Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра суперкомпьютеров и квантовой информатики

Кулагин Алексей Владимирович

Переход сложных квантовых состояний между полостями в модели Тависа-Каммингса-Хаббарда

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Научный руководитель:
д.ф.-м.н., профессор
Ю. И. Ожигов

Москва, 2018

Содержание

1 Введение

Конечномерная модель квантовой электродинамики Тависа-Каммингса описывает взаимодействие ансамбля двухуровневых атомов, помещенных в оптический резонатор, с одномодовым полем частоты, близкой к собственной частоте резонатора. Однако такие модели могут иметь и гораздо более широкое применение в силу их простоты: например, в их рамках можно описывать переход атомных возбуждений по системе оптических полостей, соединенных оптическим волокном (модель Тависа-Каммингса-Хаббарда), проводимость таких систем и связанные с ней эффекты (квантовое бутылочное горлышко, DAT) или квантовые блуждания на графах.

Для слабой связи атомов с полем справедливо приближение RWA (вращающейся волны), которое позволяет либо решить задачу аналитически (в случае одного атома) или существенно упростить компьютерное моделирование квантовой динамики - для случая нескольких атомов. Для RWA приближения пространство квантовых состояний системы распадается на сумму ортогональных подпространств, инвариантных относительно гамильтониана, и обладающих, к тому же, относительно малой размерностью, которая, в случае одинаковой силы взаимодействия поля и атомов, растет линейно с их числом. В случае же различных сил взаимодействия этот рост становится экспоненциальным, что обусловливает необходимость применения суперкомпьютеров для установления характера квантовой динамики для больших ансамблей.

В данной работе исследуется важный тип динамики – осцилляции между возбужденным состоянием атомного ансамбля и поля, при которых вся энергия системы переходит от атомов к полю и обратно с большой амплитудой. В многоатомном ансамбле имеется множество состояний, по которым амплитуда распределяется в ходе процесса, так что концентрация амплитуды на строго неравновесных состояниях не является очевидной и не может быть непосредственно выведена из вида гамильтониана. Само существование таких осцилляций не следует непосредственно из подобных осцилляций Раби для одного атома. В случае сложных систем такие осцилляции могут быть проанализированы только в результате численного компьютерного и суперкомпьютерного моделирования.

2 Постановка задачи

В настоящей работе рассматривается динамика квантовых состояний ансамбля двухуровневых атомов и одномодового поля в резонаторе в рамках модели Тависа-Каммингса в RWA приближении с шумами, а также в рамках модели Тависа-Каммингса-Хаббарда для случая двух взаимодействующих полостей.

Предметом исследования являются

- изучение характера осцилляций между двумя группами атомов одинаковой численности и одинаковой силы взаимодействия с полем
- изучение влияния фотонной накачки на качество и резкость ансамблевых осцилляций
- выявление зависимости периода осцилляций от числа атомов в группе, коэффициента взаимодействия атомов с полем, а также первоначального числа фотонов в полости резонатора
- возможность получения осцилляций хорошего качества для многоатомных ансамблей в условиях слабого взаимодействия атомов с полем
- исследование эффекта квантового "эха": перехода состояния атомов из одной полости в другую и возврат этого состояния в первоначальную полость. В частности, нахождение условий возникновения осцилляций атомных групп, распределенных между полостями

3 Модель Тависа-Каммингса

Рассмотрим N двухуровневых атомов, взаимодействующих с модой электромагнитного поля в идеальном резонаторе.

Атомы взаимодействуют с электромагнитным полем полости, испуская или поглощая фотон. При поглощении фотона атом возбуждается, при испускании – переходит в основное состояние.

Обозначим: $|0\rangle$ – основное состояние,
$|1\rangle$ – возбужденное состояние атома.

Введем также следующие операторы:

a^+, a – операторы рождения и уничтожения фотонов резонаторной моды,
σ_i^+, σ_i – повышающий и понижающий операторы i-го атома.

$$\begin{array}{rclrclrcl} a^+|n\rangle &=& \sqrt{n+1}\,|n+1\rangle & \sigma^+|0\rangle &=& |1\rangle & \sigma^+|1\rangle &=& 0 \\ a|n\rangle &=& \sqrt{n}\,|n-1\rangle & \sigma|1\rangle &=& |0\rangle & \sigma|0\rangle &=& 0 \end{array}$$

В представлении взаимодействия такая система описывается гамильтонианом Тависа-Каммингса:

$$H_{TC} = \underbrace{hw_c\, a^+a}_{H_{field}} + \underbrace{hw_a \sum_{i=1}^{N} \sigma_i^+\sigma_i}_{H_{atoms}} + \underbrace{\sum_{i=1}^{N} g_i(\sigma_i^+ + \sigma_i)(a^+ + a)}_{H_{int}} \qquad (1)$$

h – постоянная Планка
w_c – частота фотонов в полости
w_a – частота атомного перехода
g_i – константа взаимодействия i-го двухуровневого атома с полем

$|w_c - w_a| \ll w_c + w_a$ (условие применимости модели ТС)

Пренебрегая членами σ^+a^+ и σa, не сохраняющими энергию, перепишем гамильтониан в следующем виде:

$$H_{TC} \approx H_{RWA} = \underbrace{hw_c\, a^+a}_{H_{field}} + \underbrace{hw_a \sum_{i=1}^{N} \sigma_i^+\sigma_i}_{H_{atoms}} + \underbrace{\sum_{i=1}^{N} g_i(\sigma_i^+a + \sigma_i a^+)}_{H_{int}} \qquad (2)$$

Приближение (2) называется *приближением вращающейся волны* (rotating wave approximation), или RWA, и имеет место при условии

$$\frac{g_i}{hw_c} \approx \frac{g_i}{hw_a} \ll 1 \quad \forall\, i = \overline{1, N}. \qquad (3)$$

3.1 Ансамблевые осцилляции

Пусть ансамбль двухуровневых атомов разделен на две равные половины A_1 и A_2 с номерами $1, 2, ..., n$ и $n+1, n+2, \ldots, 2n$ соответственно и первоначально в полости находится m фотонов ("накачка").

Рассмотрим два состояния

$$|\Psi_0\rangle = |m\rangle_{ph}|00\ldots0\rangle_{A_1}|11\ldots1\rangle_{A_2}, \quad |\Psi_1\rangle = |m\rangle_{ph}|11\ldots1\rangle_{A_1}|00\ldots0\rangle_{A_2} \qquad (4)$$

Переходы между этими состояниями в ходе унитарной эволюции называются ансамблевыми осцилляциями. Качество таких осцилляций определяется естественным образом с помощью функций $f_0(t) = |\langle\Psi_0|\Psi(t)\rangle|^2,\ f_1(t) = |\langle\Psi_1|\Psi(t)\rangle|^2$, равных вероятности получения данных состояний при измерении текущего состояния $|\Psi(t)\rangle$ системы в момент времени t. В случае неунитарной эволюции смешанного состояния $\rho(t)$ вместо функций $f_0,\ f_1$ надлежит использовать функции согласованности $F_0(t) = Tr\left[\sqrt{\sqrt{\rho_0}\ \rho(t)\ \sqrt{\rho_0}}\right]$ и $F_1(t) = Tr\left[\sqrt{\sqrt{\rho_1}\ \rho(t)\ \sqrt{\rho_1}}\right]$ соответственно, где $\rho_0 = |\Psi_0\rangle\langle\Psi_0|,\ \rho_1 = |\Psi_1\rangle\langle\Psi_1|$.

Периодический повтор во времени состояний с близкими к единице показателями качества является критерием наличия ансамблевых осцилляций. Такие точки мы называем пиковыми. Важна также резкость осцилляций: скорость изменения функции качества в окрестности пиковых точек. Из вида гамильтониана ТС вытекает, что вектор чистого состояния при унитарной эволюции меняется в пределах подпространства $\mathcal{H}_0$, порожденного базисными состояниями вида $|c\rangle_{ph}|s_1\rangle_{A_1}|s_2\rangle_{A_2}$, такими что $c + \nu(s_1) + \nu(s_2) = n$. Его размерность $dim(\mathcal{H}_0)$ растет экспоненциально от n, и потому само существование ансамблевых осцилляций, в отличие от осцилляций Раби, является совершенно не тривиальным фактом.

Введем состояние $\{i\succ_{at} = \dfrac{1}{\sqrt{C_n^i}} \sum_{j:\ \nu(j)=i} |j\rangle$, где $\nu(j)$ – вес Хэмминга двоичного кортежа $|j\rangle$, равный числу возбуждений i в атомной группе.

В дальнейшем мы будем рассматривать квантовую динамику в приближении RWA состояний вида $|\Psi_0\rangle = |m\rangle_{ph}|00\ldots0\rangle_{A_1}|11\ldots1\rangle_{A_2}$ в базисе $|m\rangle_{ph}\{i, j\succ_{at}$:

$|m\rangle_{ph}$ – начальное фоковское (либо вакуумное) состояние поля,

$\{i, j\succ_{at} = \{i\succ_{A_1}\{j\succ_{A_2}$ – состояние (энергия возбуждения) атомных групп A_1, A_2.

На вертикальной оси отложена вероятность $p(t)$ получения состояния $\{i, j\succ_{at}$ в момент времени t, на горизонтальных – рассматриваемый временной интервал и базисные состояния системы ($|\Psi_0\rangle$ и $|\Psi_1\rangle$ – в крайних точках оси).

3.2 Осцилляции с накачкой фотонов

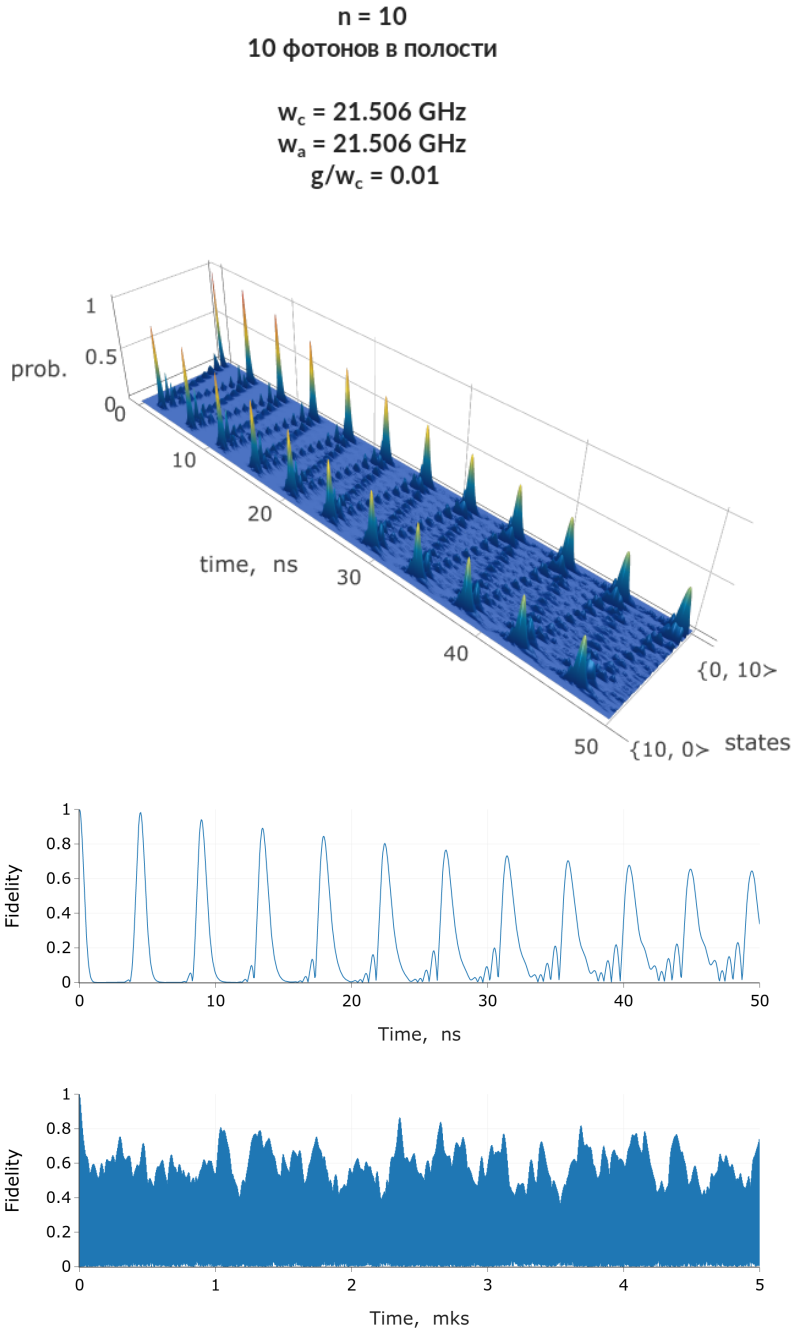


Рис. 1: $|\Psi_0\rangle = |10\rangle_{ph}\{0\succ_{A_1}\{10\succ_{A_2}$ (начальное состояние)
$|\Psi_1\rangle = |10\rangle_{ph}\{10\succ_{A_1}\{0\succ_{A_2}$

высокое качество осцилляций, доходящее до 0.8

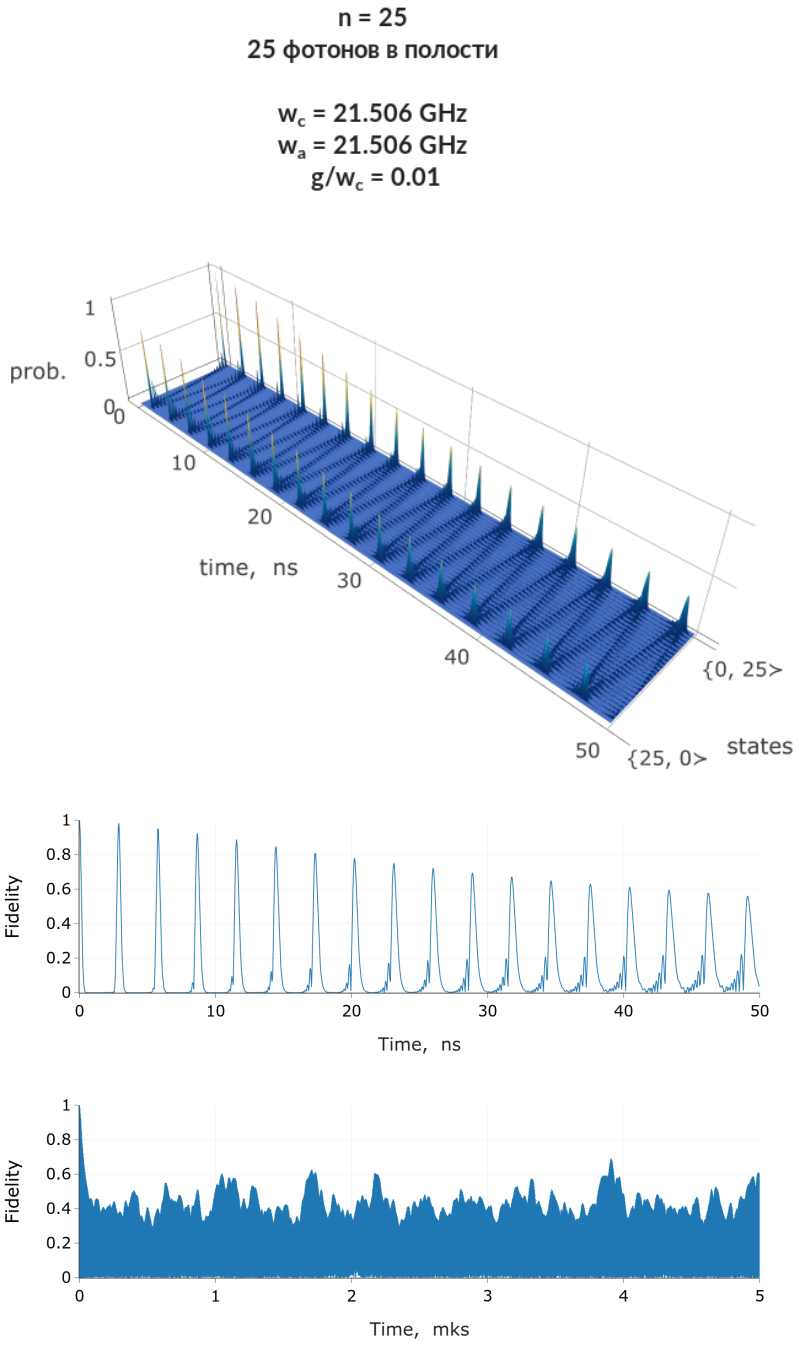


Рис. 2: $|\Psi_0\rangle = |25\rangle_{ph}\{0\succ_{A_1}\{25\succ_{A_2}$ (начальное состояние)
$|\Psi_1\rangle = |25\rangle_{ph}\{25\succ_{A_1}\{0\succ_{A_2}$

качество осцилляций в пределах 0.4-0.6;
очень отчетливый и резкий характер

3.3 Осцилляции произвольных ансамблевых состояний

Ансамблевые состояния вида $\alpha|00\ldots0\rangle_{A_1}|11\ldots1\rangle_{A_2} + \beta|11\ldots1\rangle_{A_1}|00\ldots0\rangle_{A_2}$ дают осцилляции хорошего качества в условиях фотонной накачки.

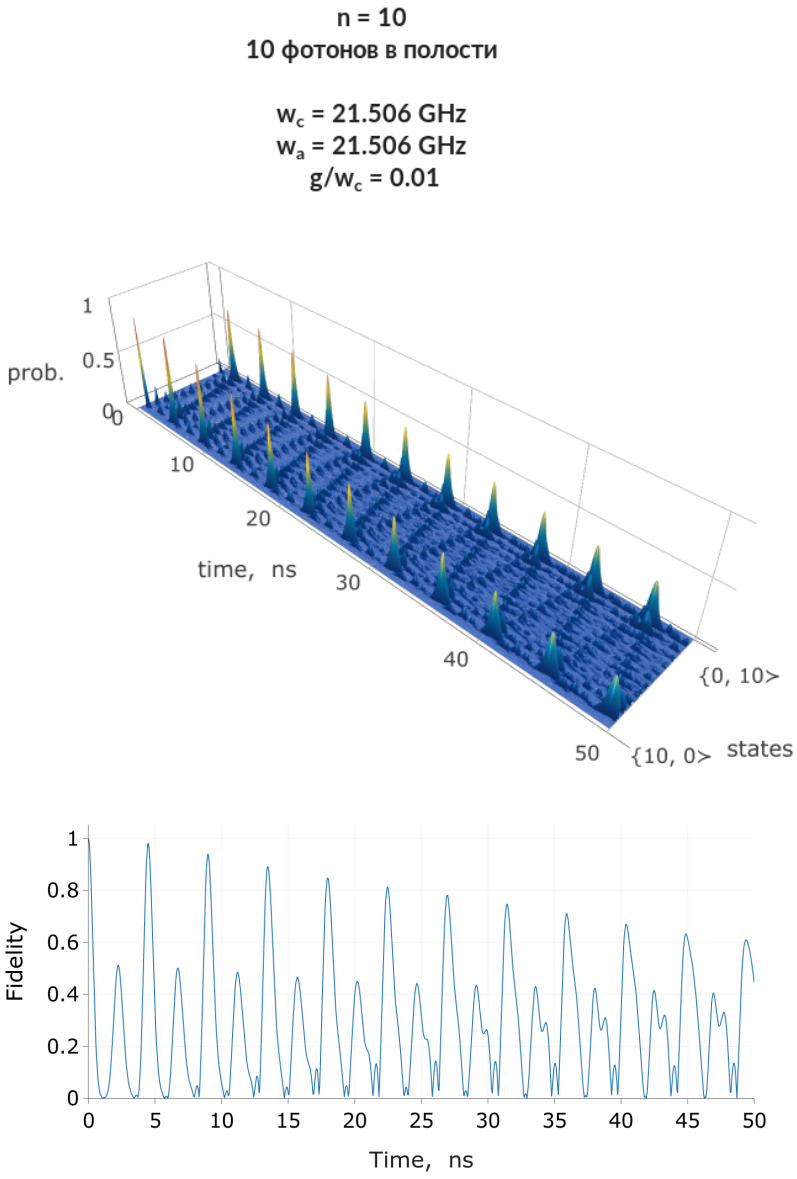


Рис. 3: $|\Psi_0\rangle = |10\rangle_{ph} \otimes (\alpha\{0\succ_{A_1}\{10\succ_{A_2} + \beta\{10\succ_{A_1}\{0\succ_{A_2})$

$$\alpha = 0.293 + 0.245\ i, \quad \beta = 0.204 + 0.901\ i$$

3.4 Осцилляции без накачки фотонов

Моделирование квантовой динамики состояний вида $|\Psi_0\rangle = |vac\rangle_{ph}\{0, n\succ_{at}$:

$|vac\rangle_{ph}$ – начальное вакуумное состояние поля,

$\{i, j\succ_{at} = \{i\succ_{A_1}\{j\succ_{A_2}$ – состояние атомных групп A_1, A_2.

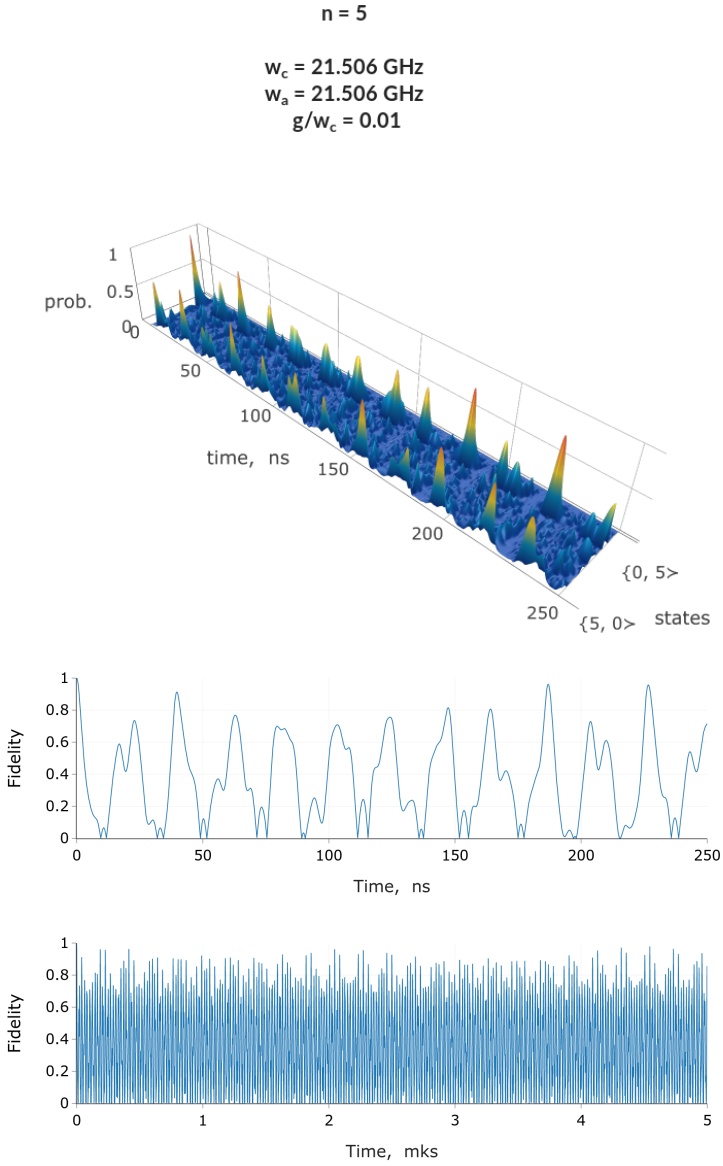


Рис. 4: $|\Psi_0\rangle = |vac\rangle_{ph}\{0\succ_{A_1}\{5\succ_{A_2}$ (начальное состояние)
$|\Psi_1\rangle = |vac\rangle_{ph}\{5\succ_{A_1}\{0\succ_{A_2}$

наблюдается высокое качество осцилляций, близкое к 1;
коллапсы и возрождения ансамблевых состояний

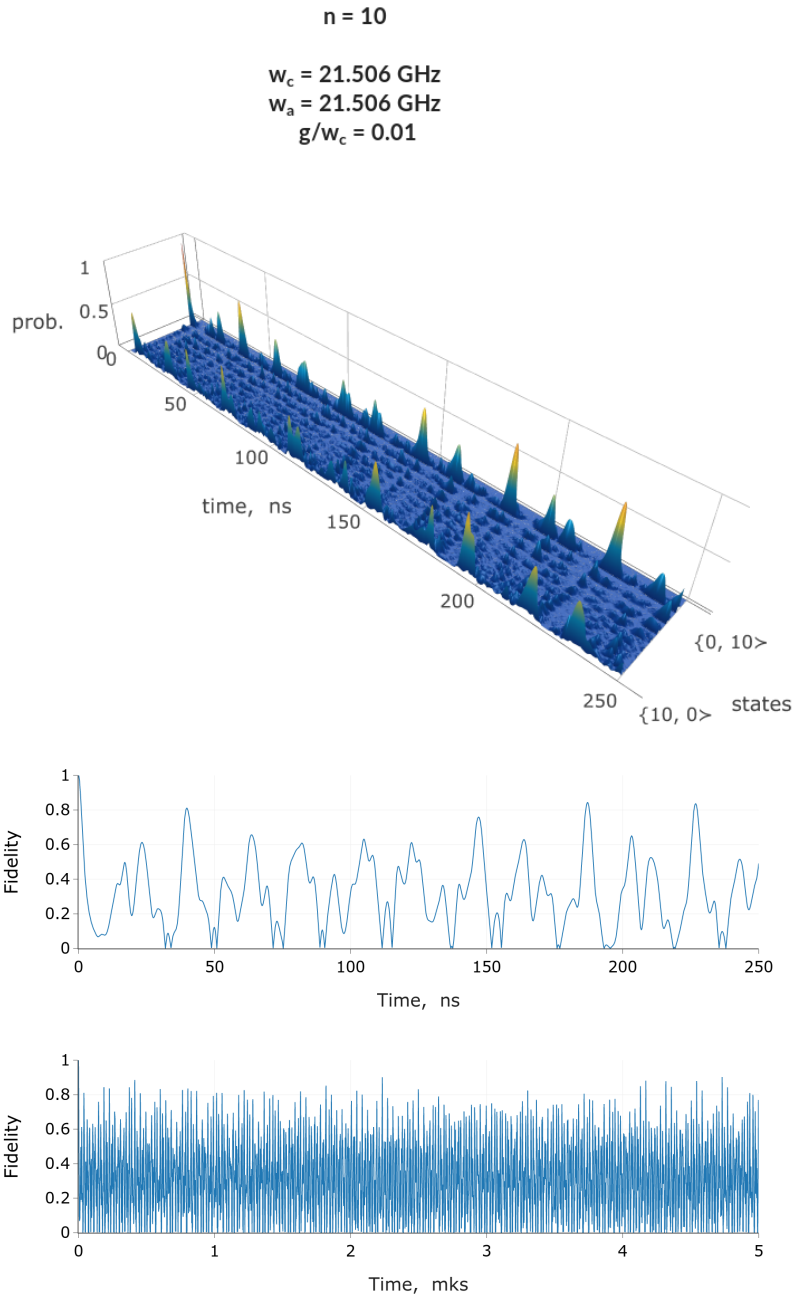


Рис. 5: $|\Psi_0\rangle = |vac\rangle_{ph}\{0\succ_{A_1}\{10\succ_{A_2}$ (начальное состояние)
$|\Psi_1\rangle = |vac\rangle_{ph}\{10\succ_{A_1}\{0\succ_{A_2}$

высокое качество осцилляций, достигающее 0.8;
коллапсы и возрождения ансамблевых состояний

3.5 Зависимость качества осцилляций от числа атомов и силы взаимодействия с полем

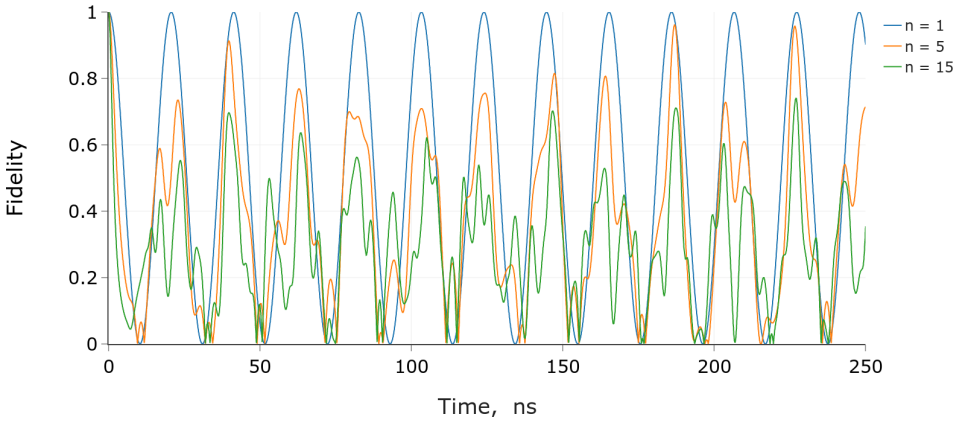
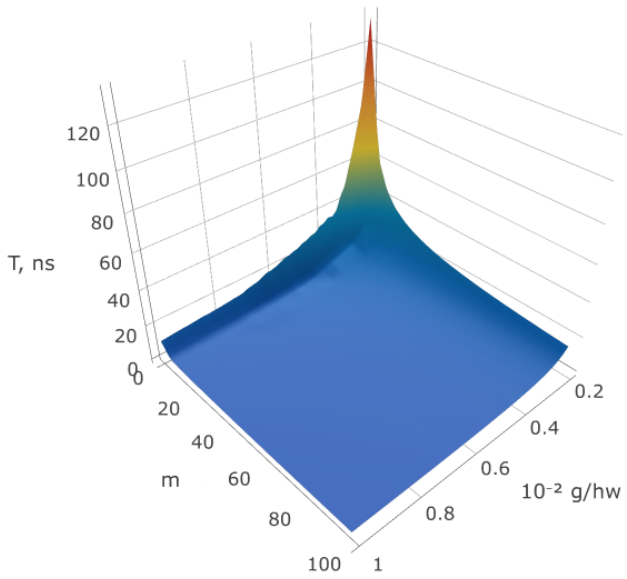
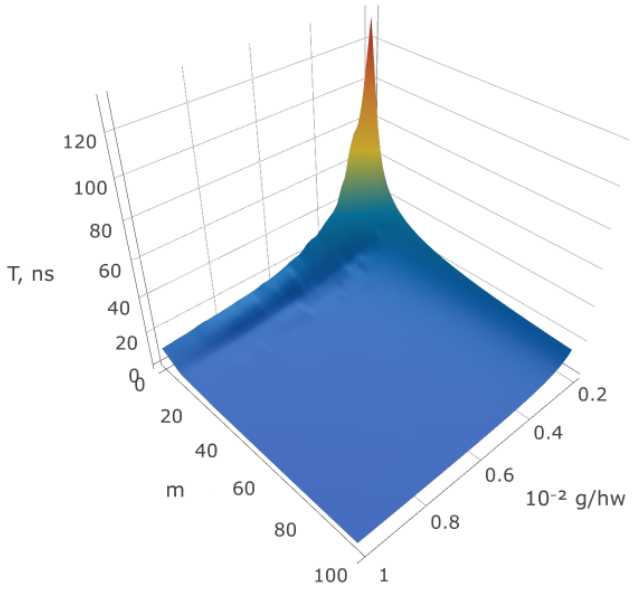


Рис. 6: Наблюдается снижение пиковых амплитуд и нарушение периодичности осцилляций состояний вида $|\Psi_0\rangle = |vac\rangle_{ph}\{0, n\succ_{at}$, $|\Psi_1\rangle = |vac\rangle_{ph}\{n, 0\succ_{at}$ при увеличении числа n атомов в группе.

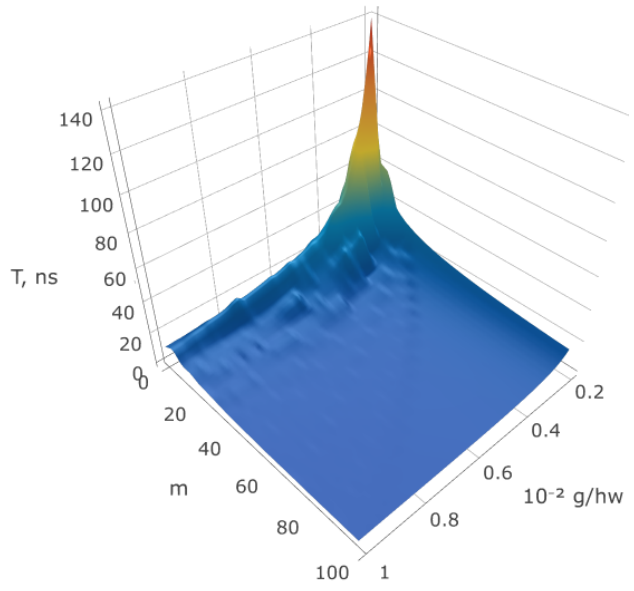
Следующие графики демонстрируют зависимость периода осцилляций состояний вида $|\Psi_0\rangle = |m\rangle_{ph}\{0, n\succ_{at}$, $|\Psi_1\rangle = |m\rangle_{ph}\{n, 0\succ_{at}$ от силы взаимодействия g и первоначального числа m фотонов в полости при различных значениях n.



$n = 10$



$n = 20$



$n = 30$

Увеличение силы взаимодействия g атомов с полем, равно как и увеличение фотонной накачки, приводит к существенному уменьшению периода осцилляций. При стремлении g к нулю период осцилляций стремится к бесконечности (значение $g = 0$ соответствует случаю отсутствия взаимодействия "атом-поле").

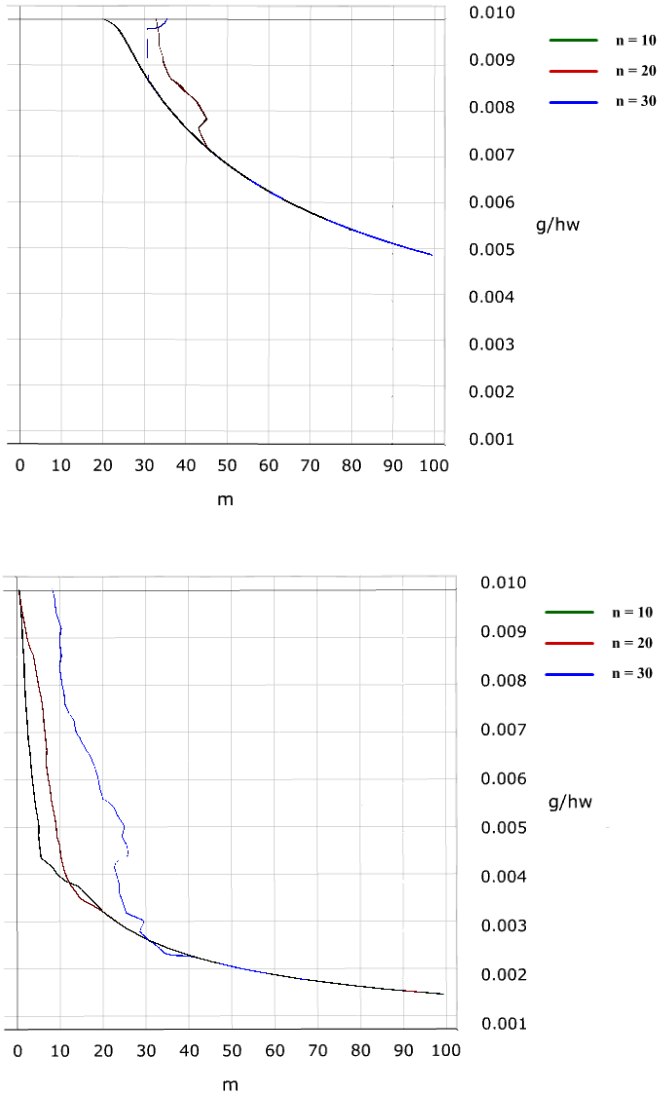


Рис. 7: Удлинение периода осцилляций при уменьшении коэффициента связи g с полем может быть скомпенсировано увеличением числа n атомов и фотонной накачки m.

Представленные изочастотные графики, соответствующие периодам осцилляций $T = 3$ ns и $T = 10$ ns для $n = 10, 20 \;\; 30$ атомов демонстрируют возможность наращивания числа атомов в группе и уменьшения силы взаимодействия "атом-поле"с одновременным сохранением периода осцилляций.

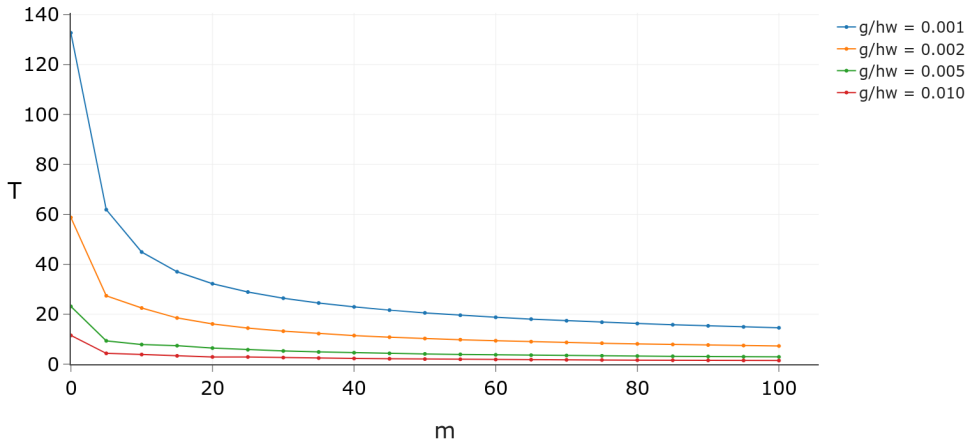


Рис. 8: зависимость периода осцилляций от первоначального числа m фотонов в полости (10 атомов)

начальное состояние: $|\Psi_0\rangle = |m\rangle_{ph}\{0, 10\succ_{at}$

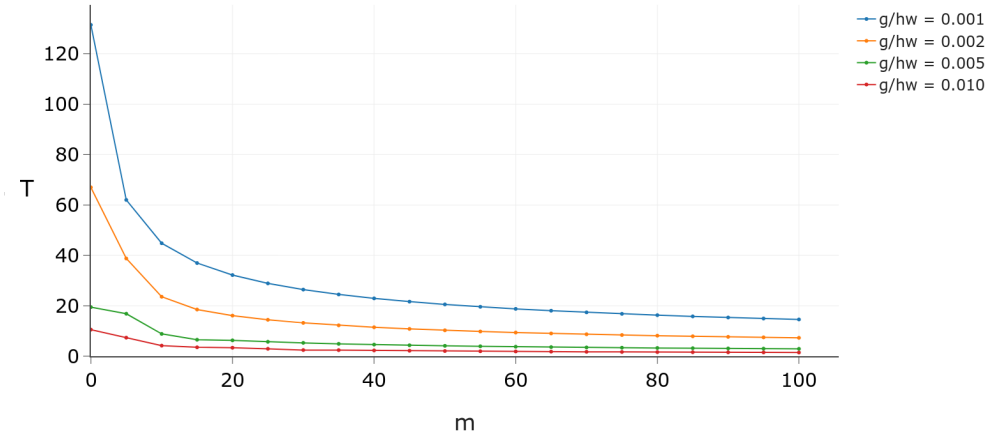


Рис. 9: зависимость периода осцилляций от первоначального числа m фотонов в полости (20 атомов)

начальное состояние: $|\Psi_0\rangle = |m\rangle_{ph}\{0, 20\succ_{at}$

Увеличение фотонной накачки в полости резонатора сопровождается асимптотическим стремлением к нулю периода осцилляций.

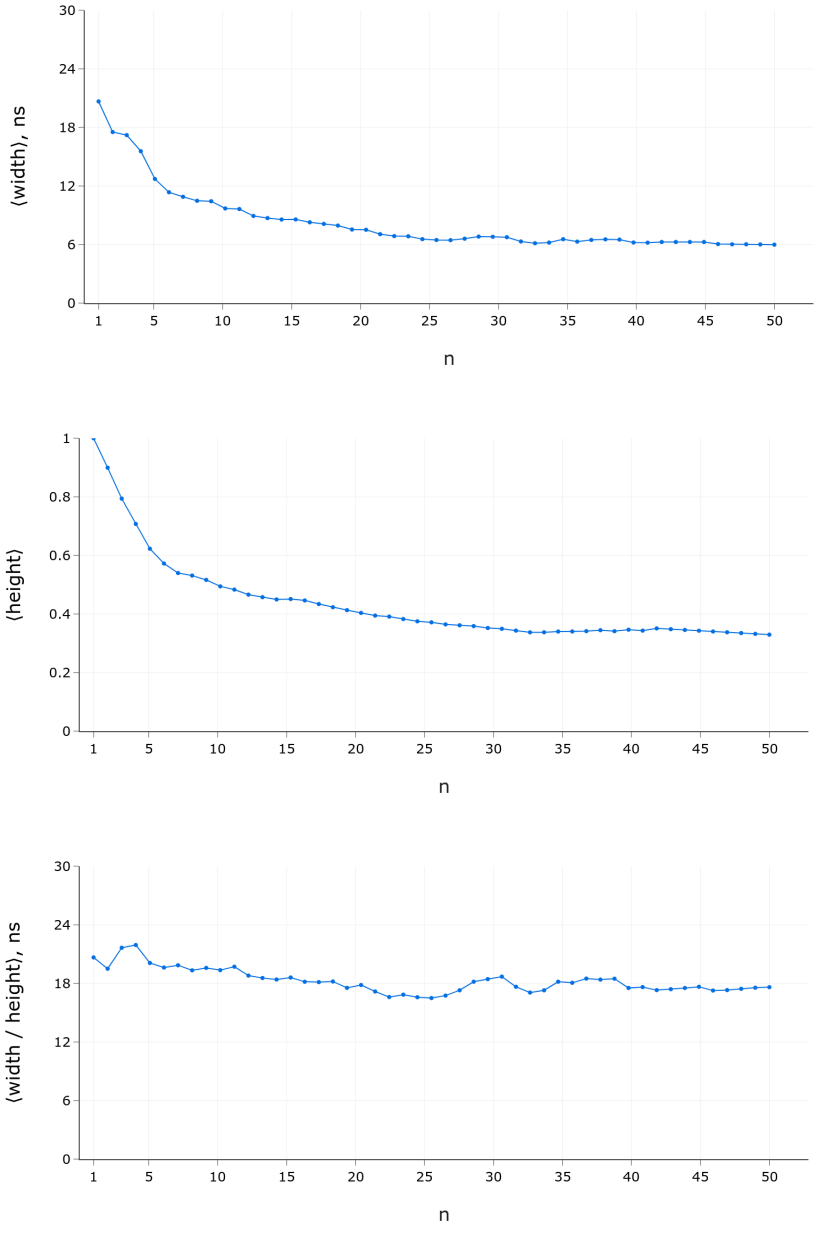


Рис. 10: $\langle$width$\rangle$ – среднее значение ширины основания пика, $\langle$height$\rangle$ – среднее значение высоты пика

квадрата амплитуды начального состояния $|\Psi_0\rangle = |vac\rangle_{ph}\{0, n\succ_{at}$.

Наблюдается уменьшение средних значений ширины основания и высоты пика осцилляций при увеличении числа атомов n в группе. Их отношение близко к константе.

4 Модель Тависа-Каммингса для неидеальной полости

Рассмотрим гамильтониан Тависа-Каммингса в RWA приближении для n двухуровневых атомов, имеющих разные координаты внутри полости. Пусть энергии возбуждения атомов равны $h\omega_a$, и отличаются от энергии фотона полости $h\omega_c$ малыми расстройками $d_i = h\omega_c - h\omega_a$. Гамильтониан такой системы имеет вид

$$H_{TC}^{(RWA)} = h\omega_c a^+ a + h\omega_a \sum_{i=1}^{n} \sigma_i^+ \sigma_i + a\bar{\sigma}^+ + a^+ \bar{\sigma}, \tag{5}$$

где $\bar{\sigma} = \sum\limits_{i=1}^{n} g_i \sigma_i$, $\bar{\sigma}^+ = \sum\limits_{i=1}^{n} g_i \sigma_i^+$ - операторы коллективной релаксации и возбуждения группы атомов, силы взаимодействия которых с полем $g_1, g_2, ..., g_n$, вообще говоря, различны.

Реальный резонатор находится в контакте с внешней средой, которая в простейшем случае имеет вид одномодового фотонного резервуара с фиксированной температурой на моде полости. Контакт предполагает возможность обмена фотонами частоты ω_c между внешней средой и полостью.

Рассмотрим простейший случай нулевой температуры на моде полости, при котором во внешнем резервуаре нет фотонов и такой контакт ведет к постоянной утечке фотонов из полости. Если $\rho(t)$ - матрица плотности системы "атомы + поле" в полости, ее динамика в указанном случае описывается квантовым основным уравнением

$$ih\dot{\rho} = \mathcal{L}(\rho), \quad \mathcal{L}(\rho) = -\frac{i}{h}[H, \rho] + \frac{1}{h}L(\rho), \quad L(\rho) = \gamma\left(a\rho a^+ - \frac{1}{2}\{a^+ a, \rho\}\right) \tag{6}$$

где $H = H_{TC}^{RWA}$, γ – параметр, отвечающий за интенсивность линдбладовского процесса, $L(\rho) = \gamma\big(a\rho a^+ - \frac{1}{2}\{a^+ a, \rho\}\big)$ – оператор Линдблада, соответствующий утечке фотона, которая выражается оператором уничтожения фотона a.

Мы будем исследовать динамику состояния атомов и поля, идущую задолго до наступления термической стабилизации, рассматривая обмен фотонами с внешним резурвуаром как фактор декогерентности.

4.1 Осцилляции с утечкой фотонов из полости

Характерная картина ансамблевых осцилляций сохраняется при условии, когда резонатор находится в контакте с внешней средой, приводящем к постоянной утечке фотонов из полости.

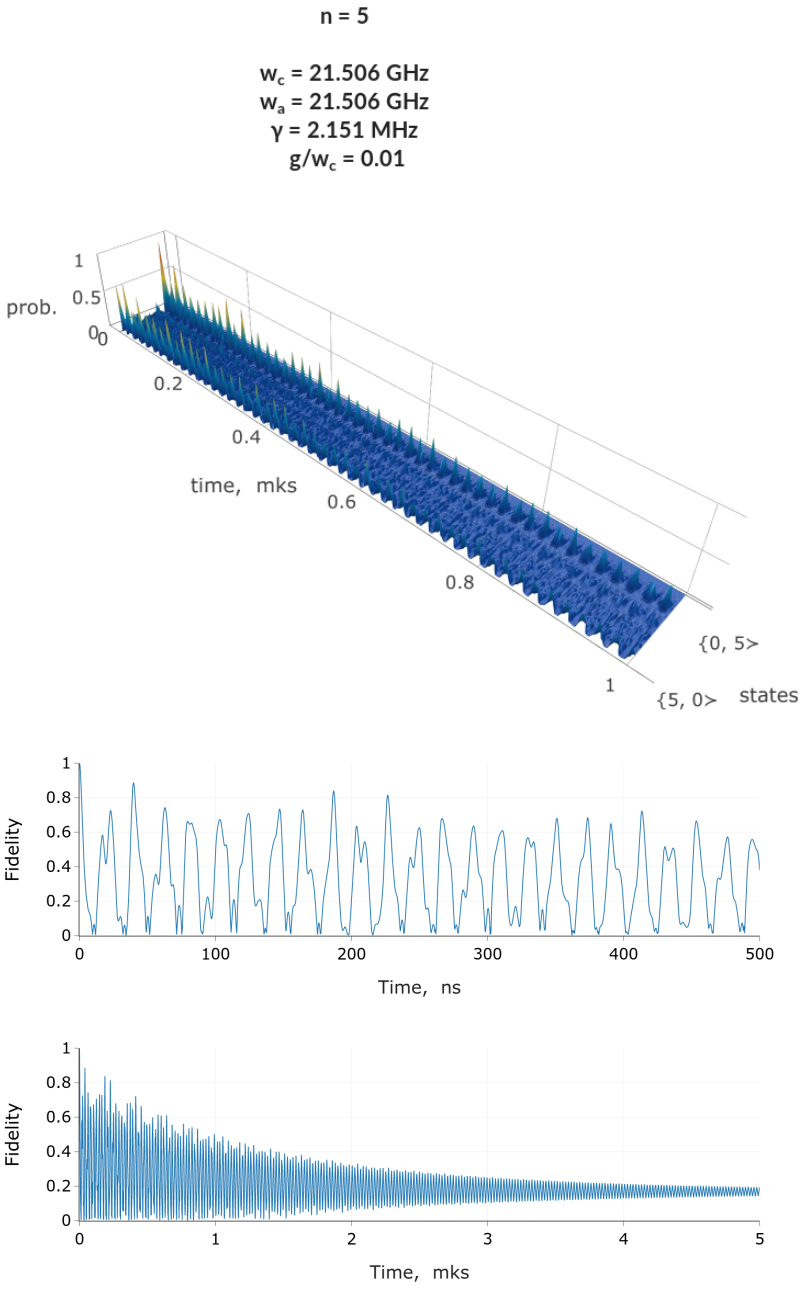
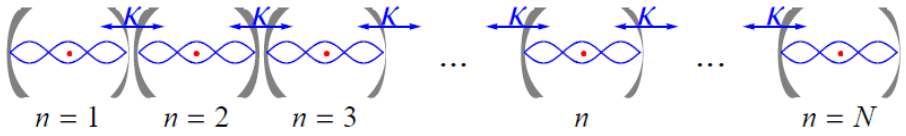


Рис. 11: $|\Psi_0\rangle = |vac\rangle_{ph}\{0, 5\succ_{at}$

5 Модель Тависа-Каммингса-Хаббарда

Модель Тависа-Каммингса можно обобщить на случай двух и более взаимодействующих полостей: фотоны могут перемещаться между полостями посредством оптического волокна.



Такая система называется моделью Тависа-Каммингса-Хаббарда и ее гамильтониан в представлении взаимодействия имеет следующий вид:

$$H_{TCH} = \sum_{j=1}^{J} \left(\underbrace{hw_{c_j}\ a_j^+ a_j}_{H_{field}} + \underbrace{hw_{a_j} \sum_{i=1}^{N} \sigma_{i_j}^+ \sigma_{i_j}}_{H_{atoms}} + \underbrace{g_j \sum_{i=1}^{N} (\sigma_{i_j}^+ + \sigma_{i_j})(a_j^+ + a_j)}_{H_{int}} \right) + \mu \sum_{j=1}^{J} \left(a_{j+1}^+ a_j + a_j^+ a_{j+1} \right) \tag{7}$$

J — количество взаимодействующих полостей

последнее слагаемое отвечает за перелет фотонов между полостями (параметр μ характеризует частоту перелета)

Аналогичным образом определяется гамильтониан ТСН в приближении вращающейся волны:

$$H_{TCH}^{RWA} = \sum_{j=1}^{J} \left(\underbrace{hw_{c_j}\ a_j^+ a_j}_{H_{field}} + \underbrace{hw_{a_j} \sum_{i=1}^{N} \sigma_{i_j}^+ \sigma_{i_j}}_{H_{atoms}} + \underbrace{g_j \sum_{i=1}^{N} (\sigma_{i_j}^+ a_j + \sigma_{i_j} a_j^+)}_{H_{int}} \right) + \mu \sum_{j=1}^{J} \left(a_{j+1}^+ a_j + a_j^+ a_{j+1} \right) \tag{8}$$

6 Ансамблевые осцилляции между полостями

Группы атомов A_1 и A_2, распределенные между полостями, дают хорошие осцилляции при достаточно большой амплитуде μ перехода возбуждения ($\mu > g$).

Моделирование следующей динамики проводилось для 10 атомов в каждой полости при $\mu \approx 3.5g$.

Начальное состояние $|\Psi_0\rangle = |vac\rangle_{ph_I}|vac\rangle_{ph_{II}}\{0, n\succ_{at}$, где

$|vac\rangle_{ph_I}$ – начальное вакуумное состояние поля I полости,

$|vac\rangle_{ph_{II}}$ – начальное вакуумное состояние поля II полости,

$\{i, j\succ_{at} = \{i\succ_I\{j\succ_{II}$ – состояние атомных групп A_1, A_2.

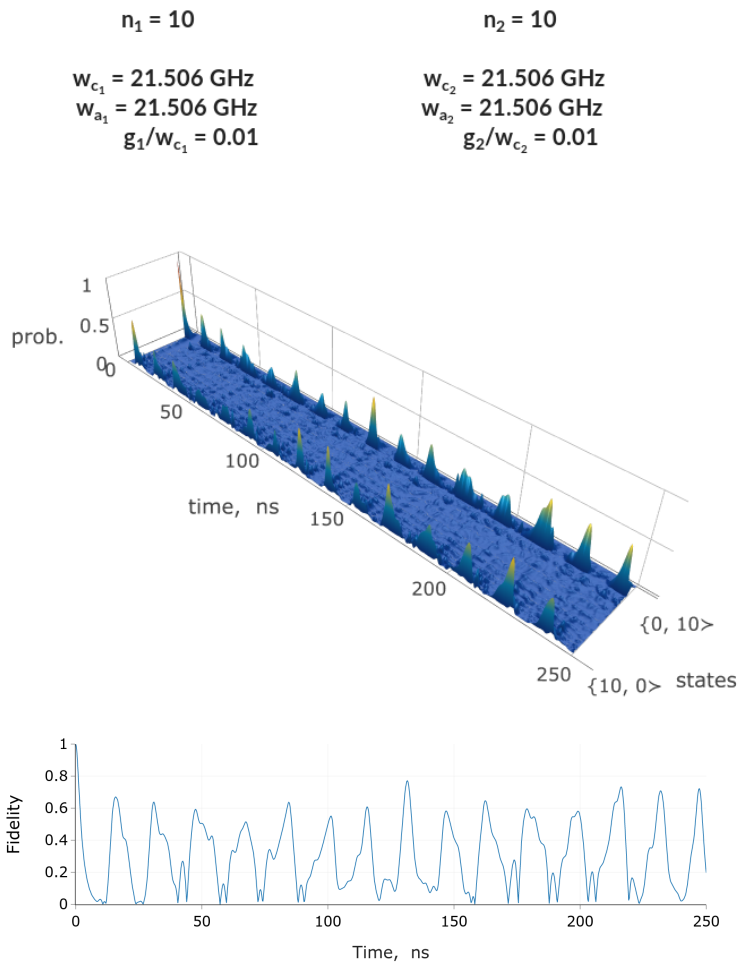


Рис. 12: $|\psi_0\rangle = |vac\rangle_{ph_I}|vac\rangle_{ph_{II}}\{0\succ_I\{10\succ_{II}$

коллапсы и возрождения ансамблевых состояний между полостями высокого качества

Для числа атомов, равного 20, была численно найдена граница возникновения осцилляций: амплитуда перехода возбуждения между полостями, равная $\mu \approx 5g$.

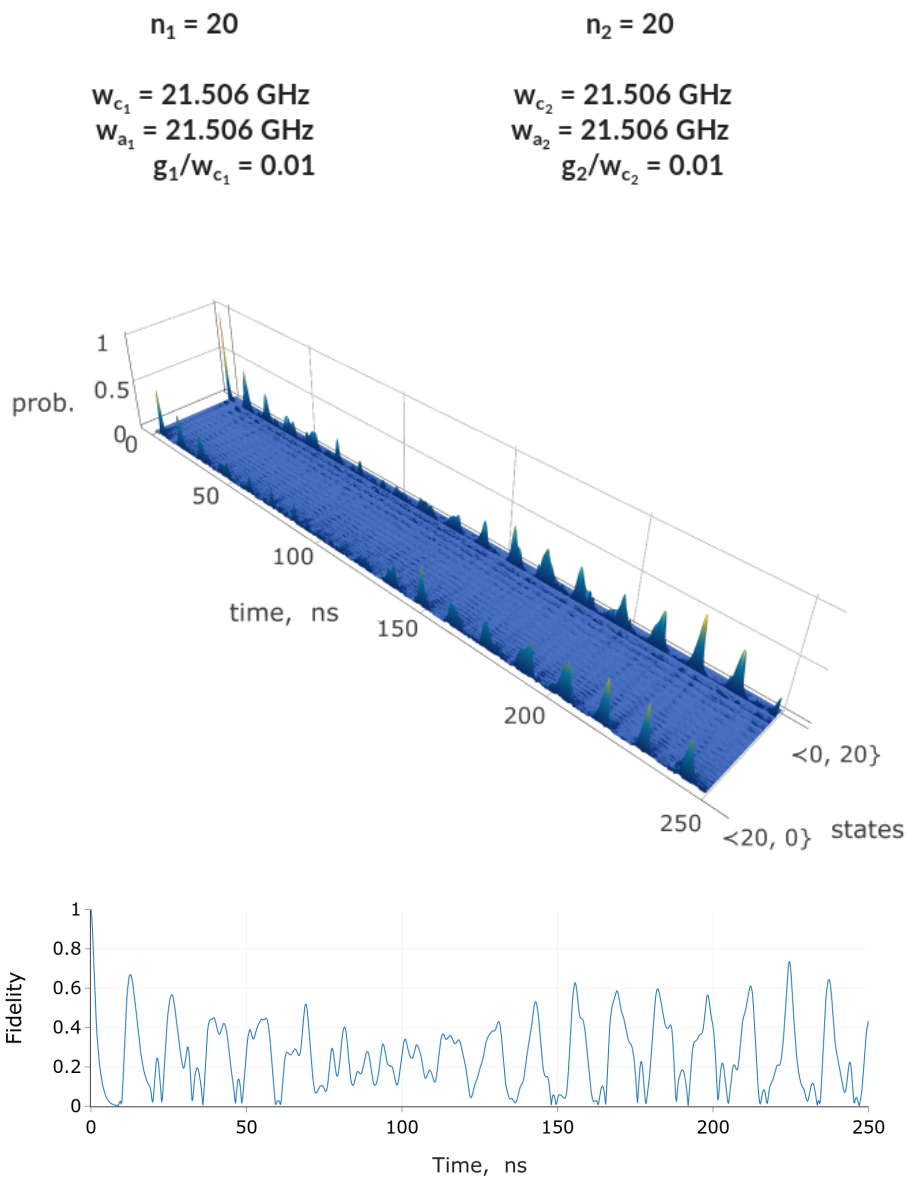


Рис. 13: $|\psi_0\rangle = |vac\rangle_{ph_I}|vac\rangle_{ph_{II}}\{0\succ_I\{20\succ_{II}$

высокое качество осцилляций, близкое к 0.8,
коллапсы и возрождения ансамблевых состояний

7 Заключение

В настоящей работе была рассмотрена динамика квантовых состояний ансамбля двухуровневых атомов и одномодового поля в резонаторе в рамках модели Тависа-Каммингса в RWA приближении с шумами, а также в рамках модели Тависа-Каммингса-Хаббарда (RWA) для случая двух взаимодействующих полостей.

- Установлен резкий характер осцилляций между двумя группами атомов одинаковой численности и одинаковой силы взаимодействия с полем.
- Резкость осцилляций в ансамбле с четным числом двухуровневых атомов предсказуемо растет с увеличением фотонной накачки полости и намного превосходит резкость обычных рабиевских осцилляций.
- Численно найдена зависимость осцилляций от разброса значений силы связи с полем. Показано, что удлинение периода этих осцилляций при уменьшении коэффициента связи с полем может быть скомпенсировано увеличением числа атомов и фотонной накачки.
- Обнаружено хорошо регистрируемое квантовое "эхо": переход состояния атомов из одной полости в другую и возврат этого состояния в первоначальную полость, что имеет место для произвольного ансамблевого состояния.

  Установлено высокое качество такого эха и его зависимость от амплитуды перехода между полостями. В частности, была установлена граница возникновения осцилляций, равная $\mu \approx 3.5g$ для 10 атомов и $\mu \approx 5g$ для 20 атомов в каждой группе.

Результаты для ансамблей из 20-50 атомов в половине ансамбля, необходимые для уверенной фиксации выявленного характера осцилляций, требовали моделирования динамики как чистых, так и смешанных квантовых состояний в пространстве около 40 кубитов, не могут быть получены на рабочей станции. Для проведения данных численных экспериментов использовалось моделирование динамики на GPU суперкомпьютера Ломоносов с применением технологии параллельных вычислений CUDA, а также пакета матричной алгебры MAGMA [5].

Автор выражает благодарность Российскому Фонду Фундаментальных исследований за грант а-18-01-00695.

8 Приложение

Листинг параллельной программы (динамика ансамблевых состояний модели Тависа-Каммингса)

Полный код последовательной и параллельной версий программных реализаций моделей Тависа-Каммингса и Тависа-Каммингса-Хаббарда доступен по адресу `https://github.com/alexfmsu/Quantum_Mechanics`.

Bipartite.cpp

```
// ------------------------------------------------------------------------------------------
// system
#include <iostream>
#include <cstring>
#include <cmath>

#include "magma-2.3.0/include/magma_v2.h"
// ------------------------------------------------------------------------------------------
// BipartiteGeneral
#include "Cavity.h"
#include "Hamiltonian.h"

#include "WaveFunction.h"
#include "DensityMatrix.cpp"

#include "Evolution.cpp"

#include "parser.cpp" // config
// ------------------------------------------------------------------------------------------
// Common
#include "ext.cpp"
// ------------------------------------------------------------------------------------------

using namespace std;

// ------------------------------------------------------------------------------------------
int main(int argc, char** argv) {
// ------------------------------------------------------------------------------------------
magma_init();

magma_queue_t queue = NULL;
magma_int_t dev = 0;

magma_queue_create(dev, &queue);
// ------------------------------------------------------------------------------------------
const char* config_filename = (argc > 1) ? (argv[1]) : ("Bipartite/config");

BGconfig config;
config.read_from_file(config_filename);
print("Parse config... done\n\n", "yellow");

cout << '[' << config.init_state()->first << "," << config.init_state()->second << "]" << endl;
// ------------------------------------------------------------------------------------------
mkdir(config.path());

string cmd = "cp " + string(config_filename) + " " + config.path() + "/config.py";

if (system(cmd.c_str())) {
Assert(false, "Error: system(cmd.c_str())", __FILE__, __LINE__);
}
// ------------------------------------------------------------------------------------------
```

```
Cavity cavity(config.n(), config.wc(), config.wa(), config.g());

cavity.print_n();
cavity.print_wc();
cavity.print_wa();
cavity.print_g();
// ------------------------------------------------------------------------------------------------
Hamiltonian H(config.capacity(), &cavity, &queue);
print("Hamiltonian... done\n\n", "yellow");

H.print_states();

#if DEBUG
// H.write_to_file(config.H_csv(), config.precision());
// H.print_bin_states();
// H.print_html();
#endif
// ------------------------------------------------------------------------------------------------
DensityMatrix ro_0(&H, H.states());

cout << "config.T = " << config.T() << endl;
cout << "config.nt = " << config.nt() << endl;
cout << "config.dt = " << config.dt() << endl;

ro_0.set_ampl(config.init_state(), MAGMA_Z_MAKE(1, 0));
print("Density matrix... done\n\n", "yellow");

#if DEBUG
// // ro_0.write_to_file(config.ro0_csv, config.precision());
#endif
// ------------------------------------------------------------------------------------------------

// run(ro_0, H, config.dt(), config, true);
// print("Evolution... done\n\n", "yellow");

WaveFunction wf(H.states());
wf.set_amplitude(config.init_state(), 1.0);

wf.print();

run_wf(wf, H, config.dt(), config, true);

print("\nSUCCESS\n");
// ------------------------------------------------------------------------------------------------
magma_queue_destroy(queue);

magma_finalize();
// ------------------------------------------------------------------------------------------------
return 0;
}
```

Cavity.h

```
// ------------------------------------------------------------------------------------------------
#pragma once
// ------------------------------------------------------------------------------------------------
// Common
#include "../Common/Assert.cpp"
#include "../Common/Print.cpp"
#include "../Common/STR.cpp"
// ------------------------------------------------------------------------------------------------
```

```
// ------------------------------------------------------------------------------------------
class Cavity {

public:
    // BEGIN---------------------------------- CONSTRUCTOR ------------------------------------
    Cavity(const int _n, const double _wc, const double _wa, const double _g) {
        Assert(_n > 0, "n <= 0", __FILE__, __LINE__);
        n_ = _n;

        Assert(_wc > 0, "wc <= 0", __FILE__, __LINE__);
        wc_ = _wc;

        Assert(_wa > 0, "wa <= 0", __FILE__, __LINE__);
        wa_ = _wa;

        Assert(_g > 0, "g <= 0", __FILE__, __LINE__);
        g_ = _g;
    }
    // END------------------------------------ CONSTRUCTOR ------------------------------------

    Cavity operator=(const Cavity& _cv) {
        return Cavity(_cv.n_, _cv.wc_, _cv.wa_, _cv.g_);
    }

    // BEGIN---------------------------------- GETTERS -----------------------------------------
    int     n() const { return  n_; }
    double wc() const { return wc_; }
    double wa() const { return wa_; }
    double  g() const { return  g_; }
    // END------------------------------------ GETTERS -----------------------------------------

    // BEGIN---------------------------------- PRINT -------------------------------------------
    void print_n() const {
        print(" n: ", "green");

        print(n_);

        println(2);
    }

    void print_wc() const {
        print("wc: ", "green");

        print(wc_str(wc_));

        println(2);
    }

    void print_wa() const {
        print("wa: ", "green");

        print(wa_str(wa_));

        println(2);
    }

    void print_g() const {
        print(" g: ", "green");

        print(g_str(g_));

        println(2);
    }
    // END------------------------------------ PRINT -------------------------------------------
```

```
    // BEGIN---------------------------------- DESTRUCTOR ---------------------------------------
    ~Cavity() {}
    // END------------------------------------ DESTRUCTOR ---------------------------------------

private:
    int n_;

    double wc_;
    double wa_;

    double g_;
};
// ------------------------------------------------------------------------------------------------
```

Hamiltonian.h

```
#include <iostream>
#include <string>
#include <map>

#include "matrix.h"

#include "Cavity.h"
#include "CV_state.h"

using namespace std;

typedef pair<int, int> BinState;
// ---------------------------
double a_cross(const int ph) {
    return sqrt(ph + 1);
}

double a(const int ph) {
    return sqrt(ph);
}
// ---------------------------

typedef pair<int, int> point;


class Hamiltonian: public Matrix {

public:
    vector<std::string> bin_states_keys;
    map<std::string, vector<int> > bin_states_;

    Hamiltonian(int _capacity, Cavity* _cv, magma_queue_t* queue) : Matrix() {
        // -------------------
        Assert(_capacity > 0, "capacity <= 0", __FILE__, __LINE__);
        capacity_ = _capacity;

        cv_ = _cv;

        queue_ = queue;

        // init_states();
        // -------------
        int M = capacity_;
        int n = cv_->n();

        int _min = std::min(M, n);
```

```
    int count = 0;
    // -------------
    for (int i1 = 0; i1 <= _min; i1++) {
        for (int i2 = 0; i2 <= min(n, M - i1); i2++) {
            states_[count] = make_pair(i1, i2);

            count++;
        }
    }

    int dime = states_.size();

    M_ = N_ = dime;
    size_ = M_ * N_;

    cpu_alloc();
    gpu_alloc();

    double wc = cv_->wc();
    double wa = cv_->wa();
    double g = cv_->g();

    int i = 1, j;
    count = 0;

    std::pair<int, int> p;

    // FROM PYTHON
    for (int i1 = 0; i1 <= _min; i1++) {
        for (int i2 = 0; i2 <= min(n, M - i1); i2++) {
            j = 1;

            for (int j1 = 0; j1 <= _min; j1++) {
                for (int j2 = 0; j2 <= min(n, M - j1); j2++) {
                    if (i1 != j1) {
                        p = make_pair(i1, j1);
                    } else if (i2 != j2) {
                        p = make_pair(i2, j2);
                    } else {
                        p = make_pair(1, 2);
                    }

                    int mi = min(p.first, p.second);

                    double kappa = sqrt((n - mi) * (mi + 1));

                    if (abs(i1 - j1) + abs(i2 - j2) == 1) {
                        set(i - 1, j - 1, g * sqrt(max(M - i1 - i2, M - j1 - j2)) * kappa);
                    } else if (abs(i1 - j1) + abs(i2 - j2) == 0) {
                        set(i - 1, j - 1, (M - (i1 + i2)) * wc + (i1 + i2) * wa);
                    } else {
                        set(i - 1, j - 1, 0);
                    }

                    j++;
                }
            }

            i++;
        }
    }

    cout << states_.size() << endl;
}
```

```
    // BEGIN---------------------------------- GETTERS ----------------------------------------
    double capacity() const { return capacity_; }

    Cavity* cavity() { return cv_; }
    // END---------------------------------- GETTERS ----------------------------------------
    void init_states() {
        int n = cv_->n();

        int k = 0;

        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= capacity_ - i; j++) {
                states_[k++] = make_pair(i, j);
            }
        }
    }

    map<int, point>* states() {
        return &states_;
    }

    // ------------------------------------------------------------------------------------------

    void print_states() {
        cout << "STATES:" << endl;

        for (auto k : states_) {
            cout << "[" << k.second.first << " " << k.second.second << "]" << endl;
            // k.second.print();
        }
    }
    // ------------------------------------------------------------------------------------------

private:
    int capacity_;

    Cavity* cv_;

    map<int, point> states_;
};
```

WaveFunction.h

```
#pragma once

#include <omp.h>

typedef pair<int, int> point;

class WaveFunction : public Matrix {

public:
WaveFunction(map<int, point>* states) : Matrix() {
size_ = M_ = states->size();
N_ = 1;

cpu_alloc();
gpu_alloc();

states_ = states;
}

WaveFunction(WaveFunction& wf) {
```

```
size_ = M_ = wf.M_;
N_ = 1;

cpu_alloc();
gpu_alloc();

memcpy(cpu_data_, wf.cpu_data(), size_ * sizeof(magmaDoubleComplex));

states_ = wf.states_;
}

void set_amplitude(point* state, double val) {
for (auto k : *states_) {
if (k.second == *state) {
set(k.first, 0, Complexd(1, 0));

return;
}
}

cout << "ERROR\n";
}

void print(int precision = 3) const {
double eps = 1.0 / pow(10, precision);

for (int i = 0; i < M_; i++) {
Complexd v = get(i, 0);

double re = v.real();
double im = v.imag();

if (abs(re) < eps) {
re = 0;
}

if (abs(im) < eps) {
im = 0;
}

cout << fixed << setprecision(precision) << "(" << re << "," << im << ")" << endl;
}

cout << endl;
}

void friend GEMV(Matrix* A, Matrix* B, Matrix* C, magma_queue_t queue, Complexd* alpha,
Complexd* beta, const char _a = 'N', const char _b = 'N') {
magmaDoubleComplex Alpha = MAGMA_Z_MAKE(1, 0);
magmaDoubleComplex Beta = MAGMA_Z_MAKE(0, 0);

Matrix* B_tmp;

if (B == C) {
B_tmp = new Matrix(B->M(), B->N());
memcpy(B_tmp->cpu_data(), B->cpu_data(), B_tmp->M() * B_tmp->N() * sizeof(magmaDoubleComplex));
} else {
B_tmp = B;
}

magma_zsetmatrix (A->M(), A->N(), A->cpu_data(), A->M() , A->gpu_data(), A->M(), queue);

magma_zsetvector (A->N() , B_tmp->cpu_data(), 1 , B_tmp->gpu_data() , 1 , queue);
magma_zsetvector (A->M() , C->cpu_data() , 1 , C->gpu_data() , 1 , queue);
```

```
magma_zgemv( MagmaNoTrans, A->M(), A->N(), Alpha, A->gpu_data(), A->M(),
B_tmp->gpu_data(), 1, Beta, C->gpu_data(), 1, queue);

magma_zgetvector (A->M() , C->gpu_data(), 1 , C->cpu_data() , 1 , queue);

if (B == C) {
delete B_tmp;
}
}

double norm() {
// magmaDoubleComplex norm;
// Complexd norm(0, 0);
double norm_d = 0;

int i;
// #pragma omp parallel private(i) reduction(+:norm_d)
for (i = 0; i < M_; i++) {
norm_d += (get(i, 0).conj() * get(i, 0)).abs();
}

return norm_d;
// return MAGMA_Z_ABS(norm);
}

vector<double> diag() {
vector<double> diag_;

Complexd val(0, 0);

for (int i = 0; i < M_; i++) {
val = get(i, 0);

diag_.push_back((val.conj() * val).abs());
}

return diag_;
}

void print_diag() {
vector<double> diag_ = diag();

for (int i = 0; i < M_; i++) {
cout << diag_[i] << " ";
}

cout << endl;
}

~WaveFunction() {
magma_free_pinned(cpu_data_);
magma_free(gpu_data_);
}

private:
map<int, point>* states_;
};
```

DensityMatrix.cpp

```
// ------------------------------------------------------------------------------------------
#pragma once
// ------------------------------------------------------------------------------------------
```

```
#include "matrix.h"
// ------------------------------------------------------------------------------------------------

using namespace std;

// ------------------------------------------------------------------------------------------------
class DensityMatrix: public Matrix {

public:
    // BEGIN------------------------------------ CONSTRUCTOR ------------------------------------
    DensityMatrix(Matrix* _H, map<int, point>* _states) : Matrix(_H->M(), _H->N()) {
        states_ = _states;
    }
    // END-------------------------------------- CONSTRUCTOR ------------------------------------

    // ------------------------------------------------------------------------------------------------
    void set_ampl(point* _state, Complexd _val) {
        bool found = false;

        for (auto k : *states_) {
            bool eq = (k.second == *_state);

            if (eq) {
                found = true;

                set(k.first, k.first, _val);

                break;
            }
        }

        Assert(found, "not found", __FILE__, __LINE__);
    }
    // ------------------------------------------------------------------------------------------------

    // ------------------------------------------------------------------------------------------------
    double norm(vector<Complexd>& diag_) {
        double norm_ = 0;

        for (int i = 0; i < diag_.size(); i++) {
            norm_ += diag_[i].abs();
        }

        return norm_;
    }

    double norm() {
        double norm_ = 0;

        vector<Complexd> diag_ = diag();

        for (int i = 0; i < diag_.size(); i++) {
            norm_ += diag_[i].abs();
        }

        return norm_;
    }
    // ------------------------------------------------------------------------------------------------

    // BEGIN------------------------------------ DESTRUCTOR ------------------------------------
    ~DensityMatrix() {}
    // END-------------------------------------- DESTRUCTOR ------------------------------------

private:
    map<int, point>* states_;
```

```
};
// ----------------------------------------------------------------------------------------------
```

Unitary.h

```
#pragma once

#include <iostream>

#include "matrix.h"
#include "Types.h"

# диагонализация гамильтониана и вычисление матричной экспоненты
Matrix getU(Matrix* H, double dt) {
    int M = H->M();
    int N = H->N();

    magma_int_t n2 = M * N;

    Matrix a(M, N);

    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            a.set(i, j, H->get(i, j));
        }
    }

    double *s1;
    magma_int_t info ;
    magma_int_t ione = 1;
    double error = 1.;
    double mone = -1.0;
    magma_int_t lwork;
    magma_int_t min_mn = min(M, N);

    double* rwork;

    magma_int_t* iwork = new magma_int_t[N];
    magma_int_t liwork = 3 + 5 * N;

    magma_dmalloc_cpu(&s1 , min_mn );


    magma_int_t nb = magma_get_zhetrd_nb(N);

    lwork = max( N + N * nb, 2 * N + N * N );
    magma_int_t lrwork = 1 + 5 * N + 2 * N * N;

    magma_dmalloc_pinned (& rwork , lrwork);
    complexd* work;

    magma_zmalloc_pinned(&work, lwork);

    magma_int_t aux_iwork[1], lda = N;
    magmaDoubleComplex aux_work[1];
    double aux_rwork[1];

    magma_zheevd( MagmaVec, MagmaLower,
                  N, NULL,
                  lda, NULL,
                  aux_work,  -1,
                  aux_rwork, -1,
                  aux_iwork, -1,
```

```
            &info );

lwork = (magma_int_t) MAGMA_Z_REAL( aux_work[0] );
lrwork = (magma_int_t) aux_rwork[0];
liwork = aux_iwork[0];

magmaDoubleComplex *h_A, *h_R, *h_work;
double *w1, *w2, result[3], eps;

magma_queue_t queue = NULL ;
magma_device_t device;
magma_queue_create(device, &queue);

magma_zmalloc_cpu(&h_A, N * lda);
magma_zsetmatrix(H->M(), H->N(), H->cpu_data(), H->M(), h_R, H->M(), queue);

magma_dmalloc_cpu(&w1, N);
magma_dmalloc_cpu(&w2, N);
magma_dmalloc_cpu(&rwork, lrwork);
magma_imalloc_cpu(&iwork, liwork);

magma_zmalloc_pinned(&h_R, N * lda);

magma_zmalloc_pinned(&h_work, lwork);

magma_zheevd( MagmaVec, MagmaLower,
              N, a.cpu_data(), lda, w1,
              h_work, lwork,
              rwork, lrwork,
              iwork, liwork,
              &info );


Matrix V(M, M);
Matrix D(M, M);

Matrix VD(M, M);
Matrix U(M, M);

for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        V.set(i, j, a.get(i, j));
    }

    D.set(i, i, w1[i]);
}

D.exp_diag(Complexd(0, -dt));

Complexd alpha = cONE;
Complexd beta = cZERO;

GEMM(&V, &D, &VD, H->queue(), &alpha, &beta, 'N', 'N');
GEMM(&VD, &V, &U, H->queue(), &alpha, &beta, 'N', 'C');

free(s1);
free(w1);
free(w2);
free(rwork);
free(iwork);

magma_free_pinned(h_work);
magma_free_pinned(h_R);

return U;
```

```
}
```

Evolution.cpp

```
// ------------------------------------------------------------------------------------------------
#pragma once
// ------------------------------------------------------------------------------------------------
// Common
#include "ext.cpp"
#include "Fidelity.cpp"
// ------------------------------------------------------------------------------------------------
#include "parser.cpp"
#include "Unitary.h"
#include <sstream>
// ------------------------------------------------------------------------------------------------


// ------------------------------------------------------------------------------------------------
void run(DensityMatrix& ro_0, Hamiltonian& H, double dt, BGconfig& config, bool fidelity_mode = false) {
cout << "dt = " << dt << endl;

// оператор эволюции
Matrix U = getU(&H, dt);

#if DEBUG
U.write_to_file(config.U_csv(), config.precision());
#endif

File f(config.z_csv());

f.erase();
f.open("w");

vector<Complexd> diag_;

map<std::string, double> p_bin;
vector<double> v_bin;

Complexd alpha = cONE;
Complexd beta = cZERO;

vector<double> FIDELITY;

Matrix ro_0_sqrt = ro_0.get_sqrt();

DensityMatrix ro_t(ro_0);

// Matrix* A_tmp;
// A_tmp = new Matrix(ro_t.M(), ro_t.N());

// Matrix* B_tmp;
// B_tmp = new Matrix(ro_t.M(), ro_t.N());

// Реализация схемы Эйлера с шагом по времени dt
for (int i = 0; i < config.nt(); i++) {
if (fidelity_mode) {
double fidelity_t = fidelity(&ro_0_sqrt, &ro_t, H.queue());
FIDELITY.push_back(fidelity_t);
}

diag_ = ro_t.diag();

double diag_norm = ro_t.norm(diag_);
```

```
if (abs(1 - diag_norm) > 1) {
cout << "Not normed" << endl;
cout << "i = " << i << ", " << diag_norm << endl;

// string cmd = "if [ -d '" + config.path() + "' ]; then rm -r " + config.path() + "; fi";

// int _sys = system(cmd.c_str());

exit(1);
}

f.write(diag_, config.precision());

GEMM(&U, &ro_t, &ro_t, H.queue(), &alpha, &beta, 'N', 'N');
GEMM(&ro_t, &U, &ro_t, H.queue(), &alpha, &beta, 'N', 'C');

// memcpy(B_tmp->cpu_data(), ro_t.cpu_data(), B_tmp->M() * B_tmp->N() * sizeof(magmaDoubleComplex));
// GEMM(&U, B_tmp, &ro_t, H.queue(), &alpha, &beta);
// GEMM(&U, &ro_t, &ro_t, H.queue(), &alpha, &beta);

// memcpy(A_tmp->cpu_data(), ro_t.cpu_data(), A_tmp->M() * A_tmp->N() * sizeof(magmaDoubleComplex));
// GEMM(A_tmp, &U, &ro_t, H.queue(), &alpha, &beta, 'N', 'C');
// GEMM(&ro_t, &U, &ro_t, H.queue(), &alpha, &beta, 'N', 'C');

}

// delete A_tmp;
// delete B_tmp;

f.close();
// ------------------------------------------------------------------------------------------
vector<std::string> st;

for (auto k : *H.states()) {
std::stringstream ss;

if (k.second == make_pair(0, H.cavity()->n()) || k.second == make_pair(H.cavity()->n(), 0)) {
ss << "[" << k.second.first << ", " << k.second.second << "]";
} else {
ss << "";
}

st.push_back(ss.str());
}

write_states(config.x_csv(), st);

write_t(config.t_csv(), T_str_v(config.T()), config.nt());
// ------------------------------------------------------------------------------------------
if (fidelity_mode) {
File f_fid_csv(config.fid_csv());
f_fid_csv.open("w");

f_fid_csv.writeln("fidelity");

for (size_t i = 0; i < FIDELITY.size(); i++) {
f_fid_csv.writeln(FIDELITY[i], config.precision());
}

f_fid_csv.close();
}
// ------------------------------------------------------------------------------------------
}
// ------------------------------------------------------------------------------------------
```

```
// ------------------------------------------------------------------------------------------
void run_wf(WaveFunction& wf_0, Hamiltonian& H, double dt, BGconfig& config, bool fidelity_mode = false) {
Matrix U = getU(&H, dt);

// #if DEBUG
// U.write_to_file(config.U_csv(), config.precision());
// #endif

File f(config.z_csv());

f.erase();
f.open("w");

vector<double> diag_;

map<std::string, double> p_bin;
vector<double> v_bin;

Complexd alpha = cONE;
Complexd beta = cZERO;

vector<double> z_0;
vector<double> z_1;
vector<double> FIDELITY;
vector<double> FIDELITY_SMALL;

WaveFunction wf_t(wf_0);

double dt_ = config.nt() / (config.T() / 1e-6) / 20000 * 1000;
int nt_ = int(config.nt() / dt_);

// H.states();
int ind_0 = -1, ind_1 = -1;

for (auto k : *H.states()) {
if (k.second == make_pair(0, (H.cavity())->n())) {
ind_0 = k.first;
} else if (k.second == make_pair((H.cavity())->n(), 0)) {
ind_1 = k.first;
}
}

for (int i = 0; i <= config.nt(); i++) {
if (fidelity_mode) {
double fidelity_t = fidelity_wf(&wf_0, &wf_t);

FIDELITY.push_back(fidelity_t);

if (i % nt_ == 0) {
FIDELITY_SMALL.push_back(fidelity_t);
}
}

double diag_norm = wf_t.norm();
diag_ = wf_t.diag();

if (abs(1 - diag_norm) > 0.1) {
cout << "Not normed" << endl;
cout << "i = " << i << ", " << diag_norm << endl;

string cmd = "if [ -d '" + config.path() + "' ]; then rm -r " + config.path() + "; fi";

int _sys = system(cmd.c_str());
```

```
exit(1);
}

// if (config.T() > 0.5 * config.mks()) {
//  if (i % nt_ == 0) {
//  f.write(diag_, config.precision());
//  }
// } else {
//  f.write(diag_, config.precision());
// }

z_0.push_back(diag_[ind_0]);
z_1.push_back(diag_[ind_1]);


GEMV(&U, &wf_t, &wf_t, H.queue(), &alpha, &beta, 'N', 'N');
}

f.close();
// ------------------------------------------------------------------------------------------
vector<std::string> st;

for (auto k : *H.states()) {
std::stringstream ss;

if (k.second == make_pair(0, H.cavity()->n()) || k.second == make_pair(H.cavity()->n(), 0)) {
ss << "[" << k.second.first << ", " << k.second.second << "]";
} else {
ss << "";
}

st.push_back(ss.str());
}

write_states(config.x_csv(), st);

write_t(config.t_csv(), T_str_v(config.T()), config.nt());
// ------------------------------------------------------------------------------------------
File z_0_csv(config.path() + "/z_0.csv");
z_0_csv.open("w");

z_0_csv.writeln("fidelity");

for (size_t i = 0; i < z_0.size(); i++) {
z_0_csv.writeln(z_0[i], config.precision());
}

z_0_csv.close();
// ------------------------------------------------------------------------------------------
File z_1_csv(config.path() + "/z_1.csv");
z_1_csv.open("w");

z_1_csv.writeln("fidelity");

for (size_t i = 0; i < z_1.size(); i++) {
z_1_csv.writeln(z_1[i], config.precision());
}

z_1_csv.close();
// ------------------------------------------------------------------------------------------
if (fidelity_mode) {
File f_fid_csv(config.fid_csv());
f_fid_csv.open("w");
```

```
f_fid_csv.writeln("fidelity");

for (size_t i = 0; i < FIDELITY.size(); i++) {
f_fid_csv.writeln(FIDELITY[i], config.precision());
}

f_fid_csv.close();
// ---------------------------------------- -
File f_fid_small_csv(config.path() + "fid_small.csv");
f_fid_small_csv.open("w");

f_fid_small_csv.writeln("fidelity");

for (size_t i = 0; i < FIDELITY_SMALL.size(); i++) {
f_fid_small_csv.writeln(FIDELITY_SMALL[i], 5);
}

f_fid_small_csv.close();
}
// ------------------------------------------------------------------------------------------
}
// ------------------------------------------------------------------------------------------
```

Список литературы

[1] Х. Бройер, Ф. Петруччионе, Теория открытых квантовых систем, 2010.

[2] Y.I. Ozhigov, N.A. Skovoroda, Conductivity measurements in JCH like models, 2018.

[3] Y.I. Ozhigov, N.A. Skovoroda, Dark states of atomic ensembles, 2015.

[4] Y.I. Ozhigov, Space of dark states in Tavis-Cummings model, 2018.

[5] http://icl.cs.utk.edu/projectsfiles/magma/doxygen/