456/mmg/2020-831))
- [2] J. Preskill. *Quantum Computing in the NISQ era and beyond*. Quantum, vol. 2, p. 79, 2018.  
([quantum-journal:q-2018-08-06-79](https://arxiv.org/abs/quantum-journal:q-2018-08-06-79))
- [3] M. Cerezo, et al. *Variational Quantum Algorithms*. Nature Reviews Physics, vol. 3, pp. 625-644, 2021.  
([nature:42254-021-00348-9](https://arxiv.org/abs/nature:42254-021-00348-9))
- [4] A. Peruzzo, et al. *A variational eigenvalue solver on a photonic quantum processor*. Nature Communications, vol. 5, p. 4213, 2014.  
([nature:ncomms5213](https://arxiv.org/abs/nature:ncomms5213))
- [5] E. Farhi, J. Goldstone, and S. Gutmann. *A Quantum Approximate Optimization Algorithm*. arXiv preprint arXiv:1411.4028, 2014.  
([arXiv:1411.4028](https://arxiv.org/abs/1411.4028))
- [6] N. Cawley, Z. Howard, M. Kleinert et al. *Analytical study of level crossings in the Stark-Zeeman spectrum of ground state OH*. Eur. Phys. J. D, vol. 67, 233, 2013. – M. Bhattacharya. S. Marin, M. Kleinert. *Coherent cancellation of geometric phase for the OH molecule in external fields*, 2014. (<https://doi.org/10.48550/arXiv.1404.6285>)
- [7] A. Kandala, et al. *Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets*. Nature, vol. 549, pp. 242-246, 2017. ([nature:nature23879](https://arxiv.org/abs/nature:nature23879))

- [8] A. W. Harrow, A. Hassidim, and S. Lloyd. *Quantum algorithm for linear systems of equations*. Physical Review Letters, vol. 103, no. 15, p. 150502, 2009.  
([aps:PhysRevLett.103.150502](#))
- [9] J. Biamonte, et al. *Quantum machine learning*. Nature, vol. 549, pp. 195-202, 2017.  
([nature:nature23474](#))
- [10] A. A. Lopatin. *Квантовая механика и её приложения*. Санкт-Петербургский Государственный Университет.  
([math.spbu:user/gran/sb1/lopatin](#))
- [11] A. Aspuru-Guzik, A. D. Dutoi, P. J. Love, M. Head-Gordon. *Simulated Quantum Computation of Molecular Energies*. Science, vol. 309, no. 5741, pp. 1704-1707, 2005.  
([science:1113479](#))
- [12] M. Schuld, I. Sinayskiy, F. Petruccione. *An introduction to quantum machine learning*. Contemporary Physics, vol. 56, no. 2, pp. 172-185, 2015.  
([tandfonline:00107514.2014.964942](#))
- [13] A. Daskin, S. Kais. *Decomposition of unitary matrices for finding quantum circuits: Application to molecular Hamiltonians*. The Journal of Chemical Physics, vol. 141, no. 23, p. 234115, 2014.  
([aip:1.4904315](#))
- [14] J. Romero, R. Babbush, J. R. McClean, C. Hempel, P. J. Love, A. Aspuru-Guzik. *Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz*. Quantum Science and Technology, vol. 4, no. 1, p. 014008, 2018.  
([iopscience:2058-9565/aad3e4](#))
- [15] V. Havlicek, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, J. M. Gambetta. *Supervised learning with quantum-enhanced feature spaces*. Nature, vol. 567, pp. 209-212, 2019.  
([nature:s41586-019-0980-2](#))

- [16] N. Moll, P. Barkoutsos, L. Bishop, J. M. Chow, A. Cross, D. J. Egger, S. Filipp, A. Fuhrer, J. M. Gambetta, M. Ganzhorn, et al. *Quantum optimization using variational algorithms on near-term quantum devices*. Quantum Science and Technology, vol. 3, no. 3, p. 030503, 2018.  
([iopscience:2058-9565/aab822](https://iopscience.iop.org/article/10.1088/2058-9565/aab822))

# Приложения

## Приложение А

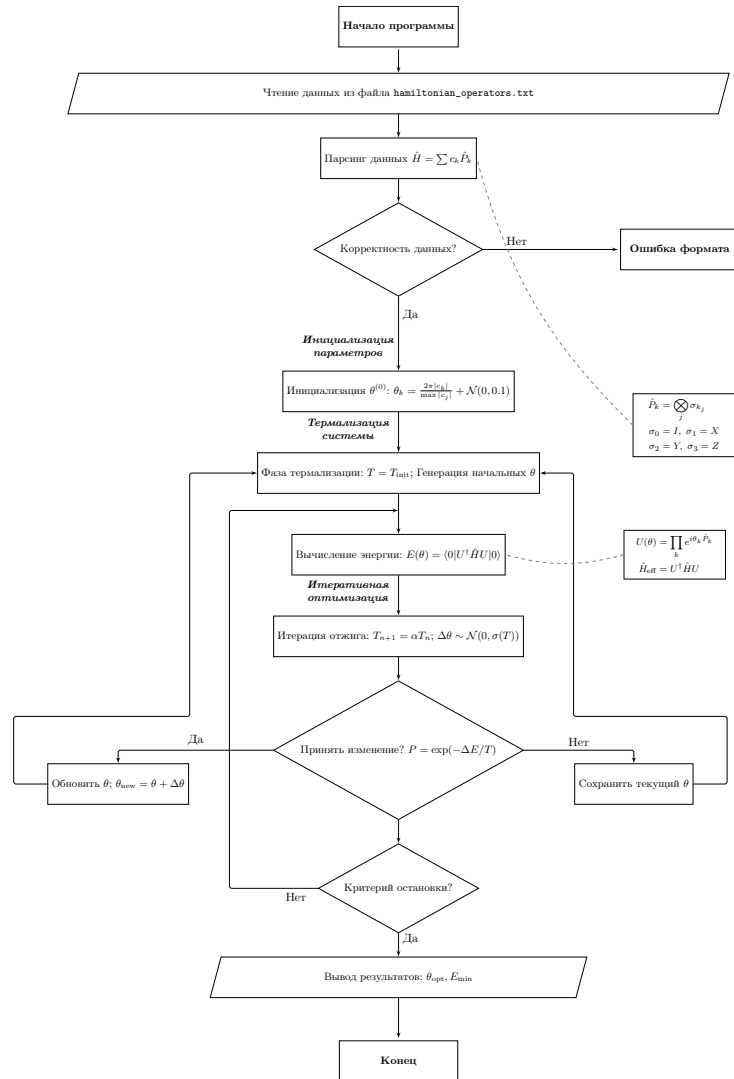


Рис. 2.1: Подробная схема квантового вариационного алгоритма

# Приложение Python

```
1 import sys
2 import io
3 import numpy as np
4 from pathlib import Path
5 from rich.console import Console
6 from rich.table import Table
7 from rich.panel import Panel
8 from rich.progress import Progress, BarColumn, TextColumn, SpinnerColumn
9 from rich import box
10 from sympy import re as sp_re, im as sp_im
11 from functools import lru_cache
12 from typing import Any, List, Dict, Tuple, Union, Callable
13
14
15 def get_base_path() -> Path:
16     """
17     Определяет базовую директорию проекта.
18
19     Returns:
20         Path: Абсолютный путь к папке с .exe-файлом (если приложение зам
                орожено)
                или к корню проекта (в режиме разработки).
21
22     Примечания:
23         - sys.frozen (атрибут пакета PyInstaller) используется для опред
                еления,
24         был ли код скомпилирован в исполняемый файл.
25         - Это позволяет корректно определять относительные пути при запу
                ске
26         как из исходников, так и из собранного .exe.
27     """
28
29     if getattr(sys, "frozen", False):
30         # Для скомпилированного EXE возвращаем директорию исполняемого ф
31         айла.
32         return Path(sys.argv[0]).parent
33     else:
34         # Для разработки возвращаем корень проекта (на уровень выше
35         constants/).
36         return Path(__file__).parent.parent
37
38 # Абсолютный путь к файлу с гамильтонианом (описание операторов Паули и
39 их коэффициентов).
40 HAMILTONIAN_FILE_PATH: Path = get_base_path() / "params" /
41 "hamiltonian_operators.txt"
42
43 # Абсолютный путь к файлу для логирования вывода (очищается при запуске
44 ).
45 OUTPUT_FILE_PATH: Path = get_base_path() / "output.log"
```

```

42 # Карта произведения для базисных операторов Паули:
43 # (i, j) -> (мнимая единица с правильным знаком, индекс результата)
44 # Индексы: 0=I, 1=X, 2=Y, 3=Z
45 PAULI_MAP = {
46     (1, 2): (1j, 3), # X*Y = iZ
47     (2, 1): (-1j, 3), # Y*X = -iZ
48     (3, 1): (1j, 2), # Z*X = iY
49     (1, 3): (-1j, 2), # X*Z = -iY
50     (2, 3): (1j, 1), # Y*Z = iX
51     (3, 2): (-1j, 1), # Z*Y = -iX
52 }
53
54
55 def calculate_temp_steps(
56     initial_temp: float, cooling_rate: float, min_temp: float
57 ) -> int:
58     """
59     Вычисляет количество температурных шагов для метода отжига.
60
61     Алгоритм: на каждом шаге температура уменьшается по формуле:
62          $T_{n+1} = T_n * cooling\_rate$ 
63     Шаги считаются, пока температура не станет меньше min_temp.
64
65     Args:
66         initial_temp (float): Начальная температура.
67         cooling_rate (float): Множитель охлаждения ( $0 < cooling\_rate < 1$ ).
68         min_temp (float): Минимально допустимая температура.
69
70     Returns:
71         int: Число температурных шагов до достижения min_temp.
72     """
73     steps = 0
74     current_temp = initial_temp
75     while current_temp > min_temp:
76         current_temp *= cooling_rate
77         steps += 1
78     return steps
79
80
81 def console_and_print(console: Console, message: Any) -> None:
82     """
83     Выводит сообщение в консоль и дублирует его в лог-файл.
84
85     Args:
86         console (Console): объект rich.Console для форматированного вывода.
87         message (Any): строка, rich.Panel или другой объект, печатаемый в консоль.
88
89     Примечания:

```

```

90     - Используется rich для красивого форматирования в консоли и лог
91       ах.
92     - Лог хранит весь вывод, включая цветовые коды (если
93       export_text это поддерживает).
94     """
95     console.print(message)
96     with open(OUTPUT_FILE_PATH, "a", encoding="utf-8") as file:
97         file.write(console.export_text() + "\n")
98
99 def create_table(
100     columns: List[Dict[str, str]],
101     data: List[List[Any]],
102     title: str,
103     border_style: str = "yellow",
104 ) -> Panel:
105     """
106     Создает таблицу rich.Table и оборачивает ее в rich.Panel для консоль
107       ного вывода.
108
109     Args:
110         columns (List[Dict[str, str]]): Описание столбцов (ключи: name,
111           style, justify).
112         data (List[List[Any]]): Массив данных для строк таблицы.
113         title (str): Заголовок панели.
114         border_style (str): Цвет рамки панели.
115
116     Returns:
117         Panel: Панель с таблицей для печати в консоль.
118     """
119     table = Table(box=box.ROUNDED, border_style=border_style)
120     for col in columns:
121         table.add_column(
122             col["name"],
123             justify=col.get("justify", "default"),
124             style=col.get("style", ""),
125         )
126     for row in data:
127         table.add_row(*row)
128     return Panel(table, title=title, border_style=border_style)
129
130 def format_ansatz(
131     pauli_operators: List[Tuple[complex, List[int]]],
132     result: Dict[Tuple[int, ...], complex],
133 ) -> Tuple[str, str]:
134     """
135     Форматирует вариационный анзац в символьное и численное представлени
136       е.
137
138     Args:

```



```

136     pauli_operators (List[Tuple[complex, List[int]]]): Список операторов Паули для анзаца,
137     где каждый оператор представлен кортежем из коэффициента и вектора индексов.
138     result (Dict[Tuple[int, ...], complex]): Разложение анзаца после экспоненцирования
139     по базису Паули (ключ: вектор индексов, значение: коэффициент).
140
141 Returns:
142     Tuple[str, str]: Строки символьного (произведение экспонент) и численного (разложение) представления.
143
144 Символьное представление важно для понимания структуры унитарного оператора:
145 
$$U(\theta) = \exp(i \cdot \theta_1 \cdot c_1 \cdot \sigma_1) \cdot \exp(i \cdot \theta_2 \cdot c_2 \cdot \sigma_2) \cdots$$

146
147 Численное разложение полезно для анализа конечного состояния оператора:
148 
$$U = \alpha_0 \cdot I + \alpha_1 \cdot \sigma_1 + \alpha_2 \cdot \sigma_2 + \cdots$$

149 """
150 ansatz_symbolic = "U(θ) = " + " * ".join(
151     [
152         f"exp(i·θ_{i+1}·{format_complex_number(coeff)})·σ_{''.join(map(str, op))}"
153         for i, (coeff, op) in enumerate(pauli_operators)
154     ]
155 )
156
157 ansatz_numeric = "U = " + " + ".join(
158     [
159         f"{format_complex_number(c)}·σ_{''.join(map(str, op))}"
160         for op, c in result.items()
161         if abs(c) > 1e-12
162     ]
163 )
164 return ansatz_symbolic, ansatz_numeric
165
166
167 def format_complex_number(c) -> str:
168     """
169     Форматирует комплексное число (или выражение SymPy) в строку с подавлением артефактов.
170
171 Args:
172     c (complex|sympy.Expr): Комплексное число или выражение.
173
174 Returns:
175     str: Отформатированное комплексное число (пример: '1.2-3i').
176
177 Особенности:
178     - Если число очень близко к нулю (<1e-12), оно не выводится.

```

```

179     - Мнимая часть  $\pm 1$  форматируется как  $\pm i$ .
180     - Корректно обрабатывает как стандартные числа, так и объекты
      SymPy.
181     """
182     real = float(sp_re(c))
183     imag = float(sp_im(c))
184
185     real_str = format_number(real) if abs(real) > 1e-12 else ""
186     imag_str = ""
187
188     if abs(imag) > 1e-12:
189         abs_imag = abs(imag)
190         imag_value = format_number(abs_imag)
191         if imag_value == "1":
192             imag_str = "i" if imag > 0 else "-i"
193         else:
194             imag_sign = "" if imag > 0 else "-"
195             imag_str = f"{imag_sign}{imag_value}i"
196
197     parts = []
198     if real_str:
199         parts.append(real_str)
200     if imag_str:
201         parts.append(imag_str)
202
203     if not parts:
204         return "0"
205
206     result = parts[0]
207     for part in parts[1:]:
208         if part.startswith("-"):
209             result += f"-{part[1:]}"
210         else:
211             result += f"+{part}"
212
213     return result.replace("+ -", "- ").replace("1i",
214         "i").replace(".0i", "i")
215
216 def format_number(num: float | int) -> str:
217     """
218     Форматирует число для вывода, корректно подавляя артефакты округлени
219     я.
220
221     Args:
222         num (float|int): Число для форматирования.
223
224     Returns:
225         str: Число в виде строки, без лишних нулей и ошибок округления.
226     """
227     if abs(num - round(num)) < 1e-15:

```

```

227         return str(int(round(num)))
228     s = f"{num:.14f}".rstrip("0").rstrip(".")
229     if s.startswith("."):
230         s = "0" + s
231     elif s.startswith("-."):
232         s = s.replace("-.", "-0.")
233     if "." in s:
234         int_part, dec_part = s.split(".")
235         dec_part = dec_part[:4].ljust(4, "0").rstrip("0")
236         s = f"{int_part}.{dec_part}" if dec_part else int_part
237     return s
238
239
240 def get_operator_for_console(c: Union[complex, float, int], i: str) ->
    str:
241     """
242     Формирует строку для красивого вывода оператора Паули.
243
244     Args:
245         c (complex|float|int): Коэффициент перед оператором.
246         i (str): Индексная строка оператора (например, '03' для  $I \otimes Z$ ).
247
248     Returns:
249         str: Строка вида ' $\sigma_i$ ' или ' $c \cdot \sigma_i$ '
250             (если коэффициент не равен 1).
251     """
252     if c == 1:
253         return f" $\sigma_{\{i\}}$ "
254     else:
255         return f"{format_complex_number(c)} $\sigma_{\{i\}}$ "
256
257
258 def initialize_environment() -> Console:
259     """
260     Инициализирует окружение для запуска программы:
261     - Очищает лог-файл с прошлых запусков.
262     - Возвращает объект rich.Console для форматированного вывода.
263
264     Returns:
265         Console: Готовый к использованию rich.Console.
266     """
267     if OUTPUT_FILE_PATH.exists():
268         OUTPUT_FILE_PATH.unlink()
269     return Console(force_terminal=True, color_system="truecolor",
270                   record=True)
271
272 def print_composition_table(
273     console: Console,
274     pauli_compose: Callable[[tuple, tuple], Tuple[complex, tuple]],
275     pauli_strings: List[List[int]],

```

```

276 ) -> None:
277     """
278     Выводит таблицу композиции операторов Паули для всех их пар.
279
280     Args:
281         console (Console): Объект rich.Console.
282         pauli_compose (Callable): Функция для композиции двух операторов
283             Паули.
284         pauli_strings (List[List[int]]): Список векторных индексов опера-
285             торов Паули.
286     """
287     results = []
288     # Перебор всех пар операторов Паули
289     for s1 in pauli_strings:
290         for s2 in pauli_strings:
291             coeff, product = pauli_compose(tuple(s1), tuple(s2))
292             results.append((s1, s2, format_complex_number(coeff),
293                             product))
294
295     table_data = [
296         [str(s1), str(s2), str(h).lower(), str(p)] for s1, s2, h, p in
297         results
298     ]
299     console_and_print(
300         console,
301         create_table(
302             columns=[
303                 {"name": "Оператор 1", "style": "cyan", "justify":
304                     "center"},
305                 {"name": "Оператор 2", "style": "magenta", "justify":
306                     "center"},
307                 {"name": "Коэффициент", "style": "green", "justify":
308                     "center"},
309                 {"name": "Результат", "style": "red", "justify":
310                     "center"},
311             ],
312             data=table_data,
313             title="Композиции операторов Паули",
314             border_style="green",
315         ),
316     )
317
318 def print_hamiltonian(
319     console: Console, pauli_operators: List[Tuple[complex, List[int]]]
320 ) -> None:
321     """
322     Выводит гамильтониан в удобочитаемом виде.
323
324     Args:
325         console (Console): rich.Console для вывода.

```

```

319     pauli_operators (List[Tuple[complex, List[int]]]): Список операторов Паули с коэффициентами.
320     """
321     hamiltonian_str = "H = " + " + ".join(
322         [get_operator_for_console(c, "".join(map(str, i))) for c, i in
          pauli_operators]
323     )
324     console_and_print(
325         console,
326         Panel(
327             hamiltonian_str,
328             title="[bold]Введенный гамильтониан[/bold]",
329             border_style="green",
330         ),
331     )
332
333
334 def print_pauli_table(
335     console: Console, pauli_operators: List[Tuple[complex, List[int]]]
336 ) -> None:
337     """
338     Выводит таблицу всех операторов Паули из гамильтониана.
339
340     Args:
341         console (Console): rich.Console для вывода.
342         pauli_operators (List[Tuple[complex, List[int]]]): Список операторов Паули с коэффициентами.
343     """
344     table_data = [[format_complex_number(c), str(i)] for c, i in
                    pauli_operators]
345     console_and_print(
346         console,
347         create_table(
348             columns=[
349                 {"name": "Коэффициент", "style": "cyan"},
350                 {"name": "Индекс", "style": "magenta", "justify":
                  "center"},
351             ],
352             data=table_data,
353             title="Операторы Паули",
354             border_style="purple",
355         ),
356     )
357
358
359 from pathlib import Path
360 from typing import List, Union
361
362
363 def read_file_lines(file_path: Union[str, Path], ignore_comments: bool)
    -> List[str]:

```

```

364 """
365 Считывает строки из файла, игнорируя комментарии (начинающиеся с
    '#').
366
367 Args:
368     file_path (str|Path): Путь к файлу.
369     ignore_comments (bool): Если True, строки, начинающиеся с '#', и
        гнорируются.
370
371 Returns:
372     List[str]: Список строк без лишних пробелов и пустых строк.
373
374 Raises:
375     FileNotFoundError: Если файл не существует.
376 """
377 file_path = Path(file_path) if not isinstance(file_path, Path) else
    file_path
378 if not file_path.exists():
379     raise FileNotFoundError(f"Файл {file_path} не найден.")
380 with open(file_path, "r") as file:
381     return [
382         line.strip()
383         for line in file
384         if not (ignore_comments and line.strip().startswith("#"))
385     ]
386
387
388 def read_hamiltonian_data(
389     file_path,
390 ) -> Tuple[List[Tuple[complex, List[int]]], List[List[int]]]:
391     """
392     Читает список операторов Паули из текстового файла.
393
394     Формат файла:
395         <действительная часть> <мнимая часть> <строка Паули>
396
397     Returns:
398         Tuple[List[Tuple[complex, List[int]]], List[List[int]]]:
399             - Список операторов (коэффициент, индексы Паули)
400             - Список только индексов (без коэффициентов)
401     """
402     lines = read_file_lines(file_path, ignore_comments=False)
403     pauli_operators: List[Tuple[complex, List[int]]] = []
404     pauli_strings: List[List[int]] = []
405     for line in lines:
406         parts = line.strip().split()
407         if len(parts) == 3:
408             real_part, imag_part, index_str = (
409                 float(parts[0]),
410                 float(parts[1]),
411                 str(parts[2]),

```

```

412         )
413         coefficient = np.complex128(real_part + imag_part * 1j)
414         index_list = [int(c) for c in index_str]
415         if coefficient != 0:
416             pauli_operators.append((coefficient, index_list))
417             pauli_strings.append(index_list)
418     return pauli_operators, pauli_strings
419
420
421 def calculate_ansatz(
422     theta: np.ndarray, pauli_operators: List[Tuple[complex, List[int]]]
423 ) -> Tuple[Dict[Tuple[int, ...], complex], str, str]:
424     """
425     Вычисляет вариационный анзац в виде произведения экспонент операторов
426     в Паули.
427
428     Args:
429         theta (np.ndarray): Вектор параметров (обычно одного размера с ч
430         ислом операторов).
431         pauli_operators (List[Tuple[complex, List[int]]]): Операторы Пау
432         ли с коэффициентами.
433
434     Returns:
435         Tuple[
436             Dict[Tuple[int, ...], complex], # Разложение анзаца по Пау
437             ли-операторам
438             str, # Символьное представление
439             (произведение экспонент)
440             str # Численное разложение
441         ]
442
443     Алгоритм:
444          $U(\theta) = \prod_j \exp(i \cdot \theta_j \cdot |c_j| \cdot \sigma_j)$ 
445         Реализуется по принципу покомпонентного разложения через формулу
446         Эйлера:
447          $\exp(i \cdot \alpha \cdot \sigma) = \cos(\alpha) \cdot I + i \cdot \sin(\alpha) \cdot \sigma$ 
448         С каждым новым оператором результат рекурсивно обновляется через
449         pauli_compose.
450     """
451     operator_length = len(pauli_operators[0][1])
452     result: Dict[Tuple[int, ...], complex] = {tuple([0] *
453         operator_length): 1.0}
454
455     for t, (coeff, op) in zip(theta, pauli_operators):
456         angle = t * abs(coeff) # Используем абсолютное значение коэффиц
457         иента!
458         cos_t = np.cos(angle)
459         sin_t = np.sin(angle)
460         new_result: Dict[Tuple[int, ...], complex] = {}
461         op_tuple = tuple(op)
462         for existing_op, existing_coeff in result.items():

```

```

454         # cos(angle) * I (сохраняем индекс базисного оператора)
455         new_result[existing_op] = (
456             new_result.get(existing_op, 0) + existing_coeff * cos_t
457         )
458         # i*sin(angle)*σ (композиция Паули)
459         compose_coeff, compose_op = pauli_compose(existing_op,
460             op_tuple)
461         final_coeff = existing_coeff * 1j * sin_t * compose_coeff
462         new_result[compose_op] = new_result.get(compose_op, 0) +
463             final_coeff
464         result = new_result
465
466     symbolic_str, numeric_str = format_ansatz(pauli_operators, result)
467     return result, symbolic_str, numeric_str
468
469
470
471 from typing import Tuple, Dict
472
473
474 def calculate_expectation(uhu_dict: Dict[Tuple[int, ...], complex]) ->
475     float:
476     """
477     Вычисляет среднее значение энергии  $\langle 0|U^\dagger H U|0\rangle$ .
478
479     Args:
480         uhu_dict (Dict[Tuple[int, ...], complex]): Разложение оператора
481              $U^\dagger H U$ 
482             по базису Паули (ключ: индекс, значение: коэффициент).
483
484     Returns:
485         float: Ожидаемое значение (энергия).
486
487     Примечание:
488         - Только те операторы, которые не изменяют  $|0\dots 0\rangle$ 
489           (то есть содержащие только I и Z),
490           могут дать нетривиальный вклад в среднее значение.
491     """
492     expectation = 0.0
493     for op, coeff in uhu_dict.items():
494         if all(p in (0, 3) for p in op): # Только I или Z на каждом куб
495             ите
496             expectation += coeff.real
497     return expectation
498
499
500 def compute_uhu(
501     u_dict: Dict[Tuple[int, ...], complex], h_terms:
502         List[Tuple[complex, List[int]]]
503 ) -> Dict[Tuple[int, ...], complex]:
504     """
505     Вычисляет оператор  $U^\dagger H U$  в базисе Паули.

```



```

499
500     Args:
501         u_dict (Dict[Tuple[int, ...], complex]): Разложение оператора U.
502         h_terms (List[Tuple[complex, List[int]]]): Операторы гамильтониа
           на с коэффициентами.
503
504     Returns:
505         Dict[Tuple[int, ...], complex]: Разложение  $U^\dagger H U$ 
506         по Паули операторам.
507
508     Алгоритм:
509          $(U^\dagger H U)_{\{kl\}} = \sum_{\{i,j\}} \text{conj}(U_{\{ik\}}) * H_{\{ij\}} * U_{\{jl\}}$ 
510         Реализовано через перебор всех Паули-операторов.
511     """
512     uhu_dict: Dict[Tuple[int, ...], complex] = {}
513     u_items = list(u_dict.items())
514     for coeff_h, op_h in h_terms:
515         op_h_tuple = tuple(op_h)
516         for j_op, j_coeff in u_items:
517             conj_j_coeff = np.conj(j_coeff)
518             c1, op_uh = pauli_compose(j_op, op_h_tuple)
519             for k_op, k_coeff in u_items:
520                 c2, op_uhu = pauli_compose(op_uh, k_op)
521                 total_coeff = conj_j_coeff * k_coeff * coeff_h * c1 * c2
522                 uhu_dict[op_uhu] = uhu_dict.get(op_uhu, 0) + total_coeff
523     return uhu_dict
524
525
526 def generate_neighbor_theta(
527     current_theta: np.ndarray, step_size: float = 0.1
528 ) -> np.ndarray:
529     """
530     Генерирует новое состояние  $\theta$ , добавляя нормальный шум
531     с заданной дисперсией, и приводит все значения к диапазону  $[0, 2\pi)$ .
532
533     Args:
534         current_theta (np.ndarray): Текущий вектор параметров.
535         step_size (float): Стандартное отклонение для гауссового шума.
536
537     Returns:
538         np.ndarray: Новый вектор параметров.
539     """
540     perturbation = np.random.normal(scale=step_size,
541                                     size=current_theta.shape)
542     return (current_theta + perturbation) % (2 * np.pi)
543
544 def generate_shifted_theta(
545     pauli_operators: List[Tuple[complex, List[int]]]
546 ) -> np.ndarray:
547     """

```

```

548     Генерирует начальный вектор  $\theta$  для анзаца, масштабируя его пропорцион
        ально
549     модулям коэффициентов операторов Паули.
550
551     Args:
552         pauli_operators (List[Tuple[complex, List[int]]]): Операторы Пау
            ли.
553
554     Returns:
555         np.ndarray: Начальный вектор  $\theta$  (размер соответствует числу опера
            торов).
556     """
557     if not pauli_operators:
558         return np.array([], dtype=np.float64)
559     # Используем модуль коэффициента (абсолютная величина, чтобы избежат
        ь ошибок для комплексных коэффициентов)
560     coeffs = np.array([abs(op[0]) for op in pauli_operators],
        dtype=np.float64)
561     norm = np.linalg.norm(coeffs)
562     if norm < 1e-12:
563         return np.zeros(len(coeffs))
564     # Масштабируем на диапазон  $[0, 2\pi)$  и добавляем небольшой случайный ш
        ум
565     scaled = (coeffs / norm) * 2 * np.pi
566     return scaled + np.random.normal(0, 0.1, len(scaled))
567
568
569 def multiply_pauli(i: int, j: int) -> Tuple[complex, int]:
570     """
571     Перемножает два базисных оператора Паули.
572
573     Args:
574         i (int): Первый индекс (0=I, 1=X, 2=Y, 3=Z).
575         j (int): Второй индекс (0=I, 1=X, 2=Y, 3=Z).
576
577     Returns:
578         Tuple[complex, int]: Коэффициент и индекс результата.
579     """
580     if i == j:
581         return (1, 0)
582     if i == 0:
583         return (1, j)
584     if j == 0:
585         return (1, i)
586     return PAULI_MAP.get((i, j), (1, 0))
587
588
589 @lru_cache(maxsize=4096)
590 def pauli_compose(s1: tuple, s2: tuple) -> Tuple[complex, tuple]:
591     """
592     Перемножает два оператора Паули, заданных покубитно.

```

```

593
594     Args:
595         s1 (tuple): Индексы первого оператора (например, (0,3)
596                     для  $I \otimes Z$ ).
597         s2 (tuple): Индексы второго оператора.
598
599     Returns:
600         Tuple[complex, tuple]: Коэффициент и индексы результата.
601     """
602     coefficient = 1.0
603     result = []
604     for a, b in zip(s1, s2):
605         coeff, idx = multiply_pauli(a, b)
606         coefficient *= coeff
607         result.append(idx)
608     return coefficient, tuple(result)
609
610
611 def simulated_annealing(
612     initial_theta: np.ndarray,
613     pauli_operators: List[Any],
614     progress: Any,
615     task: Any,
616     initial_temp: float = 1000.0,
617     cooling_rate: float = 0.99,
618     min_temp: float = 1e-5,
619     num_iterations_per_temp: int = 500,
620     step_size: float = 0.5,
621 ) -> Tuple[np.ndarray, float]:
622     """
623     Алгоритм отжига (simulated annealing) для оптимизации параметров  $\theta$  в
        вариационного анзаца.
624
625     Args:
626         initial_theta (np.ndarray): Начальный вектор  $\theta$ .
627         pauli_operators (List[Tuple[complex, List[int]]]): Операторы Пау
            ли.
628         progress (Any): Индикатор прогресса (может быть None).
629         task (Any): Задача для прогресс-бара.
630         initial_temp (float): Начальная температура (чем выше - тем веро
            ятнее принять ухудшающее решение).
631         cooling_rate (float): Множитель охлаждения ( $0 < \text{cooling\_rate} < 1$ ).
632         min_temp (float): Минимально допустимая температура.
633         num_iterations_per_temp (int): Количество шагов на каждой темпер
            атуре.
634         step_size (float): Стандартное отклонение для шума  $\theta$ .
635
636     Returns:
637         Tuple[np.ndarray, float]: Оптимальный найденный  $\theta$  и соответствую
            щая энергия.

```

```

638
639 Принцип работы:
640     - На каждом шаге генерируется новое состояние  $\theta$  (случайным образом).
641     - Если энергия уменьшилась -- принимаем новое состояние.
642     - Если энергия увеличилась -- принимаем с вероятностью  $\exp(-\Delta E/T)$ .
643     - Температура постепенно понижается (охлаждение).
644
645 Важно:
646     - В термализации используется локальный случайный шаг (как и в основном цикле).
647     - Это обеспечивает более 'физичное' поведение отжига.
648 """
649 current_theta = initial_theta.copy()
650 best_theta = current_theta.copy()
651 best_energy = float("inf")
652 rng = np.random.default_rng()
653 temp = initial_temp
654 thermalization_steps = int(num_iterations_per_temp * 0.2)
655
656 while temp > min_temp:
657     # Этап термализации: локальные случайные шаги для прогрева цепочки
658     for _ in range(thermalization_steps):
659         neighbor_theta = generate_neighbor_theta(current_theta,
660             step_size)
661         ansatz_dict, _, _ = calculate_ansatz(neighbor_theta,
662             pauli_operators)
663         uhu_dict = compute_uhu(ansatz_dict, pauli_operators)
664         current_energy = calculate_expectation(uhu_dict)
665         if current_energy < best_energy:
666             best_theta = neighbor_theta.copy()
667             best_energy = current_energy
668             current_theta = neighbor_theta.copy()
669         if progress is not None:
670             progress.update(task, advance=1)
671
672     # Основной цикл отжига с возможностью принимать ухудшения
673     for _ in range(num_iterations_per_temp):
674         perturbation = rng.normal(
675             0, step_size * (temp / initial_temp),
676             current_theta.shape
677         )
678         neighbor_theta = (current_theta + perturbation) % (2 *
679             np.pi)
680         ansatz_dict, _, _ = calculate_ansatz(neighbor_theta,
681             pauli_operators)
682         uhu_dict = compute_uhu(ansatz_dict, pauli_operators)
683         current_energy = calculate_expectation(uhu_dict)
684         energy_diff = current_energy - best_energy

```

```

680         # Классическое правило Метрополиса
681         if energy_diff < 0 or rng.random() <
            np.exp(-energy_diff / temp):
682             current_theta = neighbor_theta.copy()
683             if current_energy < best_energy:
684                 best_theta = current_theta.copy()
685                 best_energy = current_energy
686         if progress is not None:
687             progress.update(task, advance=1)
688         temp *= cooling_rate
689
690     return best_theta, best_energy
691
692
693 # Установка корректной кодировки стандартных потоков для поддержки
Unicode
694 sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding="utf-8")
695 sys.stderr = io.TextIOWrapper(sys.stderr.buffer, encoding="utf-8")
696
697
698 def main() -> None:
699     """
700     Основная функция программы. Реализует следующий цикл:
701     1. Чтение гамильтониана из файла.
702     2. Вывод операторов Паули и их композиции.
703     3. Оптимизация анзаца методом отжига для последовательных подмноже
       ств операторов.
704     4. Поиск минимальной энергии и вывод оптимального анзаца.
705
706     Программа построена как демонстрация вариационного квантового алгори
       тма
707     с классическим оптимизатором (simulated annealing).
708     """
709     console = initialize_environment()
710
711     # Проверка наличия файла с гамильтонианом
712     if not HAMILTONIAN_FILE_PATH.exists():
713         msg = (
714             f"Файл [bold]{HAMILTONIAN_FILE_PATH}[/] не найден!\n"
715             "Убедитесь, что рядом с EXE есть папка [bold]params[/] с фай
               лом [bold]hamiltonian_operators.txt[/]. "
716         )
717         console_and_print(console, Panel(msg, border_style="red"))
718         return
719
720     try:
721         pauli_operators, pauli_strings =
            read_hamiltonian_data(HAMILTONIAN_FILE_PATH)
722         print_hamiltonian(console, pauli_operators)
723         print_pauli_table(console, pauli_operators)
724         print_composition_table(console, pauli_compose, pauli_strings)

```

```

725 except FileNotFoundError:
726     console_and_print(
727         console,
728         Panel(
729             f"[red]Файл {HAMILTONIAN_FILE_PATH} не найден[/red]",
730             border_style="red"
731         ),
732     )
733     return
734
735 if len(pauli_operators) < 2:
736     console_and_print(
737         console,
738         Panel("[red]Требуется минимум 2 оператора Паули[/red]",
739             border_style="red"),
740     )
741     return
742
743 # Параметры отжига
744 SA_PARAMS = {
745     "initial_temp": 100.0,
746     "cooling_rate": 0.95,
747     "min_temp": 1e-3,
748     "num_iterations_per_temp": 100,
749     "step_size": 0.1,
750 }
751
752 # Оценка общего количества шагов для прогресс-бара
753 thermalization_steps = int(SA_PARAMS["num_iterations_per_temp"] *
754     0.2)
755 temp_steps = calculate_temp_steps(
756     SA_PARAMS["initial_temp"], SA_PARAMS["cooling_rate"],
757     SA_PARAMS["min_temp"]
758 )
759 steps_per_m = temp_steps * (
760     thermalization_steps + SA_PARAMS["num_iterations_per_temp"]
761 )
762 total_steps = steps_per_m * (len(pauli_operators) - 1)
763
764 best_energy = float("inf")
765 best_result = None
766 all_results = []
767
768 # Запуск прогресс-бара с симпатичным оформлением
769 with Progress(
770     SpinnerColumn(),
771     TextColumn("[progress.description]{task.description}"),
772     BarColumn(bar_width=None),
773     TextColumn("[progress.percentage]{task.percentage:>3.0f}%"),
774 ) as progress:
775     task = progress.add_task("[cyan]Отжиг...", total=total_steps)

```

```

772
773     # Последовательно увеличиваем число операторов в анзаце
774     for m in range(2, len(pauli_operators) + 1):
775         current_ops = pauli_operators[:m]
776         initial_theta = generate_shifted_theta(current_ops)
777
778     # Оптимизация параметров для текущего поднабора операторов
779     optimized_theta, energy = simulated_annealing(
780         initial_theta=initial_theta,
781         pauli_operators=current_ops,
782         progress=progress,
783         task=task,
784         **SA_PARAMS,
785     )
786
787     all_results.append(
788         {
789             "m": m,
790             "theta": optimized_theta,
791             "energy": energy,
792             "operators": current_ops,
793         }
794     )
795
796     if energy < best_energy:
797         best_energy = energy
798         best_result = all_results[-1]
799
800     # Выводим результаты оптимизации
801     if best_result is None:
802         console_and_print(
803             console, Panel("[red]Не удалось найти решение[/red]",
804                             border_style="red")
805         )
806         return
807
808     _, ansatz_symbolic, ansatz_numeric = calculate_ansatz(
809         best_result["theta"], best_result["operators"]
810     )
811
812     console_and_print(
813         console,
814         Panel(
815             ansatz_symbolic,
816             title="[bold]Символьное представление анзаца[/]",
817             border_style="green",
818         ),
819     )
820
821     console_and_print(
822         console,

```

```

822     Panel(
823         ansatz_numeric,
824         title="[bold]Численное представление анзаца[/]",
825         border_style="purple",
826     ),
827 )
828
829 console_and_print(
830     console,
831     Panel(
832         f"{best_result['energy']:.6f}",
833         title="[bold]Энергия ( $\langle 0|U^\dagger H U|0 \rangle$  для состояния  $|0\dots 0\rangle$ )[/]",
834         border_style="green",
835     ),
836 )
837
838 input("Нажмите Enter для выхода...")
839
840
841 if __name__ == "__main__":
842     main()

```