Содержание

1 Введение

Данная статья посвящена рассмотрению алгоритма моделирования системы идеальных твёрдых сфер вблизи идеальных стенок для компьютерных экспериментов по изучению процесса кристаллизации жидкости твёрдых сфер вблизи идеальной стенки. В этой статье описывается модель жёстких сфер, модель идеальной стенки и модель используемых периодических граничных условий.

Используемый нами ранее алгоритм [1] был существенно переработан, в частности, был применён ряд оптимизаций [2] [3] [4].

Описание, приведённое в данной статье, соответствует программе, написанной для моделирования исследуемой нами системы жёстких сфер, и может быть использовано для анализа алгоритма, реализованного в написанной программе.

Программа, реализующая описанный в этой статье алгоритм, написана на языке C++. Существует несколько версий программы, в которых реализация периодических граничных условий и некоторых функций может отличаться от описанного в этой статье алгоритма, в данной статье описывается только последний вариант реализации алгоритма, актуальный на данный момент, а так же приводится краткий обзор используемых ранее алгоритмов и периодических граничных условий с целью исторического обзора пути развития применяемых нами алгоритмов, а так же в целях объяснения причин, по которым мы выбрали именно данные граничные условия и определённую реализацию некоторых алгоритмов (выбор используемых алгоритмов, в основном, продиктован их оптимальностью - предпочтение отдавалось алгоритмам, требующим наименьшего времени для получения точного решения).

В разработке физической модели моделируемой системы и алгоритма, описанного в данной статье и реализованного в программе на языке C++, принимали активное участие:

Вешнев Владимир Петрович

Гераськин Алексей Сергеевич

Нурлыгаянова Марина Николаевна

Нурлыгаянов Тимур Артурович

Данная статья и любая информация, описанная в ней, а так же исходный код программы, реализующий данный алгоритм, может быть использован, изменён или опубликован только с согласия всех описанных лиц.

Во время работы над этой статьёй авторы использовали различную литературу, ссылки на которую будут даны в тексте статьи.

2 Общее описание моделируемой системы

2.1 Модель идеальной стенки

Модель идеальной жёсткой стенки используется для симуляции границы системы, через которую частицы не могут покинуть рассматриваемый объём. Модель идеальной жёсткой стенки широко применяется в различных компьютерных экспериментах и теоретических исследованиях.

Идеальная жёсткая стенка имеет следующие свойства:

- Имеет большую массу, стремящуюся к бесконечности:

$$M_{ideal\ wall} \to \infty, \tag{1}$$

- Является абсолютно гладкой для всех частиц в системе, т.е. при касании стенки любой частицей в системе возникающая между этой частицей и идеальной стенкой сила трения равна нулю,
- Является атермичной, т.е. не изменяет кинетическую энергию частиц в системе,
- Имеет импульс, равный нулю, т.е. не движется:

$$p_{ideal\ wall} = 0, \tag{2}$$

2.2 Модель идеальной жёсткой сферы

В данном эксперименте мы используем модель идеальной жёсткой сферы для моделирования частиц. Эта модель применяется во многих компьютерных экспериментах [5] [6] [7].

Идеальная жёсткая сфера обладает следующими свойствами:

- Имеет форму идеального равномерного шара, центр масс которого находится в его геометрическом центре [8],
- Не совершает вращательного движения,
- Является абсолютно гладкой для других идеальных жёстких сфер и идеальной стенки (т.е. при соприкосновении с другими жёсткими сферами или идеальной стенкой сила трения F, возникающая между двумя идеальными жёсткими сферами или идеальной жёсткой сферой и идеальной стенкой, равна нулю),
- Сталкивается с другими идеальными жёсткими сферами или идеальной стенкой мгновенно, т.е. промежуток времени Δt, на протяжении которого происходит соударение, стремится к нулю,
- В любой момент времени, если идеальная жёсткая сфера не сталкивается с другой идеальной жёсткой сферой или с идеальной стенкой, эта идеальная жёсткая сфера движется равномерно и прямолинейно,
- Имеет ненулевой размер и занимает объём системы, равный $\frac{4}{3}\pi R^3$, где R - радиус идеальной сферы.
- Сталкивается с другими идеальными жёсткими сферами и идеальной стенкой абсолютно упруго. Потенциал взаимодействия между двумя идеальными жёсткими сферами описывается следующей системой уравнений:

$$\begin{cases} p = 0, r > 2R, \\ p = \infty, r <= 2R, \end{cases} \tag{3}$$

где p - потенциал взаимодействия между двумя идеальными сферами, R - радиус идеальной жёсткой сферы, r - расстояние между центрами идеальных жёстких сфер.

Потенциал взаимодействия идеальной жёсткой сферы с идеальными стенками, которые ограничивают рассматриваемую нами систему, можно описать системой уравнений

$$\begin{cases} p = 0, r > R, \\ p = \infty, r <= R, \end{cases} \tag{4}$$

где p - потенциал взаимодействия между идеальной жёсткой сферой и идеальной твёрдой стенкой, R - радиус идеальной жёсткой сферы, r - расстояние между центром идеальной жёсткой сферы и идеальной твёрдой стенкой.

Потенциал взаимодействия между двумя жёсткими сферами или идеальной стенкой и жёсткой сферой, равный ∞, накладывает условие невозможности такого состояния системы, в котором расстояние между центрами двух идеальных жёстких сфер было бы меньше $2R$, а расстояние центра идеальной жёсткой сферы и идеальной стенкой было бы меньше R.

Идеальная жёсткая сфера соударяется с идеальной стенкой абсолютно упруго, при этом энергия частицы не изменяется, изменяется лишь проекция скорости частицы на ось OX:

$$v'_x = -v_x, \tag{5}$$

где v_x - проекция скорости частицы на ось OX до соударения с идеальной стенкой, v'_x - проекция скорости частицы на ось OX после соударения с идеальной стенкой.

Условие соударения идеальной жёсткой сферы с идеальными стенками в рассматриваемой системе можно записать так:

$$\begin{cases} x = R, v_x < 0.0 \\ or \\ x = L - R, v_x > 0.0, \end{cases} \tag{6}$$

где x - это значение координаты частицы вдоль оси OX, v_x - значение проекции скорости частицы на ось OX, R - радиус идеальной сферы, L - расстояние между двумя идеальными стенками в системе.

Частицы не могут находиться на расстоянии менее R от идеальной стенки и координаты всех частиц в системе в любой момент времени удовлетворяют следующим правилам:

$$\begin{cases} x >= R \\ x <= L - R. \end{cases} \tag{7}$$

2.3 Периодические граничные условия

Периодические граничные условия не дают частицам беспрепятственно покидать выделенный объём, позволяя тем самым моделировать бесконечную систему частиц, рассчитывая динамику системы, содержащую только небольшое количество частиц.

Условия, которые накладываются на частицы, находящиеся в моделируемой системе:

1. Центр каждой частицы всегда находится в системе или на её границе. Положения центров каждой частицы в системе в любой момент времени удовлетворяют следующей системе уравнений:

$$\begin{cases} x >= R, \\ x <= L - R, \\ 0 <= y <= A, \\ 0 <= z <= A, \end{cases} \tag{8}$$

где x, y и z - координаты частицы в трехмерной системе координат, R - радиус частицы, представляющей из себя идеальную жесткую сферу, L - расстояние между двумя идеальными стенками и A - это расстояние между двумя противоположными плоскостями, представляющими периодические границы.

2. Частицы могут пересекать границы системы, заданные уравнениями:

$$y = 0 \tag{9}$$

$$y = A \tag{10}$$

$$z = 0 \tag{11}$$

$$z = A \tag{12}$$

И не могут пересекать границы, заданные уравнением:

$$x = 0 \tag{13}$$

$$x = L \tag{14}$$

т.к. эти границы являются идеальными стенками.

При пересечении границы в момент пересечения одной из периодических границ системы центром частицы на одной из других периодических границ создаётся частица с энергией, равной энергии данной частицы и движущаяся согласно тому же уравнению движения, что и исходная частица. Исходная частица, пересекающая границу системы, при этом удаляется.

3. Если для координат центра частицы выполняется одно из следующих условий:

$$0 <= y <= 1 \tag{15}$$

$$A - 1 <= y <= A \tag{16}$$

$$0 <= z <= 1 \tag{17}$$

$$A - 1 <= z <= A \tag{18}$$

то существует один "образ"данной частицы, который движется согласно тому же уравнению движения, что и исходная частица. При пересечении центром частицы одной из границ системы "образ"становится обычной частицей, а сама частица уничтожается. При этом энергия частицы не изменяется, меняются лишь координаты центра частицы.

4. Центр "образа"всегда находится вне системы, что позволяет не учитывать "образ"как отдельную частицу. Любой частице в системе может соответствовать либо ни одного "образа либо только один "образ".

5. При столкновении частицы или ее "образа"с другой частицей, "образом"другой частицей или идеальной стенкой скорость частицы изменяется, как и её уравнение движения. При этом "образ соответствующий этой частице, уничтожается (если он существовал) и создаётся новый "образ"(если это необходимо - см. пункт №3), который будет двигаться согласно новому уравнению движения.

6. "Образ"может иметь скорость, отличную от скорости самой частицы, при этом он обязательно должен перемещаться согласно тому же уравнению движения, что и исходная частица (по той же линии скорости или по линии скорости, полученной параллельным переносом по OX, что будет рассмотрено отдельно). При столкновении "образа"с другой частицей или "образом"другой частицы учитывается скорость исходной частицы, а не её "образа"(при расчёте новых скоростей), что позволяет сохранять общую энергию системы постоянной.

7. При создании "образа"некоторой частицы позиция его центра выбирается так, чтобы он находился на линии скорости исходной частицы, в точке, из которой он достигнет одной из периодических границ системы в то же самое время, когда центр исходной частицы будет находиться в плоскости пересекаемой ею границы.

Рассмотрим подробнее вычисление координат центра создаваемого "образа". Уравнение движения центра сферы по прямолинейной траектории с постоянной скоростью в векторном виде можно записать так:

$$\vec{r} = \vec{r_0} + \vec{v} * t \tag{19}$$

где $\vec{r}$ - радиус вектор нового положения центра частицы, $\vec{r_0}$ - радиус вектор начального положения центра частицы, $\vec{v}$ - скорость частицы и t - время движения.

Распишем это выражение в проекциях на оси координат:

$$\begin{cases} r_x = r_{0x} + v_x * t, \\ r_y = r_{0y} + v_y * t, \\ r_z = r_{0z} + v_z * t. \end{cases} \tag{20}$$

из этой системы уравнений получаем:

$$\begin{cases} t = \dfrac{r_x - r_{0x}}{v_x}, \\ t = \dfrac{r_y - r_{0y}}{v_y}, \\ t = \dfrac{r_z - r_{0z}}{v_z}. \end{cases} \tag{21}$$

Полученная система уравнений (21) позволяет рассчитывать время, требующееся для перемещения центра частицы из одной позиции в другую.

Когда центр частицы находится на границе системы, прямая, описывающая уравнение её движения, пересекает плоскость, которой описана эта граница. Рассчитав времена, которые потребуются частице для того, чтобы её центр оказался в плоскости каждой границы, мы можем вычислить границу, на которой центр выбранной частицы окажется быстрее всего. Для этого необходимо решить четыре системы уравнений (одна система уравнений для каждой периодической границы):

$$\begin{cases} y = 0, \\ t = \dfrac{r_y - r_{0y}}{v_y}. \end{cases} \tag{22}$$

$$\begin{cases} y = A, \\ t = \dfrac{r_y - r_{0y}}{v_y}. \end{cases} \tag{23}$$

$$\begin{cases} z = 0, \\ t = \dfrac{r_z - r_{0z}}{v_z}. \end{cases} \tag{24}$$

$$\begin{cases} z = A, \\ t = \dfrac{r_z - r_{0z}}{v_z}. \end{cases} \tag{25}$$

полагая при решении этих систем уравнений $r_y = y$ и $r_z = z$, мы получим 4 решения для t:

$$\begin{cases} t_1 = \dfrac{0 - r_{0y}}{v_y}, \\ t_2 = \dfrac{A - r_{0y}}{v_y}, \\ t_3 = \dfrac{0 - r_{0z}}{v_z}, \\ t_4 = \dfrac{A - r_{0z}}{v_z}. \end{cases} \tag{26}$$

где r_{0y}, r_{0z} - проекции радиус вектора текущего положения частицы на оси OY и OZ.

Необходимо найти t_{min}, удовлетворяющее условиям:

$$\begin{cases} t_{min} > 0, \\ t_{min} <= t_1, \\ t_{min} <= t_2, \\ t_{min} <= t_3, \\ t_{min} <= t_4, \end{cases} \tag{27}$$

где t_{min} есть время, за которое частица достигнет ближайшей границы системы, если будет двигаться с текущей скоростью. В соответствии с условием №7 'образ', соответствующий данной частице, должен за это же время достигнуть другой периодической границы системы.

Для того, чтобы вычислить координаты центра создаваемого "образа возьмём новые значения v_x, v_y и v_z, равные:

$$\begin{cases} v'_x = -v_x, \\ v'_y = -v_y, \\ v'_z = -v_z. \end{cases} \tag{28}$$

и рассчитаем для них уравнения (26) и (27), подставив полученное в результате решения этих уравнений значение t'_{min} значения v'_x, v'_y и v'_z в систему уравнений (49), получим:

$$\begin{cases} r'_x = r_{0x} + v'_x * t'_{min}, \\ r'_y = r_{0y} + v'_y * t'_{min}, \\ r'_z = r_{0z} + v'_z * t'_{min}. \end{cases} \tag{29}$$

где r'_x, r'_y и r'_z есть значения новых координат центра частицы, которые она будет иметь после прохождения через периодическую границу.

Сложив t_{min} и t'_{min} и подставив получившееся значение t в систему уравнений (49) мы получим значения координат создаваемого образа, соответствующего данной частице:

$$\begin{cases} r''_x = r_{0x} + v'_x * (t_{min} + t'_{min}), \\ r''_y = r_{0y} + v'_y * (t_{min} + t'_{min}), \\ r''_z = r_{0z} + v'_z * (t_{min} + t'_{min}). \end{cases} \tag{30}$$

здесь r''_x, r''_y и r''_z - координаты создаваемого образа, r_{0x}, r_{0y} и r_{0z} - текущие координаты исходной частицы, для которой необходимо создать виртуальную частицу.

3 Расчёт столкновения идеальных жёстких сфер

Необходимо описать подробно.

3.1 Расчёт времени до соударения двух идеальных жёстких сфер

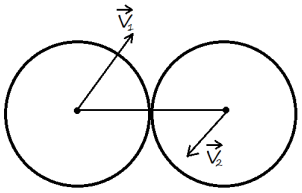
Необходимо описать подробно.

3.2 Расчёт новых скоростей идеальных жёстких сфер после соударения

Рассмотрим общий случай соударения двух частиц, представляющих из себя идеальные жёсткие сферы.

При любом столкновении двух идеальных жёстких сфер центры масс этих сфер, а так же точка соприкосновения поверхностей этих двух сфер лежат в одной плоскости.

Рассмотрим проекцию двух сфер и их скоростей на эту плоскость в момент соударения.



Здесь $\vec{v}_1$ - скорость первой частицы, $\vec{v}_2$ - скорость второй частицы.

Так как рассматриваемые сферы являются равномерными идеальными сферами и точка касания двух сфер находится на одной линии с центрами масс обеих сфер, то и силы, действующие со стороны первой частицы на вторую и со стороны второй частицы на первую будут направлены по линии, соединяющей центры масс сфер.

Идеальные сферы являются абсолютно гладкими и не могут совершать вращательное движение, поэтому силы, действующие на первую и вторую частицы, действуют только вдоль линии, соединяющей центры масс данных частиц.

Изобразим действующие на обе идеальные сферы силы:

Согласно третьему закону Ньютона должно выполняться равенство:

$$\vec{F}_1 = -\vec{F}_2 \tag{31}$$

Распишем скорость $\vec{v}_1$ как сумму $\vec{v}_{1\perp}$ и $\vec{v}_{1\parallel}$ и скорость $\vec{v}_2$ как сумму $\vec{v}_{2\perp}$ и $\vec{v}_{2\parallel}$, где $\vec{v}_{1\perp}$ и $\vec{v}_{2\perp}$ составляющие скоростей $\vec{v}_1$ и $\vec{v}_2$, перпендикулярные линии, соединяющей центры сталкивающихся частиц, $\vec{v}_{1\parallel}$ и $\vec{v}_{2\parallel}$ - параллельные составляющие скоростей $\vec{v}_1$ и $\vec{v}_2$ относительно линии, соединяющей центры частиц.

Учитывая, что силы, действующие на первую и вторую частицы в момент их соударения, действуют только вдоль прямой, соединяющей центры масс данных частиц, можно утверждать, что составляющие скоростей движения частиц, перпендикулярные линии, соединяющей центры этих частиц, не влияют на величину и направление сил $\vec{F}_1$ и $\vec{F}_2$.

Поэтому учитывая, что масса идеальных жёстких сфер постоянна, мы можем записать:

$$\begin{cases} \vec{F}_1 = \dfrac{m_1 * d\vec{v}_{1\parallel}}{dt}, \\ \dfrac{m_1 * d\vec{v}_{1\perp}}{dt} = 0, \\ \vec{F}_2 = \dfrac{m_2 * d\vec{v}_{2\parallel}}{dt}, \\ \dfrac{m_2 * d\vec{v}_{2\perp}}{dt} = 0, \end{cases} \tag{32}$$

Учитывая второй закон Ньютона для данной системы двух идеальных твёрдых сфер (31), а так же то, что массы всех идеальных твёрдых сфер равны, мы можем записать (32) в виде:

$$\begin{cases} \dfrac{d\vec{v}_{2\parallel}}{dt} = -\dfrac{d\vec{v}_{1\parallel}}{dt}, \\ \vec{v}_{1\perp} = const, \\ \vec{v}_{2\perp} = const, \end{cases} \tag{33}$$

Идеальные жёсткие сферы соударяются абсолютно упруго, а значит при соударении двух идеальных жёстких сфер должны выполняться закон сохранения импульса и закон сохранения энергии.

Закон сохранения импульса для рассматриваемой системы двух частиц может быть записан как

$$\begin{cases} \vec{v}_{1\perp} + \vec{v}_{2\perp} = \vec{v}\,'_{1\perp} + \vec{v}\,'_{2\perp}, \\ \vec{v}_{1\parallel} + \vec{v}_{2\parallel} = \vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel}, \end{cases} \tag{34}$$

где $\vec{v}\,'_{1\perp}$, $\vec{v}\,'_{1\parallel}$, $\vec{v}\,'_{2\perp}$, $\vec{v}\,'_{2\parallel}$ - скорости частиц после соударения.

Запишем закон сохранения энергии для данной системы из двух сталкивающихся идеальных сфер:

$$\begin{cases} E_1 = \dfrac{mv_{1\perp}^2}{2} + \dfrac{mv_{1\parallel}^2}{2} + \dfrac{mv_{2\perp}^2}{2} + \dfrac{mv_{2\parallel}^2}{2}, \\ E_2 = \dfrac{mv\,'^2_{1\perp}}{2} + \dfrac{mv\,'^2_{1\parallel}}{2} + \dfrac{mv\,'^2_{2\perp}}{2} + \dfrac{mv\,'^2_{2\parallel}}{2}, \\ E_1 = E_2, \end{cases} \tag{35}$$

где E_1 - полная кинетическая энергия системы из двух частиц перед соударением, E_2 - полная кинетическая энергия системы из двух частиц после соударения. Потенциальная энергия частиц при соударении не изменяется, т.к. происходит абсолютно упругий удар.

Учитывая (33) мы можем записать, что $\vec{v}_{1\perp} = \vec{v}\,'_{1\perp}$ и $\vec{v}_{2\perp} = \vec{v}\,'_{2\perp}$, тогда учитывая закон сохранения импульса (34) и закон сохранения энергии (35) мы можем записать следующую систему уравнений:

$$\begin{cases} \vec{v}_{1\parallel} + \vec{v}_{2\parallel} = \vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel}, \\ \dfrac{mv_{1\parallel}^2}{2} + \dfrac{mv_{2\parallel}^2}{2} = \dfrac{mv\,'^2_{1\parallel}}{2} + \dfrac{mv\,'^2_{2\parallel}}{2} \end{cases} \tag{36}$$

Сразу упростим второе уравнение, учтя равенство масс двух идеальных жёстких сфер:

$$\begin{cases} \vec{v}_{1\parallel} + \vec{v}_{2\parallel} = \vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel}, \\ v_{1\parallel}^2 + v_{2\parallel}^2 = v\,'^2_{1\parallel} + v\,'^2_{2\parallel} \end{cases} \tag{37}$$

Преобразуем первое уравнение в (37), домножив обе части уравнения на $\vec{v}_{1\parallel}$:

$$\vec{v}_{1\parallel}^2 + \vec{v}_{2\parallel}\vec{v}_{1\parallel} = \vec{v}\,'_{1\parallel}\vec{v}_{1\parallel} + \vec{v}\,'_{2\parallel}\vec{v}_{1\parallel} \tag{38}$$

Теперь выразим из этого уравнения $\vec{v}_{1\parallel}^2$ и подставим получившееся выражение во второе уравнение системы уравнений (37), получим:

$$\vec{v}_{1\parallel}(\vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}) + \vec{v}_{2\parallel}^2 = \vec{v}\,'^2_{1\parallel} + \vec{v}\,'^2_{2\parallel} \tag{39}$$

и выразим из полученного уравнения $\vec{v}_{1\parallel}$:

$$\vec{v}_{1\parallel} = \frac{\vec{v}\,'^2_{1\parallel} + \vec{v}\,'^2_{2\parallel} - \vec{v}^2_{2\parallel}}{\vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}} \tag{40}$$

подставим результат в первое уравнение из (37), получим:

$$\frac{\vec{v}\,'^2_{1\parallel} + \vec{v}\,'^2_{2\parallel} - \vec{v}^2_{2\parallel}}{\vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}} + \vec{v}_{2\parallel} = \vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel} \tag{41}$$

что можно записать как:

$$\vec{v}\,'^2_{1\parallel} + \vec{v}\,'^2_{2\parallel} - \vec{v}^2_{2\parallel} = (\vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}) * (\vec{v}\,'_{1\parallel} + \vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}) \tag{42}$$

после раскрытия скобок во второй части равенства и приведения подобных слагаемых мы получаем:

$$-2\vec{v}^2_{2\parallel} = 2\vec{v}\,'_{1\parallel}\vec{v}\,'_{2\parallel} - 2\vec{v}\,'_{1\parallel}\vec{v}_{2\parallel} - 2\vec{v}_{2\parallel}\vec{v}\,'_{2\parallel} \tag{43}$$

сократив обе части равенства на 2 и вынеся за скобку $\vec{v}\,'_{1\parallel}$ мы получаем:

$$\vec{v}\,'_{1\parallel}(\vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}) - \vec{v}_{2\parallel}\vec{v}\,'_{2\parallel} = -\vec{v}^2_{2\parallel} \tag{44}$$

Выразим $\vec{v}\,'_{1\parallel}$ из (44) мы можем записать:

$$\vec{v}\,'_{1\parallel} = \frac{\vec{v}_{2\parallel}\vec{v}\,'_{2\parallel} - \vec{v}^2_{2\parallel}}{\vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}} \tag{45}$$

и вынеся за скобку $\vec{v}_{2\parallel}$, запишем это выражение как

$$\vec{v}\,'_{1\parallel} = \frac{\vec{v}_{2\parallel}(\vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel})}{\vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}} \tag{46}$$

Если мы сократим в (46) числитель и знаменатель дроби на $\vec{v}\,'_{2\parallel} - \vec{v}_{2\parallel}$ то получим конечное выражение для $\vec{v}\,'_{1\parallel}$:

$$\vec{v}\,'_{1\parallel} = \vec{v}_{2\parallel} \tag{47}$$

что с учётом (33) даёт нам право записать конечную систему уравнений, определяющую выражения для новых скоростей идеальных сфер после соударения:

$$\begin{cases} \vec{v}\,'_{1\parallel} = \vec{v}_{2\parallel}, \\ \vec{v}\,'_{2\parallel} = \vec{v}_{1\parallel}, \\ \vec{v}\,'_{1\perp} = \vec{v}_{1\perp}, \\ \vec{v}\,'_{2\perp} = \vec{v}_{2\perp}. \end{cases} \tag{48}$$

4 Начальное состояние системы. Посев

4.1 Посев

Для начала моделирования необходимо создать N жёстких сфер в моделируемой системе таким образом, чтобы центры всех частиц находились в исследуемом объёме, частицы не проникали друг в друга, а так же чтобы начальный импульс системы и момент импульса системы по осям OX, OY и OZ были равны нулю.

Процедура инициализации начальных параметров объёма, положений частиц и их скоростей называется "посев". В дальнейшем мы будем использовать этот термин для обозначения функции программы, которая позволяет произвести начальную инициализацию моделируемой системы.

Начальная плотность данной системы η_0 будет определяться задаваемым параметрами объёма исследуемой системы и количеством частиц в данном объёме.

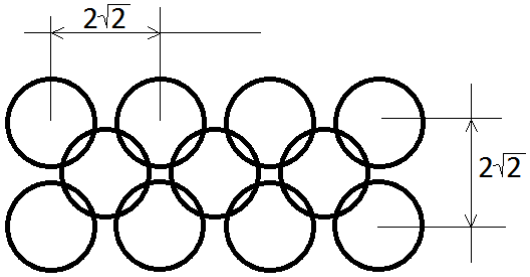
Изначально центры частиц размещаются в системе в "узлах"объёмноцентрированной кристаллической решётки, расстояние между частицами регулируется начальной плотностью системы, но при этом расстояние между двумя частицами не может быть меньше, чем два радиуса идеальных сфер (чтобы частицы не проникали друг в друга).

Расчёт начальных координат и скоростей идеальных сфер производится в несколько этапов, рассмотрим их более подробно:

1. Создаётся "кристалл"жёстких сфер с объёмноцентрированной кристаллической решёткой, содержащий M частиц. Частицы в данном кристалле касаются друг друга и образуют прямоугольный параллелепипед, рёбра которого по Y и Z равны между собой, при этом рёбра по X примерно в два раза больше рёбер по Y и Z. Это продиктовано особенностями моделируемой системы, т.к. нам необходимо задавать длину системы L в четыре раза больше, чем размер системы по Y и Z, чтобы иметь возможность исследовать свойства кристалла вблизи идеальной стенки, исключая при этом влияние второй, противоположной идеальной стенки, которая будет достаточно удалена от исследуемой части системы.

Один из углов данного параллелепипеда будет совпадать с началом координат.

Расстояние между двумя одинаковыми слоями объёмноцентрированного кристалла в случае плотного расположения слоёв равно $2R * \sqrt{2}$:



Координаты центра данного прямоугольного параллелепипеда можно рассчитать по формулам:

$$\begin{cases} y = \dfrac{r_b * (K-1) + 2R}{2}, \\ z = \dfrac{r_b * (K-1) + 2R}{2}, \\ x = \dfrac{r_b * (2*K-1) + 2R}{2}. \end{cases} \tag{49}$$

где r_b - это расстояние между одинаковыми слоями объёмноцентрированного кристалла, равное $2R*\sqrt{2}$, K - количество частиц в ребре куба по Y и Z, это число задаётся при новом "посеве"частиц (см. главу "Описание управляющих команд программы").

2. Созданный кристалл копируется несколько раз так, чтобы увеличить кристалл вдвое по трём измерениям (x, y, z), таким образом, делается 7 копий созданного на шаге №1 кристалла, кото-

рые располагаются вплотную к первоначальному кристаллу так, чтобы частицы, находящиеся на границах параллелепипедов, касались друг друга.

В результате получается кристалл жёстких сфер в форме прямоугольно параллелепипеда, один угол которого совпадает с началом координат и одна сторона которого в два раза больше двух других сторон.

3. Для частиц, находящихся в кристалле, созданном на шаге №1, случайным образом задаются скорости v_x, v_y и v_z, при этом скорости остальных частиц в системе задаются так, чтобы суммарный импульс и момент импульса системы были равны нулю.

4. Основываясь на задаваемой плотности системы рассчитываются параметры моделируемого объёма A и L, а так же коэффициент расширения β (мы рассмотрим подробнее вывод коэффициента расширения системы чуть ниже), на который умножаются все координаты частиц, благодаря чему объёмноцентрированный кристалл жёстких сфер расширяется до размеров моделируемой системы и частицы перестают касаться друг друга. При этом суммарный импульс и момент импульса системы остаются равными нулю, т.к. импульсы одних частиц уравновешены импульсами других частиц, расположенных симметрично относительно центра моделируемой системы.

5. Все частицы копируются ещё раз и их координаты изменяются параллельным переносом вдоль оси OX так, чтобы в результате частицы заполнили весь объём системы, заданный из расчёта $L = 4 * A$. Скорости частиц не изменяются, поэтому общий импульс и момент импульса системы при этом так же не изменяются. Таким образом размер системы вдоль оси OX получается в четыре раза больше размера системы вдоль осей OY и OZ, центры идеальных сфер распределены по всему объёму моделируемой системы так, чтобы частицы не проникали друг в друга и суммарный импульс системы и момент импульса системы были равны нулю.

После проведения "посева"состояние системы сохраняется во временный файл и загружается из него. Во время загрузки сохранённого состояния системы мы рассчитываем ближайшие события для каждой частицы и начинаем расчёт динамики системы. Более подробно это описано в главе "Сохранение и загрузка состояния системы".

4.2 Расчёт коэффициента расширения системы β

Рассмотрим подробнее вывод коэффициента расширения системы β.

Необходимо описать подробно.

5 Динамика моделируемой системы. События

Существует два метода расчёта динамики системы многих тел:

1. Итеративный. При этом подходе мы рассчитываем каждое следующее состояние системы, перемещая её во времени на некоторое δt. После каждого смещения по времени необходимо рассчитать новые положения и скорости каждого тела в системе.

2. Событийный. При таком подходе мы рассматриваем динамику системы многих тел как последовательность некоторых событий. Событиями в данном контексте называются моменты времени, в которые происходит изменение уравнений движения частиц в системе. На каждом шаге мы рассчитываем положение всех частиц в момент времени, когда произойдет следующее событие, изменяющее уравнения движения частиц и сразу же переводим всю систему в этот момент времени. После этого необходимо рассчитать изменение уравнений движения частиц и перейти на следующий шаг, рассчитывая новые положения частиц в момент времени, когда произойдёт следующее событие, меняющее уравнение движения частиц.

Идеальные сферы в моделируемой системе движутся равномерно и прямолинейно в промежутках времени между столкновениями.

Это означает, что во время перемещения каждой частицы в пространстве значение и направление её скорости не изменяется, и это справедливо для всех частиц.

Благодаря тому, что уравнения движения идеальных сфер изменяются только при наступлении определённых 'событий', мы применяем событийный подход при моделировании динамики моделируемой системы.

5.1 Типы событий

В рассматриваемой нами системе могут происходить следующие события:

1. Соударение частицы или её "образа"с идеальной стенкой.

2. Соударение частицы или её "образа"с другой частицей или "образом"другой частицы.

3. Создание "образа"для частицы, находящейся в области, близкой к периодическим граничным условиям.

4. Переход частицы из одной "ячейки"в другую.

5. Замена "образа"частицей - "перерождение"частицы при прохождении через периодические граничные условия.

6. Удаление "образа"частицы.

Необходимо описать подробно.

5.2 Очередь событий

Представим, что у нас есть несколько частиц в системе и мы рассчитали времена ближайших событий для каждой частицы. Для начала запишем их в один линейный список:

<Вставить картинку случайно расположенных элементов>

Здесь каждое событие представлено в виде прямоугольника, содержащего информацию о номерах частиц, которые будут участвовать в событии, типе событий и времени dt, через которое эти события произойдут в системе.

Пока состояние системы не изменилось не меняется и список ближайших событий для всех частиц в системе. Как только произойдёт первое из этих событий, состояние системы изменится и мы должны будем обнаружить следующее ближайшее событие для той частицы, чьё событие только что наступило (для удобства будем считать что это частица с индексом i).

При расчёте следующего ближайшего события для частицы i может оказаться, что одно из уже записанных в данный список событий для другой частицы j никогда не произойдёт, например, потому что столкновение двух частиц i и j произойдёт раньше, чем ранее рассчитанное событие для частицы j. Тогда нам незачем хранить в памяти событие, которое никогда не произойдёт, и мы можем удалить из этого списка старое событие с участием частицы j и добавить к этому списку новое событие соударение частиц i и j.

События других частиц, чьи уравнения движения и времена ближайших событий не изменились после наступления события с частицей i, можно не удалять из списка ближайших событий и не рассчитывать их заново, т.к. они не изменятся.

Так как мы рассчитываем динамику системы на основе событийного подхода, на каждом шаге нам необходимо выбирать из списка всех событий событие с наименьшим временем (чтобы выбрать событие, которое произойдёт в системе раньше других). Поиск такого события в списке событий требует последовательного обращения ко всем элементам этого списка и сравнения времени каждого записанного события со временем других событий в этом же списке.

Чтобы упростить задачу определения следующего события в системе упорядочим список событий по времени наступления событий, в порядке возрастания:

<Вставить картинку упорядоченных линейных элементов>

Тогда ближайшее событие системы всегда будет храниться в первой ячейке нашего списка, и нам не нужно будет проверять все события в этом списке на каждом шаге расчёта динамики системы.

При расчёте следующего ближайшего события для частицы i необходимо будет вставить новое событие в список событий так, чтобы упорядоченность списка не была нарушена, т.е. необходимо проверить события, уже находящиеся в очереди событий и вставить новое событие в некоторую позицию в данном списке так, чтобы последовательность событий не нарушилась.

Сформулируем правила для 'очереди событий':

1. В очереди событий в каждый момент времени хранится информация о ближайших событиях для всех частиц и 'образов' частиц, если эти 'образы' существуют;

2. В очереди событий хранится информация только о ближайшем событии для каждой частицы и 'образа'. Информация о неактуальных событиях удаляется из очереди событий;

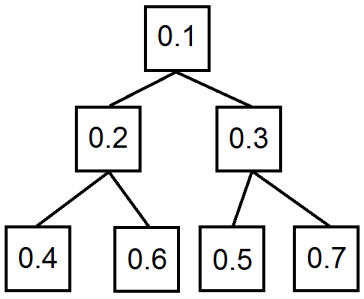
3. Элементы в очереди событий расположены упорядочено, в порядке возрастания времени до наступления события;

4. Для каждой частицы и для каждого существующего в системе образа в очереди событий хранится только одно событие. Это значит, что число событий в очереди событий не может быть больше, чем $2N$, где N - число частиц в системе(здесь мы учитываем, что у каждой частицы может быть только один образ).

5.3 Бинарное дерево как структура данных

Бинарное (двоичное) дерево - это структура данных, представляющая собой совокупность элементов и отношений, образующих иерархическую структуру этих элементов.

Если представить бинарное дерево в виде диаграммы, то оно напоминает настоящее перевёрнутое дерево с ветвями:



Элементы дерева называются вершинами или узлами дерева. Вершины дерева соединены направленными дугами, которые называются ветвями дерева. Начальная вершина дерева называется корнем дерева, ей соответствует нулевой уровень. Листьями дерева называют вершины, в которые входит одна ветвь и из которых не выходит ни одной ветви.

Вершины, в которые входят ветви, исходящие из одной общей вершины, называются потомками или наследниками этой вершины, а вершина, из которой выходят эти узлы, называется родительским узлом. Уровень потомка на единицу превосходит уровень его предка. Корень дерева не имеет предка, а листья дерева не имеют потомков.

5.4 Бинарное дерево для сохранения информации о всех ближайших событиях в системе

Для сохранения информации о событиях в системе мы будем использовать упорядоченное нестрогое двоичное дерево, обладающее следующими свойствами:

1. Каждый элемент имеет не более двух дочерних элементов;
2. Значение dt в любой вершине не меньше, чем значения dt её потомков;
3. Глубина листьев (расстояние до корня) отличается не более чем на 1 слой;
4. Последний слой заполняется слева направо;
5. Если некоторый родительский элемент имеет два дочерних элемента, то левый дочерний элемент имеет значение dt_1 меньшее, чем значение правого дочернего элемента dt_2;

5.5 Добавление элемента в бинарное дерево

Необходимо описать подробно.

5.6 Удаление элемента из бинарного дерева

Необходимо описать подробно.

5.7 Сравнение алгоритмов на основе бинарного дерева и линейного массива

Необходимо описать подробно.

6 Собственные времёна частиц и глобальное время системы

Необходимо описать подробно.

6.1 Изменение положения частиц в системе

Необходимо описать подробно.

6.2 Синхронизация частиц

Необходимо описать подробно.

6.3 Поиск ближайшего события для частицы

Необходимо описать подробно.

7 Сохранение и загрузка состояния системы

Необходимо описать подробно.

8 Измерение относительной плотности

Необходимо описать подробно.

9 Получение данных о структуре системы

Необходимо описать подробно.

9.1 Профиль плотности

Необходимо описать подробно.

9.2 Исследование структуры выделенного слоя частиц. Разрезы

Необходимо описать подробно.

10 Изменение объёма системы

Необходимо описать подробно.

11 Измерение давления

Данная часть алгоритмов и программы ещё не реализовано, необходимо добавить описание того, как мы будем производить измерения данных и описать реализацию алгоритма в программе.

12 Измерение химического потенциала

Данная часть алгоритмов и программы ещё не реализовано, необходимо добавить описание того, как мы будем производить измерения данных и описать реализацию алгоритма в программе.

13 Описание управляющих команд программы

Для управления ходом компьютерного эксперимента необходимо использовать управляющие команды, которые должны быть последовательно описаны в текстовом файле *program.txt* в директории с запускаемой программой. Программа последовательно считывает управляющие команды и файла *program.txt* и так же последовательно выполняет их, выводя информацию о текущей задаче на экран.

Все команды имеют строго определённый синтаксис, опечатки и неправильная последовательность аргументов, пустые строки и некорректные символы в файле описания хода эксперимента могут привести к аварийному завершению программы, сохранению данных в неправильные файлы и даже перезапись уже существующих экспериментальных данных (если в программе эксперимента будет указано, что нам необходимо сохранить некоторые данные в уже существующий файл). Необходимо аккуратно заполнять файл описания эксперимента *program.txt* и проверять его содержимое до запуска эксперимента.

Все функции, которые вызываются для выполнения команд, детально описаны отдельно в различных главах данной статьи.

Рассмотрим подробнее каждую управляющую команду, параметры для различных команд и примеры использования.

13.1 new

new - команда, позволяющая запустить процедуру инициализации начального состояния системы (см. главу Посев).

Аргументы:

- Количество частиц в ребре куба объёмноцентрированного кристалла, который будет использоваться во время посева. От числа частиц в ребре этого куба зависит конечное число частиц в моделируемой системе. Например, если число частиц в ребре равно 2, то всего частиц в системе будет 64, если число частиц ребре равно 8, то частиц в системе будет 6976.

- Относительная плотность частиц в системе, которую необходимо задать при инициализации начального состояния. Данная плотность не может быть больше 0.68 (максимальная относительная плотность кристалла с объёмноцентрированной кристаллической решёткой).

Пример:

new 8 0.4

Данная команда создаст новую систему из 6976 частиц, начальная относительная плотность частиц в системе η будет равна 0.4. После инициализации системы так же будут рассчитаны времена ближайших событий для каждой частицы.

После выполнения этой команды мы можем использовать другие управляющие команды, такие, как *step*, *compress*, $profile$ и т.д.

13.2 save

Для сохранения состояния системы используется команда *save*. При сохранении состояния системы в указанный нами текстовый файл будут сохранены число частиц в системе, число "образов"частиц, существующих на данный момент в системе, параметры объёма A и L моделируемой системы а так же координаты и скорости каждой частицы и всех существующих на данный момент времени в системе "образов"частиц.

В дальнейшем сохранённое состояние системы можно будет загрузить с помощью команды *load*.

Аргументы:

- Имя текстового файла, в который необходимо сохранить данные о состоянии системы. Если этот файл не существует, он будет создан в директории с программой или по указанному пути, если указать имя файла с полным путём к нему. Если такой файл уже существует, то существующий файл будет перезаписан новыми данными.

Пример:

save $save\_file\_0.40000\_10000.txt$

Данная команда создаст файл $save\_file\_0.40000\_10000.txt$ в папке с программой и запишет в этот файл информацию о текущем состоянии системы.

Рекомендуем в имени файла, содержащего информацию о системе, добавлять информацию о текущей плотности системы а так же о количестве соударений, прошедших в систем с момента последнего изменения плотности системы. Это поможет в дальнейшем быстро найти необходимое состояние в системе и вспомнить как именно данное состояние было получено.

13.3 load

Для загрузки некоторого ранее сохранённого состояния моделируемой системы из текстового файла используется команда *load*. Данная команда обнуляет текущее состояние системы, загружает данные о состоянии системы из текстового файла и запускает перерасчёт времён ближайших событий для каждой частицы и каждого "образа"в системе.

Для корректной работы данной команды необходимо существование файла с информацией о некотором состоянии моделируемой системы, синтаксис такого файла строго определён и не должен подвергаться ручному изменению, такие файлы создаются автоматически при сохранении некоторого состояния системы с помощью команды *save*.

Аргументы:

- Имя текстового файла, из которого необходимо загрузить информацию о некотором, сохранённом ранее, состоянии системы. Данный файл должен существовать и быть доступным для чтения запускаемой программой, он должен иметь строго определённую структуру и синтаксис. В случае, если указанный файл не существует, программа будет аварийно завершена.

Пример:

load $save\_file\_0.40000\_10000.txt$

Данная команда загрузит состояние системы из файла $save\_file\_0.40000\_10000.txt$ и произведёт расчёт времен ближайших событий для всех частиц и "образов"в системе.

После выполнения этой команды мы можем использовать другие управляющие команды, такие, как *step*, *compress*, $profile$ и т.д.

13.4 step

Управляющая команда *step* позволяет запустить моделирование динамики системы частиц.

Аргументы:

- Количество соударений в системе на одну частицу, в течении которых необходимо рассчитать динамику моделируемой системы. Этот параметр должен быть натуральным числом.

Пример:

step 1000

Данная команда просчитает динамику системы начиная от последнего состояния системы и до тех пор, пока в системе не произойдёт $500 * N$ соударений между частицами, где N - количество частиц в системе. Так как в каждом соударении частиц участвует одновременно две частицы, можно считать что при этом в среднем в системе произошло 1000 соударений для каждой частицы, хотя на самом деле для каждой отдельной частицы число соударений будет разным.

В логику расчёта динамики системы частиц входит множество различных операций, все они подробно описаны в данной статье, а в этой главе мы ограничимся описанием управляющей команды *step*, которая позволяет управлять продолжительностью моделирования динамики этой системы.

Примечание:

Мы можем использовать команду *step* несколько раз подряд, при этом состояние системы каждый раз не будет сбрасываться и возвращаться в некоторое исходное состояние, команда *step* будет использовать текущее состояние системы как исходное. Например, если мы напишем такой алгоритм:

step 1000

step 10000

то программа рассчитает сначала динамику системы в течении тысячи соударений на каждую частицу в системе, а затем рассчитает динамику системы в течении ещё десяти тысяч соударений, начиная с того состояния системы, на котором мы остановились после расчёта тысячи соударений в системе.

13.5 compress

Управляющая команда *compress* позволяет изменять параметры объёма моделируемой системы, за счёт чего изменяется относительная плотность частиц в системе.

Изменение параметров объёма системы происходит вследствие изменения L (длины системы по оси OX), изменяя длину моделируемой системы мы симулируем движение идеальных стенок, которые ограничивают моделируемую систему в плоскостях $X = 0$ и $X = L$.

Существует три типа изменения плотности системы: $compress\_left\_wall$, $compress\_right\_wall$, $compress\_both\_walls$, которые симулируют, соответственно, движение только левой стенки, движение только правой стенки и движение одновременно двух стенок.

Все три типа процедур изменения параметров системы действуют по примерно одному и тому же алгоритму, который коротко можно описать так:

1. Найти ближайшую к идеальной стенке частицу и определить расстояние Δr от поверхности сферы, ограничивающей частицу, до идеальной стенки.

2. Сдвинуть стенку на расстояние, меньшее Δr и меньшее некоторой заранее определённой ΔL. Важно отметить, что при смещении одной или нескольких идеальных стенок в системе перерасчёт позиций "образов"частиц произведён не будет, т.к. мы можем перемещать образы вдоль оси OX не изменяя при этом момент импульса L_x моделируемой системы.

3. Рассчитать время ближайших событий для всех частиц и образов.

5. Рассчитать динамику в системе на протяжении некоторого, заранее заданного, числа соударений на каждую частицу.

6. Если относительная плотность частиц в системе не соответствует указанной пользователем в параметрах плотности, то вернуться на шаг №1.

Рассмотрим каждый тип изменения параметров системы в отдельности.

13.5.1 *compress_left_wall*

Управляющая команда $compress\_left\_wall$ позволяет изменять параметры моделируемой системы, симулируя движение "левой"идеальной стенки (имеется ввиду идеальная стенка $X = 0$, данная стенка условно называется левой, т.к. на профиле плотности, где представлены данные о значениях относительной плотности в системе, данная граница системы расположена слева, в начале координат).

При этом сама "левая"идеальная стенка не движется, программа производит смещение всех частиц и образов на определённое ΔL в сторону этой стенки вдоль оси OX, после чего уменьшает параметр системы L на ту же величину ΔL, в результате чего идеальная стенка, ограничивающая моделируемую нами систему в плоскости $x = 0$, становится ближе ко всем частицам в системе на расстояние ΔL (или дальше, это зависит о того, хотим ли мы увеличить относительную плотность частиц в системе или хотим уменьшить её). Расстояние между частицами и второй идеальной стенкой, ограничивающей систему в плоскости $x = L$ не изменяется.

Таким образом мы моделируем изменение координаты "левой"идеальной стенки, приближая её ко всем частицам в системе или отдаляя эту границу от них.

Величина ΔL, на которую мы изменяем расстояние между частицами и стенкой не может быть больше расстояния между идеальной стенкой и самой ближней к ней частице, а так же ограничивается значением максимально допустимого смещения стенки, задаваемого в качестве аргумента для данной команды.

Аргументы:

- Относительная плотность системы, которую необходимо установить в системе с помощью движения идеальной стенки. Указываемая относительная плотность частиц в системе может быть как больше, так и меньше текущей плотности в системе, в зависимости от чего с помощью данной команды можно как сжимать моделируемую систему, уменьшая объём, в котором находятся частицы, так и наоборот, расширять данную системы, отодвигая идеальную стенку от частиц и уменьшая относительную плотность системы.

- Значение максимально допустимого смещения стенки ΔL_{max}. Если ближайшая к идеальной стенке частица находится на расстоянии, большем ΔL_{max}, то идеальная стенка может быть смещена только на расстояние ΔL_{max}.

- Число K1 соударений на частицу, которое необходимо рассчитать после каждого маленького изменения плотности моделируемой системы прежде, чем совершить следующее изменение координат идеальной стенки. Данный параметр позволяет задавать интервал изменения плотности системы, что даёт нам возможность изменять плотность на столько медленно, на сколько это требуется условиями эксперимента.

- Число K2 шагов сжатия (сдвигов стенки), после которого необходимо провести дополнительный расчёт динамики системы (соударений частиц).

- Число K3 соударений на частицу, которое необходимо рассчитать после каждых K2 шагов сжатия системы.

Пример:

compress_left_wall 0.503423 0.000001 101010000

Данная команда изменит относительную плотность частиц в системе до значения 0.503423, при этом максимальное расстояние, на которое будет меняться расстояние между частицами и идеальной стенкой будет не более $\Delta L_{max} = 0.000001$ и после каждого небольшого изменения плотности системы будет произведён расчёт динамики движения частиц в системе на протяжении десяти соударений на каждую частицу в этой системе.

Короткая запись команды:

Данная команда имеет вариант короткой записи, которая имеет те же аргументы и работает так же, как и полная запись команды:

compressl 0.49 0.503423 0.000001 101010000

13.5.2 *compress_right_wall*

Управляющая команда *compress_right_wall* позволяет изменять параметры моделируемой системы, симулируя движение "правой"идеальной стенки (данная стенка условно называется правой, т.к. на профиле плотности, где представлены данные о значениях относительной плотности в системе, данная граница системы расположена справа).

На каждом шаге изменения параметров моделируемой системы программа определяет наименьшее расстояние Δr поверхности частиц в системе до идеальной стенки, ограничивающей моделируемую систему в плоскости $x = L$ и изменяет координаты этой идеальной стенки на величину, меньшую Δr. Если расстояние Δr больше, чем задаваемое пользователем максимально допустимое смещение стенки ΔL_{max}, то стенка смещается на расстояние ΔL_{max}.

В результате изменения значения параметра системы L идеальная стенка, ограничивающая моделируемую нами систему в плоскости $x = L$, становится ближе (или дальше, это зависит о того, хотим ли мы увеличить относительную плотность частиц в системе или хотим уменьшить её) ко всем частицам в системе на расстояние ΔL, расстояние между частицами и второй идеальной стенкой, ограничивающей систему в плоскости $x = 0$ не изменяется.

Таким образом мы моделируем изменение координаты "правой"идеальной стенки, приближая её ко всем частицам в системе или отдаляя эту границу от них.

Величина ΔL, на которую мы изменяем расстояние между частицами и стенкой не может быть больше расстояния между идеальной стенкой и самой ближней к ней частице, а так же ограничивается значением максимально допустимого смещения стенки, задаваемого в качестве аргумента для данной команды.

Аргументы:

- Относительная плотность системы, которую необходимо установить в системе с помощью движения идеальной стенки. Указываемая относительная плотность частиц в системе может быть как больше, так и меньше текущей плотности в системе, в зависимости от чего с помощью данной команды можно как сжимать моделируемую систему, уменьшая объём, в котором находятся частицы, так и наоборот, расширять данную системы, отодвигая идеальную стенку от частиц и уменьшая относительную плотность системы.

- Значение максимально допустимого смещения стенки ΔL_{max}. Если ближайшая к идеальной стенке частица находится на расстоянии, большем ΔL_{max}, то идеальная стенка смещается только на расстояние ΔL_{max}.

- Число K1 соударений на частицу, которое необходимо рассчитать после каждого маленького изменения плотности моделируемой системы прежде, чем совершить следующее изменение координат идеальной стенки. Данный параметр позволяет задавать интервал изменения плотности систе-

мы, что даёт нам возможность изменять плотность на столько медленно, на сколько это требуется условиями эксперимента.

- Число K2 шагов сжатия (сдвигов стенки), после которого необходимо провести дополнительный расчёт динамики системы (соударений частиц).

- Число K3 соударений на частицу, которое необходимо рассчитать после каждых K2 шагов сжатия системы.

Пример:

compress_right_wall 0.49 0.0000001 10001010000

Данная команда изменит относительную плотность частиц в системе до значения 0.49000, при этом максимальное расстояние, на которое будет меняться расстояние между частицами и идеальной стенкой, будет не более $\Delta L_{max} = 0.0000001$ и после каждого небольшого изменения плотности системы будет произведён расчёт динамики движения частиц в системе на протяжении тысячи соударений на каждую частицу в этой системе.

Короткая запись команды:

Данная команда имеет вариант короткой записи, которая имеет те же аргументы и работает так же, как и полная запись команды:

compressr 0.49 0.0000001 10001010000

13.5.3 *compress_both_walls*

Данная команда позволяет изменять плотность моделируемой системы, двигая одновременно обе идеальные стенки на одинаковое расстояние. Расстояние, на которое необходимо сдвинуть обе стенки выбирается на основании минимального расстояния частиц до обеих идеальных стенок.

Аргументы:

- Относительная плотность системы, которую необходимо установить в системе с помощью движения идеальных стенок. Указываемая относительная плотность частиц в системе может быть как больше, так и меньше текущей плотности в системе, в зависимости от чего с помощью данной команды можно как сжимать моделируемую систему, уменьшая объём, в котором находятся частицы, так и наоборот, расширять данную системы, отодвигая идеальные стенки от частиц и уменьшая относительную плотность системы.

- Значение максимально допустимого смещения стенок ΔL_{max}. Если ближайшая к идеальной стенке частица находится на расстоянии, большем ΔL_{max}, то идеальная стенка смещается только на расстояние ΔL_{max}.

- Число K1 соударений на частицу, которое необходимо рассчитать после каждого маленького изменения плотности моделируемой системы прежде, чем совершить следующее изменение координат идеальных стенок. Данный параметр позволяет задавать интервал изменения плотности системы, что даёт нам возможность изменять плотность на столько медленно, на сколько это требуется условиями эксперимента.

- Число K2 шагов сжатия (сдвигов стенок), после которого необходимо провести дополнительный расчёт динамики системы (соударений частиц).

- Число K3 соударений на частицу, которое необходимо рассчитать после каждых K2 шагов сжатия системы.

Пример:

compress_both_walls 0.49 0.0000001 10001010000

Данная команда изменит относительную плотность частиц в системе до значения 0.49000, при этом максимальное расстояние, на которое будет меняться расстояние между частицами и идеальными стенками, будет не более $\Delta L_{max} = 0.0000001$ и после каждого небольшого изменения плотности системы будет произведён расчёт динамики движения частиц в системе на протяжении тысячи соударений на каждую частицу в этой системе.

Короткая запись команды:

Данная команда имеет вариант короткой записи, которая имеет те же аргументы и работает так же, как и полная запись команды:

compresst 0.49 0.0000001 10001010000

13.6 image

Управляющая команда *image* собирает информацию об относительной плотности частиц в системе и сохраняет эту информацию в текстовый файл.

Необходимо описать подробно.

13.7 profile

Управляющая команда $profile$ сохраняет информацию о положениях центров частиц в некотором "слое"системы, что позволяет изучать структуру системы жёстких сфер в некоторой выделенной области моделируемой системы. Информация о положении центров частиц в некотором "слое"системы, ограниченным плоскостями $x_1 = const$ и $x_2 = const$, называется "разрезом т.к. позволяет нам увидеть положение центров частиц, находящихся в данном "слое"в проекции на плоскость OYZ.

Необходимо описать подробно.

14 Глоссарий

15 Список литературы

[1] В.П. Вешнев Т.А. Нурлыгаянов. Кристаллизация твердых сфер вблизи стенки // Письма в ЖТФ. 2011. Т. 37, № 18.

[2] S. Miller S. Luding. Event-driven molecular dynamics in parallel // Journal of Computational Physics. 2003. Т. 193. с. 306–316.

[3] Rapaport Dennis C. The Event-Driven Approach to N-Body Simulation // Progress of Theoretical Physics Supplement. 2009. № 178. С. 5–14.

[4] Белкин А. А. ОБ ОДНОЙ МОДИФИКАЦИИ МЕТОДА МОЛЕКУЛЯРНОЙ ДИНАМИКИ // Сибирский журнал индустриальной математики. 2006. Октябрь–декабрь. Т. IX, № 4(28). С. 27–32.

[5] Rosenbluth Marshall N., Rosenbluth Arianna W. Further Results on Monte Carlo Equations of State // J. Chem. Phys. 22, 881 (1954); http://dx.doi.org/10.1063/1.1740207. 1954.

[6] Wood W. W., Jacobson J. D. Preliminary Results from a Recalculation of the Monte Carlo Equation of State of Hard Spheres // J. Chem. Phys. 1957. Т. 27. с. 1207.

[7] Alder B. J., Wainwright T. E. Phase Transition for a Hard Sphere System // J. Chem. Phys. 1957. Т. 27. с. 1208.

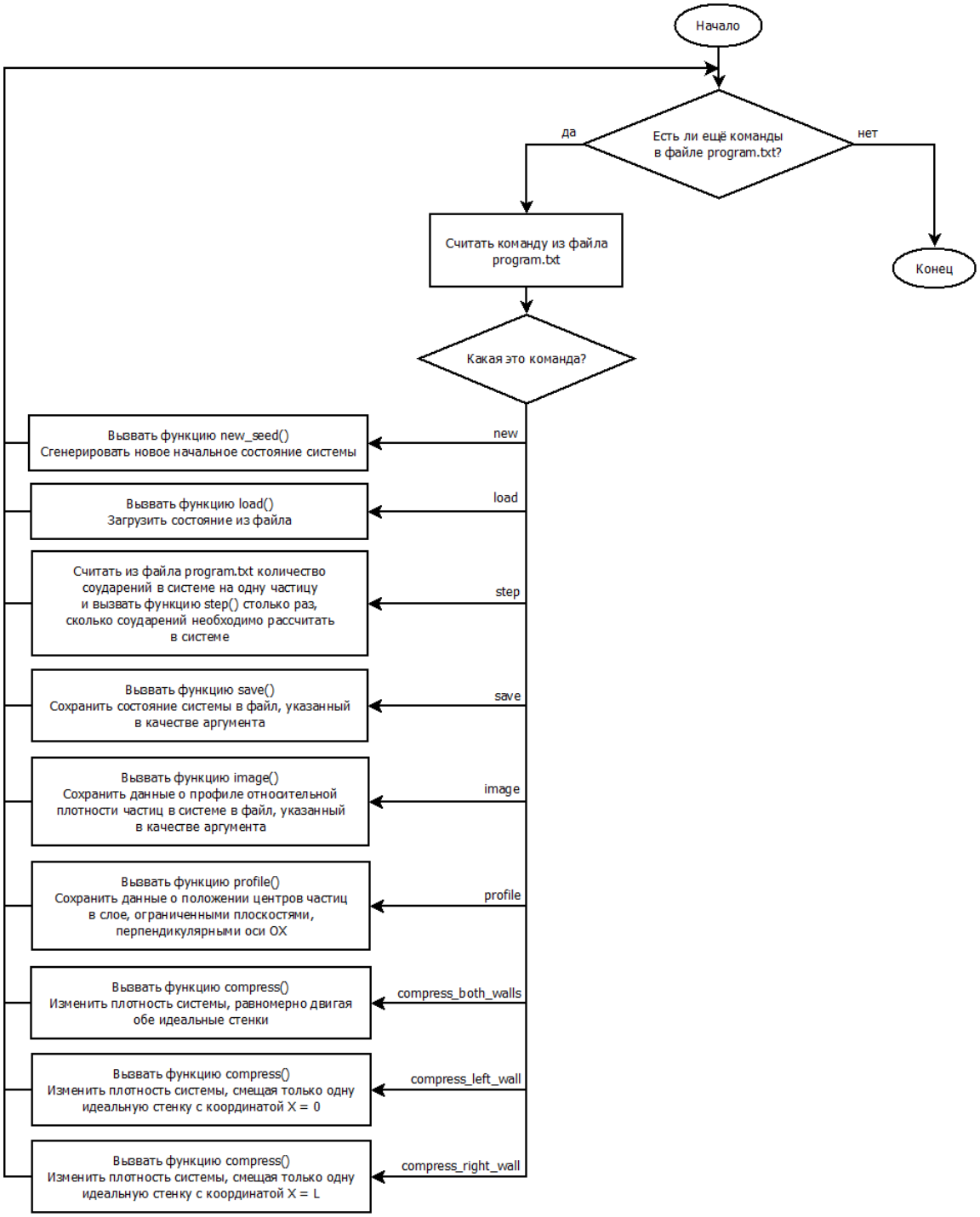
[8] Джанколи Д. Физика. Т. I. Мир, 1989.

16 Приложения

16.1 Блок схемы функций и процедур

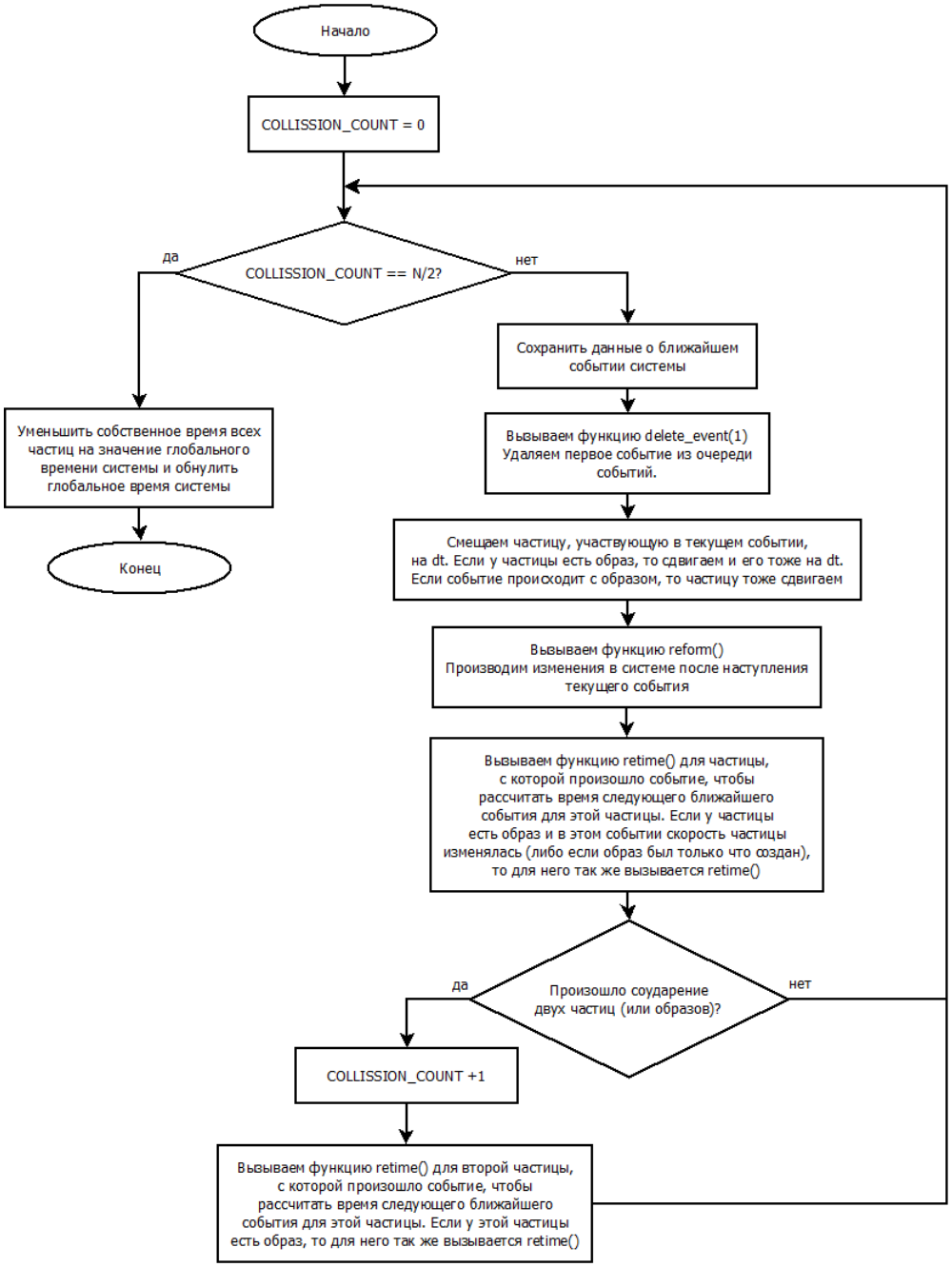
16.1.1 Функция init(). Контроль эксперимента

Функция проведения эксперимента по указанному в текстовом файле описанию.



16.1.2 Функция step(). Основной цикл программы

Функция 'шаг', основной цикл программы, производит расчёт динамики системы в течении 1 соударения на каждую частицу, т.е. N/2 столкновений в системе, где N - количество частиц в системе.



16.2 Исходный код программы с комментариями

```
/*
ИМПОРТИРУЕМЫЕ МОДУЛИ
*/
#include "stdafx.h"  // стандартная библиотека
#include <time.h>    // библиотека для подстчёта времени
#include <fstream>   // библиотека для работы с файлами
#include <cmath>     // библиотека для вызова математических функций


/*
ОБЪЯВЛЕНИЕ ПЕРЕМЕНЫХ
*/

// Количество ячеек по Y, Z и X(K2), на которые разбивается объём.
// Количество ячеек расчитывается динамически при загрузке системы
// или при посеве (см. функцию load).
short K, K2;

// число частиц во всём объёме, переопределяется в функциях load и new_seed
int NP = 6976;

// глобальный счётчик столкновений в системе
int COLL_COUNT = 0;

#define PI 3.141592653589793238462

// параметры объёма, задаются в load()
double A, A2, dA, L, dL;

// Глобальная переменная для подсчёта общей кинетической энергии всех частиц
double global_E = 0.0;

// индекс последнего элемента в очереди событий.
int last;

// объект "событие"
```

```
typedef struct Event_ {
        double t;
        int im, jm;
} Event;

// очередь событий - оптимально 8192*2 элемента
// (это должно быть число-степень двойки, большее чем максимальное число частиц)
Event time_queue[16384];

// обзект "частица"
// x, y, z - координаты частицы
// vx, vy, vz - проекции скоростей частицы
// t - собственное время частицы
// dt - время до ближайшего события этой частицы
// x_box, y_box, z_box - номер ячейки, в которой находится частица
// ti - номер события частицы в дереве событий
// box_i - номер частицы в ячейке
// i_copy - номер образа данной частицы, равно -1 если образа не существует
typedef struct particle_ {
        double x, y, z, vx, vy, vz, t, dt;
        int x_box, y_box, z_box, ti, box_i, i_copy;
} particle;

// массив частиц, размер массива N*2 + округление
// в большую сторону к числу дающее степень двойки.
particle particles[16384];

// клетка. Объём системы разделён на множество клеток,
// каждая клетка содержит в себе несколько виртуальных частиц
// x1, y1, z1, x2, y2, z2 - координаты конца и начала каждой ячейки
// particles[100] - список всех частиц, находящихся в данной ячейке
// end - индекс последней частицы в списке частиц данной ячейки
typedef struct Box_ {
        double x1, y1, z1, x2, y2, z2;
        int particles[12];
        short end;
} Box;
```

```
// массив клеток для всего объёма
Box boxes_yz[16][16][64];


// массив с номерами частиц, для которых надо сохранять историю событий
// используется на случай отладки программы для сохранения истории событий
// выбранных частиц
int particles_for_check[100];
int particles_for_check_count = 0;

/*
   Эта функция выводит на экран параметры текущей системы:
   A - размер системы (области объёма, где может находиться центр частицы) по Y и Z.
       здесь мы умножем A на 2.0, т.к. центр моделируемой системы находится в 0,
           а периодические границы находятся в плоскостях y = A, y = -A, z = A, z = -A.
   L
   N - число частиц в системе
   etta - средняя относительная плотность системы
*/
void print_system_parameters() {
        long double etta = (PI * NP) / (6.0 * A * A * (L - 1.0));
        printf("\n\n| A     | %.15le |\n| L     | %.15le |\n", 2.0 * A, 2.0 * L);
        printf("| N     | %d                    |\n", NP);
        printf("| etta  | %.15le |\n", etta);
}

/*
   Функция возвращает наибольшее собственное время частиц в системе,
   что позволяет синхронизовать все частицы по времени
*/
double get_maximum_particle_time() {
        double t_max = -1.0e+20;

        for (int i = 0; i < NP; ++i) {
                if (particles[i].t > t_max)
                        t_max = particles[i].t;
```

```
            if ((particles[i].i_copy >= 0) && (particles[particles[i].i_copy].t > t_max))
                t_max = particles[i].t;
        }

        return t_max;
}

/*
   Функция для проверки состояния системы, в ней мы провекряем, что
   все частицы находятся внутри системы, для каждой частицы у нас рассчитано
   ближайшее событие, частицы находятся в правильных ячейках системы и пр.

   В случае возникновения любых проблем данная функция прекращает работу программы
   и выводит дополнительную информацию об обнаруженной проблеме.

   Функция работает медленно, необходимо использовать в целях проверки
   изменений в программе.
*/
int check_particles() {
        double E = 0.0;
        double dt = 0.0;
        double t_global = get_maximum_particle_time();

        for (int i = 0; i < NP; i++) {
            particle p1 = particles[i];
            Box p1_box = boxes_yz[p1.y_box][p1.z_box][p1.x_box];

            dt = t_global - p1.t;
            p1.x += p1.vx * dt;
            p1.y += p1.vy * dt;
            p1.z += p1.vz * dt;

            E += p1.vx*p1.vx + p1.vy*p1.vy + p1.vz*p1.vz;

            // проверяем индексы ячеек для всех частиц
            if ((p1.x_box > K2) || (p1.y_box > K) || (p1.z_box > K) ||
                    (p1.x_box < 0) || (p1.y_box < 0) || (p1.z_box < 0)) {
```

```
            throw "Particle locates in incorrect cell.";
        }

        // проверяем что частицы находятся в объёме
        if (((abs(p1.x) - L) > 1.0e-14) ||
            ((abs(p1.y) - A) > 1.0e-14) ||
            ((abs(p1.z) - A) > 1.0e-14)) {
            printf("\n Particle %d, %.15le, %.15le, %.15le \n", i, p1.x, p1.y, p1.z);
            throw "Particle is out of the system boundaries.";
        }

        // проверяем что частицы находятся в правильных ячейках
        if (((p1.x < p1_box.x1) && (abs(p1.x - p1_box.x1) > 1.0e-14)) ||
            ((p1.x > p1_box.x2) && (abs(p1.x - p1_box.x2) > 1.0e-14)) ||
            ((p1.y < p1_box.y1) && (abs(p1.y - p1_box.y1) > 1.0e-14)) ||
            ((p1.y > p1_box.y2) && (abs(p1.y - p1_box.y2) > 1.0e-14)) ||
            ((p1.z < p1_box.z1) && (abs(p1.z - p1_box.z1) > 1.0e-14)) ||
            ((p1.z > p1_box.z2) && (abs(p1.z - p1_box.z2) > 1.0e-14))) {

            printf("Vilet za granicy %d \n", i);

            printf("Granizy:\n");
            printf("X : [%.15le ; %.15le]\n", p1_box.x1, p1_box.x2);
            printf("Y : [%.15le ; %.15le]\n", p1_box.y1, p1_box.y2);
            printf("Z : [%.15le ; %.15le]\n", p1_box.z1, p1_box.z2);
            printf("x, y, z: %.15le %.15le %.15le\n", p1.x, p1.y, p1.z);

            printf("p1.t = %.15le , p.im = %d, p1.jm = %d \n", p1.t, time_queue[p1.ti].im, time_queue[p1.ti].jm);

            throw "Particle is out of the cell boundary.";
        }

        // Если частица имеет образ, то образ должен существовать
        if ((p1.i_copy > -1) && (particles[i + NP].i_copy == -1)) {
            throw "Particle has incorrect image!";
        }
```

```
            // Проверяем что частица записана в одную из ячеек в системе
            bool w = false;
            for (int ty = 0; ty <= boxes_yz[p1.y_box][p1.z_box][p1.x_box].end; ++ty) {
                    if (boxes_yz[p1.y_box][p1.z_box][p1.x_box].particles[ty] == i) w = true;
            }
            if (w == false) {
                    for (int t = 0; t <= boxes_yz[p1.y_box][p1.z_box][p1.x_box].end; ++t) {
                            printf(" %d ", boxes_yz[p1.y_box][p1.z_box][p1.x_box].particles[t]);
                    }

                    throw "Particle doesn't store in the cell.";
            }

            // Проверяем на правильное ли событие в линейке времён ссылается частица
            if (time_queue[p1.ti].im != i && time_queue[p1.ti].jm != i) {
                    Event e = time_queue[p1.ti];
                    printf("\n i = %d ; im = %d ; jm = %d ; ti = %d ", i, e.im, e.jm, p1.ti);
                    throw "Particle has no correct link to the event.";
            }
    }

    // Проверяем текущее значение глобальной кинетической энергии системы со
    // значением энергии, которое было при загрузке системы из файла в load
    if (abs(E - global_E) > 0.1e-8) {
            printf("\nENERGY was changed: \n E_seed = %.15le \n E_now= %.15le \n", global_E, E);
            throw "ENERGY was changed.";
    }

    // Проверяем что в очереди событий нет событий для несуществующих образов
    for (int i = 0; i < last; i++) {
            if ((time_queue[i].im >= NP) && (particles[time_queue[i].im - NP].i_copy == -1)) {
                    printf("\n Time tree event # %d ", i);
                    printf("\n im, jm = %d %d \n", time_queue[i - 2].im, time_queue[i - 2].jm);
                    printf("\n im, jm = %d %d \n", time_queue[i - 1].im, time_queue[i - 1].jm);
                    printf("\n im, jm = %d %d \n", time_queue[i].im, time_queue[i].jm);
                    printf("\n im, jm = %d %d \n", time_queue[i + 1].im, time_queue[i + 1].jm);
                    printf("\n particle 34 event # %d \n", particles[34].ti);
```

```
                printf("\n last = %d \n", last);
                throw "Incorrect event!";
            }
            if ((time_queue[i].jm >= NP) && (particles[time_queue[i].jm - NP].i_copy == -1)) {
                throw "Incorrect event!";
            }
        }

        return 0;
}


/*
  Функция подъёма элемента по очереди событий к началу очереди

  Аргументы:
    i - позиция, на которой находится элемент в данный момент
    t - время до наступления данного события
*/
int get_up(int i, double &t) {
        int j = i >> 1;  // это сдвиг вправо, то же самое что j = i/2, только быстрее
        while (i > 1) {
            if (i % 2 != 0 && t < time_queue[i - 1].t) {
                particles[time_queue[i - 1].im].ti = i;
                if (time_queue[i - 1].jm >= 0) particles[time_queue[i - 1].jm].ti = i;
                time_queue[i] = time_queue[i - 1];
                i--;
            }
            if (time_queue[j].t > t) {
                particles[time_queue[j].im].ti = i;
                if (time_queue[j].jm >= 0) particles[time_queue[j].jm].ti = i;
                time_queue[i] = time_queue[j];
                i = j;
                j >>= 1;
            }
            else return i;
        }
```

```
        return 1;
}


/*
    Функция добавления события в очередь событий

    Аргументы:
     i - частица, с участием которой произойдёт новое событие
     j - номер частицы с которой столкнётся частица i или номер события
         соударения со стенкой или прохождения через периодические
             граничные условия или между ячейками системы
*/
void add_event(int &i, int &j) {
        /*
            Рассчитываем полное время нового события от начала отсчёта
            глобального времени системы, таким образом мы получаем время t,
            сравнивая которое мы можем определить какое из событий в системе
            произойдёт раньше
        */
        double t = particles[i].t + particles[i].dt;

        /*
            Находим позицию в дереве времён для нового события.
            Изначально помещаем это событие вниз дерева и позволяем
            ему подняться по дереву, если данное событие произойдёт раньше чем
            другие события
        */
        particles[i].ti = get_up(last, t);

        /*
            Если новое событие - это событие столкновения двух частиц, то
            необходимо для второй частицы сохранить данные о её новом событии
        */
        if (j >= 0) {
                particles[j].dt = particles[i].dt;
                particles[j].ti = particles[i].ti;
```

```
        }

        // записываем новое событие в выбранную ячейку в дереве времён
        time_queue[particles[i].ti].im = i;
        time_queue[particles[i].ti].jm = j;
        time_queue[particles[i].ti].t = t;

        // увеличиваем число событий на 1
        last++;
}


/*
   Функция удаления события из очереди событий

   Аргументы:
    i - позиция удаляемого элемента в очереди событий
*/
void delete_event(int i) {
        int j = i << 1;
        while (j < last) {
                if (i % 2 == 0 && time_queue[i + 1].t < time_queue[j].t) {
                        particles[time_queue[i + 1].im].ti = i;
                        if (time_queue[i + 1].jm >= 0) particles[time_queue[i + 1].jm].ti = i;
                        time_queue[i] = time_queue[i + 1];
                        i++;
                        j = i << 1;
                }
                else {
                        particles[time_queue[j].im].ti = i;
                        if (time_queue[j].jm >= 0) particles[time_queue[j].jm].ti = i;
                        time_queue[i] = time_queue[j];
                        i = j;
                        j = i << 1;
                }
        }
```

```
        if (i < last - 1 && i % 2 == 0) {
                particles[time_queue[i + 1].im].ti = i;
                if (time_queue[i + 1].jm >= 0) particles[time_queue[i + 1].jm].ti = i;
                time_queue[i] = time_queue[i + 1];
                i++;
        }

        if (i < last - 1) {
                j = get_up(i, time_queue[last - 1].t);
                particles[time_queue[last - 1].im].ti = j;
                if (time_queue[last - 1].jm >= 0) particles[time_queue[last - 1].jm].ti = j;
                time_queue[j] = time_queue[last - 1];
        }

        last--;
}


/*
   Функция удаления событий частицы из линейки событий

   Аргументы:
    i - номер частицы или образа, события которого необходимо удалить
*/
void clear_particle_events(int &i) {
        // particles[i].ti - ссылка на индекс события, которое хранит сама частица
        int f = particles[i].ti;

        if (f > 0) {
                int e = -100;
                int kim = time_queue[f].im;
                int kjm = time_queue[f].jm;

                if (time_queue[f].im == i) {
                        /*
                           Если мы удаляем событие столкновения двух частиц, то
                           необходимо переместить вторую частицу в то же время,
```

```
                    в котором находится частица, для которой мы удаляем
                    это событие и заменить событие столкновения двух частиц
                    на событие "-100", когда вторая частица просто долетит до
                    места предполагаемого столкновения и после этого для неё
                    будет рассчитано новое событие.
                */
                if (time_queue[f].jm >= 0) {
                        double dt = particles[kim].t - particles[kjm].t;  // разница в текущем времени

                        particles[kjm].x += particles[kjm].vx*dt;
                        particles[kjm].y += particles[kjm].vy*dt;
                        particles[kjm].z += particles[kjm].vz*dt;
                        particles[kjm].t = particles[kim].t;
                        particles[kjm].dt = (particles[kjm].dt - dt) / 1.1;

                        delete_event(f);
                        add_event(kjm, e);
                }
                else delete_event(f);
        }
        else if (time_queue[f].jm == i) {
                double dt = particles[kjm].t - particles[kim].t;  // разница в текущем времени

                particles[kim].x += particles[kim].vx*dt;
                particles[kim].y += particles[kim].vy*dt;
                particles[kim].z += particles[kim].vz*dt;
                particles[kim].t = particles[kjm].t;
                particles[kim].dt = (particles[kim].dt - dt) / 1.1;

                delete_event(f);
                add_event(kim, e);
        }
        particles[i].ti = 0;
    }
}
```

```
/*
    Функция расчёта ближайшего события для частицы

    Аргументы:
     i - номер частицы, для которой мы должны рассчитать ближайшее событие
*/
void retime(int &i) {
        particle p1 = particles[i];
        Box p1_box = boxes_yz[p1.y_box][p1.z_box][p1.x_box];
        int jm;  // переменная для сохранения типа ближайшего события
        double dt, dt_min;  // переменные для рассчёта времени для ближайшего события

        clear_particle_events(i);

        if (p1.vx < 0.0) {
                dt_min = (p1_box.x1 - p1.x) / p1.vx;
                jm = -2;  // событие пересечения границы X1 ячейки, в которой находится частица

                // если мы находимся вблизи идеальной стенки, то рассчитать время соударения
                // с идеальной стенкой
                if (p1.x_box == 1) {
                        dt_min = (p1_box.x1 + 1.0 - p1.x) / p1.vx;
                        jm = -1;  // событие столкновения с идеальной стенкой
                }
        }
        else {
                dt_min = (p1_box.x2 - p1.x) / p1.vx;
                jm = -4;  // событие пересечения границы X2 ячейки, в которой находится частица

                // если мы находимся вблизи идеальной стенки, то рассчитать время соударения
                // с идеальной стенкой
                if (p1.x_box == K2 - 1) {
                        dt_min = (p1_box.x2 - 1.0 - p1.x) / p1.vx;
                        jm = -1;  // событие столкновения с идеальной стенкой
                }
        }
```

```
if (p1.vy < 0.0) {
        dt = (p1_box.y1 - p1.y) / p1.vy;
        if (dt < dt_min) {
                dt_min = dt;
                jm = -5;  // событие пересечения границы Y1 ячейки, в которой находится частица
        }
        if ((p1.y_box == 1) && (i < NP)) {
                dt = (p1_box.y1 + 1.0 - p1.y) / p1.vy;
                if ((dt > 0) && (dt < dt_min)) {
                        dt_min = dt;
                        jm = -15;  // событие рождения образа частицы
                }
        }
}
else {
        dt = (p1_box.y2 - p1.y) / p1.vy;
        if (dt < dt_min) {
                dt_min = dt;
                jm = -6;  // событие пересечения границы Y2 ячейки, в которой находится частица
        }
        if ((p1.y_box == K - 1) && (i < NP)) {
                dt = (p1_box.y2 - 1.0 - p1.y) / p1.vy;
                if ((dt > 0) && (dt < dt_min)) {
                        dt_min = dt;
                        jm = -16;  // событие рождения образа частицы
                }
        }
}

if (p1.vz < 0.0) {
        dt = (p1_box.z1 - p1.z) / p1.vz;
        if (dt < dt_min) {
                dt_min = dt;
                jm = -7;  // событие пересечения границы Z1 ячейки, в которой находится частица
        }
        if ((p1.z_box == 1) && (i < NP)) {
                dt = (p1_box.z1 + 1.0 - p1.z) / p1.vz;
```

```
                    if ((dt > 0) && (dt < dt_min)) {
                            dt_min = dt;
                            jm = -17;  // событие рождения образа частицы
                    }
            }
    }
    else {
            dt = (p1_box.z2 - p1.z) / p1.vz;
            if (dt < dt_min) {
                    dt_min = dt;
                    jm = -8;  // событие пересечения границы Z2 ячейки, в которой находится частица
            }
            if ((p1.z_box == K - 1) && (i < NP)) {
                    dt = (p1_box.z2 - 1.0 - p1.z) / p1.vz;
                    if ((dt > 0) && (dt < dt_min)) {
                            dt_min = dt;
                            jm = -18;  // событие рождения образа частицы
                    }
            }
    }

    double temp, dx, dy, dz, dvx, dvy, dvz, d, dv, bij;
    int s, n, r, q, w;

    // Проходим по ячейкам, ближайшим к ячейке, в которой находится частица i
    for (r = p1.x_box - 1; r < p1.x_box + 2; ++r)
            for (q = p1.y_box - 1; q < p1.y_box + 2; ++q)
                    for (w = p1.z_box - 1; w < p1.z_box + 2; ++w) {

                            // если индекс ячейки выходит за границу системы то
                            // переходим на следующий шаг цикла
                            if (q == -1) {
                                    continue;
                            }
                            if (q == K + 1) {
                                    continue;
                            }
```

```
if (w == -1) {
        continue;
}
if (w == K + 1) {
        continue;
}

// Проходим по всем частицам в выбранной ячейке и проверяем возможность
// столкновения этих частиц с частицей i
for (s = 0; s <= boxes_yz[q][w][r].end; ++s) {
        n = boxes_yz[q][w][r].particles[s];

        // не рассчитываем столкновения частицы с собственным образом
        if (n == p1.i_copy) continue;

        /*
          Сохраняем в переменную p все данные частицы n, чтобы далее
          чтобы далее записывать все операции короче и без обращения
          в глобальную память
        */
        particle p = particles[n];

        /*
            Рассчитываем разницу собственного времени двух частиц,
            это необходимо чтобы синхронизовать их между собой и рассчитывать
            возможность и время столкновения в системе, где обе частицы будут
            иметь одинаковое собственное время
        */
        temp = p1.t - p.t;

        dvx = p.vx - p1.vx;
        dvy = p.vy - p1.vy;
        dvz = p.vz - p1.vz;

        /*
            Перемещаем частицу p во время частицы p1 (i-тая частица)
            мы можем это делать так как собственное время частицы p1 гарантированно
```

```
    либо больше либо равно собственному времени частицы p, т.к. с частицей
    p1 только что произошло событие и собственное время частицы p1 совпадает
    с глобальным временем в системе
*/
dx = p.x + p.vx * temp - p1.x;
dy = p.y + p.vy * temp - p1.y;
dz = p.z + p.vz * temp - p1.z;

bij = dx * dvx + dy * dvy + dz*dvz;
if (bij < 0.0) {
        dv = dvx * dvx + dvy * dvy + dvz*dvz;
        // рассчитываем дискриминант в уравнении для вычисления времени соударения
        d = bij * bij + dv * (4.0 - dx * dx - dy * dy - dz * dz);

        // если дискриминант больше нуля то соударение возможно
        if (d > 0.0) {
                dt = -(sqrt(d) + bij) / dv;

                /*
                    Сценарии, при котором возможны отрицательные времена:
                    1. Образ, вставленный в систему для мгновенного соударения может
                    в другие частицы, после мгновенного соударения с частицей из обл
                    образ будет уничтожен.
                    2. Частица n1 сталкивается с частицей n2, мы создаем образ для ч
                    для которого не находится места и мы сталкиваем его с другой час
                    в результате чего скорость частицы n1 снова меняется и время соу
                    частиц n1 и n2 может быть отрицательным (-1*10-14) из за погрешн
                    в расчете координат в 15ом знаке.

                    В любом случае мы не должны разрешать отрицательные времёна
                    и если отклонение от нуля мало то полагаем время соударения равны
                    т.е. частицы уже соприкасаются между собой.
                */
                if (dt > -1.0e-12 && dt < 1.0e-15) dt = 0.0;

                /*
                    К разнице в собственном времени частиц прибавляем время до их соу
```

```
                                        в результате получаем время dt через которое это соударение произ
                                        частицы p, таким образом мы получаем возможность сравнить время п
                                        рассчитанного соударения со временем ближайшего события для части
                                        принять решение - должны ли мы перезаписать событие для частицы p
                                    */
                                    temp += dt;

                                    if ((dt < dt_min) && (dt >= 0.0) &&
                                            ((temp < p.dt) || ((abs(temp - p.dt) < 0.1E-15) &&
                                                    (time_queue[p.ti].im == n) &&
                                                    (time_queue[p.ti].jm == -100)))) {
                                            dt_min = dt;
                                            jm = n;
                                    }
                                }
                            }
                        }
                    }
                }

        /*
            Если новое событие - это столкновение двух частиц, то для второй частицы
            необходимо перезаписать её предыдущее ближайшее событие и синхронизовать
            эту частицу по времени с частицей i
        */
        if (jm >= 0) {
                dt = p1.t - particles[jm].t;
                if (dt < 0.0 && dt > -1.0e-15)
                        dt = 0.0;

                particles[jm].t = p1.t;
                particles[jm].dt = dt_min;
                particles[jm].x += particles[jm].vx * dt;
                particles[jm].y += particles[jm].vy * dt;
                particles[jm].z += particles[jm].vz * dt;

                /*
                    Перед тем как добавить новое событе соударения двух частиц чистим
```

```
                линейку времен для второй частицы, которая будет учавствовать в новом соударении
            */
            clear_particle_events(jm);
        }

        particles[i].dt = dt_min;

        add_event(i, jm);

        /*
            В случае если при расчёте ближайшего события мы получили отрицательное время
            необходимо прервать выполение программы и распечатать отладочную информацию
            о рассчитанном событии
        */
        if (dt_min < -1.0e-11) {
            printf("\n retime result: %d %d, %.16le\n ", i, jm, dt_min);
            printf("\n p1.x = %.16le , p1.y = %.16le , p1.z = %.16le \n", p1.x, p1.y, p1.z);
            printf("\n p1.x_box = %d, p1.y_box = %d, p1.z_box = %d", p1.x_box, p1.z_box, p1.y_box);
            printf("\n im = %d, jm = %d, dt = %.16le , A = %.16le", i, jm, dt_min, A);
            printf("\n p1.box.x = [%.16le; %.16le]", boxes_yz[p1.y_box][p1.z_box][p1.x_box].x1, boxes_yz[p1.y_box][p1.z_
            printf("\n p1.box.y = [%.16le; %.16le]", boxes_yz[p1.y_box][p1.z_box][p1.x_box].y1, boxes_yz[p1.y_box][p1.z_
            printf("\n p1.box.z = [%.16le; %.16le]", boxes_yz[p1.y_box][p1.z_box][p1.x_box].z1, boxes_yz[p1.y_box][p1.z_
            throw "dt < 0!";
        }
    }


    /*
        Функция поиска столкновения для образа, который мы не можем вставить в систему,
        т.к. для него нет свободного места.

        Перед вызовом этой функции мы устанавливаем некоторую случайным образом
        выбранную координату X для образа и в этой функции ищем варианты столкновения
        нового образа и частиц, находящихся вдоль линии движения образа.

        Аргументы:
          i - номер образа, который необходимо столкнуть с частицей из системы
```

```
*/
int search_collission_for_new_virtual_particle(int &i) {
        particle p1 = particles[i];
        particle p2;
        double dt, dx, dy, dz, dvx, bij, d, dv, dvy, dvz, fa, fb, fc, fD, sD, t1, t2;

        // Проходим по списку всех частиц в системе
        for (int j = 0; j < NP; j++) {
                // исключая частицу, которой принадлежит данный образ
                if (j == i - NP) continue;

                p2 = particles[j];
                // рассчитываем разницу в собственном времени между частицей j и образом i
                dt = p1.t - p2.t;

                // EN: if we have small time differences for these two particles
                // we shouldn't use it for collission because other virtual particle
                // can use the same particle in the same time.
                // RU: если у частиц нет разницы во времени то мы не должны рассматривать столкновение
                // этих частиц
                if (dt == 0.0) continue;

                // перемещаем частицу в то же время, в котором находится образ
                p2.x += p2.vx*dt;
                p2.y += p2.vy*dt;
                p2.z += p2.vz*dt;

                /*
                   Рассчитываем находится ли выбранная частица вблизи линии скорости нового образа.
                   Если да, то рассчитываем время их столкновения.
                */
                fa = p1.vx*p1.vx + p1.vy*p1.vy + p1.vz*p1.vz;
                fb = 2.0*p1.x*p1.vx + 2.0*p1.y*p1.vy + 2.0*p1.z*p1.vz - 2.0*p2.x*p1.vx - 2.0*p2.y*p1.vy - 2.0*p2.z*p1.vz;
                fc = p2.x*p2.x + p1.x*p1.x + p2.y*p2.y + p1.y*p1.y + p2.z*p2.z + p1.z*p1.z - 2.0*p1.x*p2.x - 2.0*p1.y*p2.y -

                fD = fb*fb - 4.0*fa*fc;
                if (fD > 0) {
```

```
sD = sqrt(fD);

/*
   Существует две возможные точки соударения двух частиц, нам необходимо определить
   в какой из них столкновение произойдёт, если частицы будут двигаться в пространстве
   с их текущими скоростями.
*/
t1 = (-fb - sD) / (2.0*fa);
t2 = (-fb + sD) / (2.0*fa);

dx = p2.x - p1.x - p1.vx*t1;
dy = p2.y - p1.y - p1.vy*t1;
dz = p2.z - p1.z - p1.vz*t1;

dvx = p2.vx - p1.vx;
dvy = p2.vy - p1.vy;
dvz = p2.vz - p1.vz;

bij = dx * dvx + dy * dvy + dz*dvz;
if (bij < 0.0) {
        dv = dvx * dvx + dvy * dvy + dvz*dvz;
        d = bij * bij + dv * (4.0 - dx * dx - dy * dy - dz * dz);

        dt = -(sqrt(d) + bij) / dv;

        if (dt > 0.0) {
                particles[i].x += p1.vx*t1;
                particles[i].y += p1.vy*t1;
                particles[i].z += p1.vz*t1;

                particles[j].x = p2.x;
                particles[j].y = p2.y;
                particles[j].z = p2.z;

                dt = particles[i].t - particles[j].t;
                particles[j].t = particles[i].t;
                particles[j].dt -= dt;
```

```
                                return j;
                        }
                }
                else {
                        dx = p2.x - p1.x - p1.vx*t2;
                        dy = p2.y - p1.y - p1.vy*t2;
                        dz = p2.z - p1.z - p1.vz*t2;

                        bij = dx * dvx + dy * dvy + dz*dvz;
                        if (bij < 0.0) {
                                dv = dvx * dvx + dvy * dvy + dvz*dvz;
                                d = bij * bij + dv * (4.0 - dx * dx - dy * dy - dz * dz);

                                dt = -(sqrt(d) + bij) / dv;

                                if (dt > 0.0) {
                                        particles[i].x += p1.vx*t2;
                                        particles[i].y += p1.vy*t2;
                                        particles[i].z += p1.vz*t2;

                                        particles[j].x = p2.x;
                                        particles[j].y = p2.y;
                                        particles[j].z = p2.z;

                                        dt = particles[i].t - particles[j].t;
                                        particles[j].t = particles[i].t;
                                        particles[j].dt -= dt;

                                        return j;
                                }
                        }
                }
        }
}

// EN: if we can't find collission we will return -1 and it will mean
```

```
        // than we should try to search collissions with other value of X for this virtual particle.
        // RU: если мы не нашли соударения для этого образа, то надо изменить X
        // и попробовать искать соударения снова.
        return -1;

}

/*
   Функция позволяет найти свободное место для нового образа
   по оси X. Если место не найдено, функция сталкивает новый образ
   с одной из частиц, мешающих его поставить.

   Аргументы:
    i - номер нового образа
*/
void find_place_for_particle(int &i) {

        double x, dy, dz, dx, d, dt, dvx, dvy, dvz, bij, dv, x_min, x_max = 0.0;
        double no_free_space_min[300];  // массивы для сохраненения данных о промежутках,
        double no_free_space_max[300];  // занятых другими частицами

        int particle_on_the_line = -1;      // номер частицы, находящейся вдоль OX, с которой можно
                                            // столкнуть образ
        double particle_x_for_collission;   // координата X в которую надо поставить образ перед
                                            // столкновением

        // данные по месту, занятому другими частицами в вдоль оси OX, которые мешают
        // вставить новый образ в систему
        bool include;
        int spaces = 2;
        no_free_space_min[1] = -L;
        no_free_space_max[1] = 1.0 - L + 1.0e-6;
        no_free_space_min[2] = L - 1.0 - 1.0e-6;
        no_free_space_max[2] = L;

        // BUG: потенциальная проблема перебирать частицы начиная с 0 до конца системы
        // т.к. мы остановим цикл при обнаружении первой подходящей частицы, что увеличивает
```

```
// вероятность того что такая частица будет найдена вблизи "левой" идеальной стенки.
// необходимо составлять список всех подходящих частиц и случайно выбирать одну из них.
for (int x_box = 0; x_box <= K2; ++x_box)
        for (int y_box = particles[i].y_box - 1; y_box < particles[i].y_box + 2; ++y_box) {
                for (int z_box = particles[i].z_box - 1; z_box < particles[i].z_box + 2; ++z_box) {

                        if (y_box == -1) {
                                continue;
                        }
                        if (y_box == K + 1) {
                                continue;
                        }
                        if (z_box == -1) {
                                continue;
                        }
                        if (z_box == K + 1) {
                                continue;
                        }

                        for (int s = 0; s <= boxes_yz[y_box][z_box][x_box].end; ++s) {

                                int n = boxes_yz[y_box][z_box][x_box].particles[s];

                                /* игнорируем столкновение образа с самим собой */
                                if (n == i) continue;

                                include = false;

                                // calculate delta t between two particles.
                                // If temp < 1.0e-14 we shouldn't use this particle for collission
                                // because other virtual particle can use the same particle.
                                double temp = particles[i].t - particles[n].t;

                                /*
                                    Синхронизируем частицы с образом по времени и оцениваем близость
                                    частиц с образом по Y и Z, если расстояние между центрами меньше 2R,
                                    то возможно столкновение образа с данной частицей, тогда рассчитываем
```

```
   время столкновения и если оно больше нуля то это и есть искомое столкновение
*/
dy = particles[n].y + particles[n].vy * temp - particles[i].y;
dz = particles[n].z + particles[n].vz * temp - particles[i].z;
d = 4.0 - dy*dy - dz*dz;
if (d >= 0) {
        x_min = particles[n].x + particles[n].vx * temp - sqrt(d);
        x_max = particles[n].x + particles[n].vx * temp + sqrt(d);

        // search particle which can be used for collision
        if (((particle_on_the_line == -1) || (particle_on_the_line > NP)) &&
                (particles[i].vx * particles[n].vx < 0) && (temp > 1.0e-14)) {

                if (particles[i].vx < 0.0) x = x_max;
                else x = x_min;

                dx = particles[n].x + particles[n].vx * temp - x;
                dvx = particles[n].vx - particles[i].vx;
                dvy = particles[n].vy - particles[i].vy;
                dvz = particles[n].vz - particles[i].vz;

                bij = dx * dvx + dy * dvy + dz*dvz;
                if (bij < 0.0) {
                        dv = dvx * dvx + dvy * dvy + dvz*dvz;
                        d = bij * bij + dv * (4.0 - dx * dx - dy * dy - dz * dz);

                        dt = -(sqrt(d) + bij) / dv;

                        if ((dt > 0.0) && (abs(x) < L - 1.0)) {
                                particle_on_the_line = n;
                                particle_x_for_collission = x;
                        }
                }
        }

        /*
           Если частица находится вблизи линии OX, вдоль которой мы ищем свободное
```

```
                                        место для вставки нового образа, то проверяем, в какую ячейку "занятого"
                                        места добавить область, которую блокирует данная частица.
                                    */
                                    for (int m = 1; m <= spaces; ++m) {
                                            if (x_min <= no_free_space_min[m] &&
                                                    x_max <= no_free_space_max[m] &&
                                                    x_max >= no_free_space_min[m])
                                            {
                                                    no_free_space_min[m] = x_min;
                                                    include = true;
                                            }
                                            if (x_min <= no_free_space_min[m] &&
                                                    x_max >= no_free_space_max[m])
                                            {
                                                    no_free_space_min[m] = x_min;
                                                    no_free_space_max[m] = x_max;
                                                    include = true;
                                            }
                                            if (x_min <= no_free_space_max[m] &&
                                                    x_min >= no_free_space_min[m] &&
                                                    x_max >= no_free_space_max[m])
                                            {
                                                    no_free_space_max[m] = x_max;
                                                    include = true;
                                            }
                                    }
                                    if (include == false) {
                                            spaces += 1;
                                            no_free_space_min[spaces] = x_min;
                                            no_free_space_max[spaces] = x_max;
                                    }
                                }
                            }
                        }
                    }

                // сортируем массив данных о занятых областях в системе,
```

```
// куда мы уже не можем вставить образ, т.к. он будет пересекаться
// с другими частицами и образами
for (int r1 = 1; r1 < spaces; r1++)
        for (int r2 = r1 + 1; r2 <= spaces; r2++)
                if (no_free_space_min[r1] > no_free_space_min[r2])
                {
                        double temp = no_free_space_min[r1];
                        no_free_space_min[r1] = no_free_space_min[r2];
                        no_free_space_min[r2] = temp;

                        temp = no_free_space_max[r1];
                        no_free_space_max[r1] = no_free_space_max[r2];
                        no_free_space_max[r2] = temp;
                }

/*
   Необходимо объединить области "занятого пространства" если они
   пересекаются между собой.
*/
int r1 = 1;
while (r1 < spaces)
{
        if (no_free_space_max[r1] > no_free_space_min[r1 + 1])
        {
                if (no_free_space_max[r1] < no_free_space_max[r1 + 1])
                        no_free_space_max[r1] = no_free_space_max[r1 + 1];
                for (int r2 = r1 + 1; r2 < spaces; r2++)
                {
                        no_free_space_min[r2] = no_free_space_min[r2 + 1];
                        no_free_space_max[r2] = no_free_space_max[r2 + 1];
                }
                spaces--;
        }
        else r1++;
}

if (spaces < 2) {     // Если свободное место для нового образа не найдено, то:
```

```
/*
   Проверяем, можем ли мы столкнуть новый образ с какой-то частицей,
   которая мешала его вставить в систему
*/
if (particle_on_the_line > -1) {

        particles[i].x = particle_x_for_collission;

        particle p = particles[particle_on_the_line];

        // Синхронизируем частицы по времени
        dt = particles[i].t - p.t;
        p.x += p.vx * dt;
        p.y += p.vy * dt;
        p.z += p.vz * dt;
        p.t = particles[i].t;
        p.dt -= dt;
        particles[particle_on_the_line] = p;

        return;
}
else {  // если такой частицы нет, то ищем другие варианты
        int r = -1, u = 0;

        r = search_collission_for_new_virtual_particle(i);

        // пробуем найти соударения для нового образа в системе тысячу раз
        // с разной случайным образом выбираемой координатой X
        while (r == -1 && u < 1000)
        {
                // выбираем случайную позицию по X для образа в системе
                // так, чтобы образ находился не слишком близко к идеальным стенкам
                double position_step = 2.0L * (L - 1.1) / double(RAND_MAX);
                particles[i].x = 1.1L + double(rand()) * position_step - L;

                /*
```

```
                    Ищем соударения с другими частицами при новой координате
                    X для образа, т.е. поиск столкновения происходит по плоскости,
                    в которой находятсялиния скорости образа и линия
                    Y = const, Z = const, X in [1.1; L-1.1];
                */
                r = search_collission_for_new_virtual_particle(i);

                u++;
            }

            /*
                Аварийно останавливаем программу если не смогли найти места для нового образа
                а так же не смогли найти частицу, с которой данный образ может столкнуться.
            */
            if (r == -1) {
                int parent_particle = i - NP;
                particle p1 = particles[i];
                particle p2 = particles[parent_particle];

                printf("\n A = %.15le, L = %.15le, u = %d \n", A, L, u);

                printf("\n %d particle: x = %.15le, y = %.15le, z = %.15le \n", i, p1.x, p1.y, p1.z);
                printf("\n %d particle: vx = %.15le, vy = %.15le, vz = %.15le \n", i, p1.vx, p1.vy, p1.vz);
                printf("\n x_box = %d, y_box = %d, z_box = %d \n", p1.x_box, p1.y_box, p1.z_box);

                printf("\n %d particle: x = %.15le, y = %.15le, z = %.15le \n", parent_particle, p2.x, p2.y,
                printf("\n %d particle: vx = %.15le, vy = %.15le, vz = %.15le \n", parent_particle, p2.vx, p
                printf("\n x_box = %d, y_box = %d, z_box = %d \n", p2.x_box, p2.y_box, p2.z_box);

                throw "ERROR!!";
            }
        }
    }

    x_max = -L;
    x_min = L;
    double length = 2 * L, r;
```

```
        // Выбираем наименьшее свободное пространство вдоль оси OX
        for (int m = 1; m < spaces; ++m) {
                // free space distance:
                r = abs(no_free_space_max[m] - no_free_space_min[m + 1]);
                if (r < length) {
                        length = r;
                        x_min = no_free_space_max[m];
                        x_max = no_free_space_min[m + 1];
                }
        }

        // вставляем новый образ в середину свободного пространства
        particles[i].x = x_min + (x_max - x_min) / 2.0;
}


/*
   Функция удаления "образа" частицы из системы

   Аргументы:
    i - номер частицы, к которой принадлежит образ
*/
void destroy_virt_particle(int &i) {

        if (i >= NP) {
                printf("\n %d particle \n", i);
                printf(" i_copy = %d    particle icopy = %d \n",
                                particles[i].i_copy, particles[i - NP].i_copy);
                printf(" x, y, z: %.5le %.5le %.5le",
                                particles[i].x, particles[i].y, particles[i].z);

                throw "Error: can't destroy the real particle!";
        }
        if (particles[i].i_copy == -1) return;

        int new_i = i + NP;
```

```
        int x_box, y_box, z_box, box_i;

        x_box = particles[new_i].x_box;
        y_box = particles[new_i].y_box;
        z_box = particles[new_i].z_box;
        box_i = particles[new_i].box_i;
        Box p_box = boxes_yz[y_box][z_box][x_box];

        // Очищаем информацию об этом образе из списка частиц в ячейке системы
        if (p_box.particles[box_i] == new_i)
        {
                int j = p_box.particles[box_i] = p_box.particles[p_box.end];
                particles[j].box_i = box_i;
                --p_box.end;
                boxes_yz[y_box][z_box][x_box] = p_box;
        }

        particles[new_i].t = particles[i].t;
        clear_particle_events(new_i);  // очищаем события, связанные с этим образом

        particles[i].i_copy = -1;
        particles[new_i].i_copy = -1;
}


/*
   Функция обмена частицы и "образа" при пересечении частицей периодических
   граничных условий.

   Аргументы:
    im - номер частицы, пересекающей периодические граничные условия
        jm - номер границы, через которую проходит центр частицы
*/
void change_with_virt_particles(int &im, int &jm) {
        int y_box, z_box;
        int f = im + NP;  // рассчитываем номер образа данной частицы
```

```
double dt = particles[im].t - particles[f].t;
particles[im].x = particles[f].x + dt*particles[f].vx;
particles[im].y = particles[f].y + dt*particles[f].vy;
particles[im].z = particles[f].z + dt*particles[f].vz;

// записываем её в новую ячейку
y_box = particles[f].y_box;
z_box = particles[f].z_box;

if (y_box <= 0) y_box = 1;
if (y_box >= K) y_box = K - 1;
if (z_box <= 0) z_box = 1;
if (z_box >= K) z_box = K - 1;

particles[im].x_box = particles[f].x_box;
particles[im].y_box = y_box;
particles[im].z_box = z_box;

/*
   При обмене виртуального образа на частицу ставим частицу точно на границу ячейки,
   чтобы избегать накопления ошибок.
*/
if (particles[im].y > A) particles[im].y = boxes_yz[y_box][z_box][particles[f].x_box].y2;
else if (particles[im].y < -A) particles[im].y = boxes_yz[y_box][z_box][particles[f].x_box].y1;
if (particles[im].z > A) particles[im].z = boxes_yz[y_box][z_box][particles[f].x_box].z2;
else if (particles[im].z < -A) particles[im].z = boxes_yz[y_box][z_box][particles[f].x_box].z1;

/*
   Проверяем что мы разместили частицу в правильной ячейке
*/
Box b = boxes_yz[y_box][z_box][particles[f].x_box];
if (((particles[im].x < b.x1) && (abs(particles[im].x - b.x1) > 1.0e-14)) ||
        ((particles[im].x > b.x2) && (abs(particles[im].x - b.x2) > 1.0e-14)) ||
        ((particles[im].y < b.y1) && (abs(particles[im].y - b.y1) > 1.0e-14)) ||
        ((particles[im].y > b.y2) && (abs(particles[im].y - b.y2) > 1.0e-14)) ||
        ((particles[im].z < b.z1) && (abs(particles[im].z - b.z1) > 1.0e-14)) ||
        ((particles[im].z > b.z2) && (abs(particles[im].z - b.z2) > 1.0e-14))) {
```

```
                printf("\n particle %d i_copy %d x = %.15le , y = %.15le , z = %.15le \n",
                        im, particles[im].i_copy, particles[im].x, particles[im].y, particles[im].z);
                printf(" particle %d x = %.15le , y = %.15le , z = %.15le \n",
                        f, particles[f].x, particles[f].y, particles[f].z);
                printf("box x [%.15le; %.15le] \n", b.x1, b.x2);
                printf("box y [%.15le; %.15le] \n", b.y1, b.y2);
                printf("box z [%.15le; %.15le] \n", b.z1, b.z2);

                printf("\n\n particle %d t = %.15le , particle %d t = %.15le \n",
                        im, particles[im].t, f, particles[f].t);
                printf(" particle %d t+dt = %.15le , particle %d t+dt = %.15le \n",
                        im, particles[im].t + particles[im].dt, f, particles[f].t + particles[f].dt);
                printf(" particle %d event: %d %d %.15le \n", f, time_queue[particles[f].ti].im,
                        time_queue[particles[f].ti].jm, time_queue[particles[f].ti].t);

                throw "stop";
        }

        // удаляем образ частицы
        destroy_virt_particle(im);
}


/*
   Функция создания нового "образа".

   Аргументы:
    i - номер частицы, образ которой необходимо создать
        need_to_check - флаг, позволяющий не проверять нужен ли образ для данной
                        частицы или нет, и сразу создавать образ (когда мы уверены,
                                что образ необходимо создать).
*/
void create_virt_particle(int &i, bool need_to_check=true) {
        double dt, dt_min, t_min, y, z, dy, dz;
        double kv = 1.0;
        double t01, t02;
        short x_box, y_box, z_box;
```

```
int new_i = i + NP;

destroy_virt_particle(i);

if ((need_to_check == false) ||
        (((particles[i].y <= 1.0 - A) && (particles[i].vy < 0.0)) ||
         ((particles[i].y >= A - 1.0) && (particles[i].vy > 0.0)) ||
         ((particles[i].z <= 1.0 - A) && (particles[i].vz < 0.0)) ||
         ((particles[i].z >= A - 1.0) && (particles[i].vz > 0.0)))) {

        dt_min = 1.0e+20;

        y = A + particles[i].y;  // расстояние от частицы до стенок системы,
        z = A + particles[i].z;  // от которых частица удаляется

        dy = A + particles[i].y; // расстояние от частицы до стенок системы,
        dz = A + particles[i].z; // к которым частица приближается

        if (particles[i].vy < 0)
                y = A - particles[i].y;
        if (particles[i].vz < 0)
                z = A - particles[i].z;

        if (particles[i].z > 0)
                dz = A - particles[i].z;
        if (particles[i].y > 0)
                dy = A - particles[i].y;

        // если частица стремится покинуть систему и она находится за границей
        // области [1.0; A - 1.0] то рассчитать координаты образа
        if ((particles[i].y*particles[i].vy > 0) && (dy - 1.0 < 1.0e-14)) {
                // время, через которое частица достигнет периодической границы
                dt = dy / abs(particles[i].vy);

                /*
                   Присваиваем "образу" ту же скорость
                   что у исходной частицы, чтобы он двигался
```

```
    вдоль той же линии скорости, что и частица
*/
particles[new_i].vx = particles[i].vx;
particles[new_i].vy = particles[i].vy;
particles[new_i].vz = particles[i].vz;

/*
    Времена, за которые частица достигнет периодических границ
    если её скорость будет направлена в противоположную сторону
*/
t01 = y / abs(particles[i].vy);
t02 = z / abs(particles[i].vz);

/*
     Нам необходимо вычислить времена t01 и t02 для того чтобы знать
          на какой из периодических границ необходимо создать новый образ.
          Для этого мы проводим линию вдоль линии скорости частицы и находим
          точку ближайшего пересечения периодических границ в направлении, обратном
          направлению движения частицы, покидающей объём.

          Выяснив, какую из периодических границ данная линия пересекает первой,
          мы можем рассчитать точные координаты нахождения образа, считая, что он должен
          войти в систему в точке пересечения линии, проведенной вдоль линии скорости
          и периодической границы, при этом образ должен достигнуть этой точки в тот же
          момент, когда его частица пересечет другую периодическую границу (через dt)
*/

if (t01 < t02) {
        t_min = t01;

        particles[new_i].x = particles[i].x - particles[i].vx*t01 - particles[i].vx*dt;
        particles[new_i].y = particles[i].y - particles[i].vy*t01 - particles[i].vy*dt;
        particles[new_i].z = particles[i].z - particles[i].vz*t01 - particles[i].vz*dt;
}
else {
        t_min = t02;
```

```
                particles[new_i].x = particles[i].x - particles[i].vx*t02 - particles[new_i].vx*dt;
                particles[new_i].y = particles[i].y - particles[i].vy*t02 - particles[new_i].vy*dt;
                particles[new_i].z = particles[i].z - particles[i].vz*t02 - particles[new_i].vz*dt;
        }

        dt_min = dt;
}
if ((particles[i].z*particles[i].vz > 0) && (dz - 1.0 < 1.0e-14)) {
        // время, через которое частица достигнет периодической границы
        dt = dz / abs(particles[i].vz);

        if (dt < dt_min) {

                particles[new_i].vx = particles[i].vx;
                particles[new_i].vy = particles[i].vy;
                particles[new_i].vz = particles[i].vz;

                /*
                   Времена, за которые частица достигнет периодических границ
                   если её скорость будет направлена в противоположную сторону
                */
                t01 = y / abs(particles[i].vy);
                t02 = z / abs(particles[i].vz);

                if (t01 < t02) {
                        t_min = t01;

                        particles[new_i].x = particles[i].x - particles[i].vx*t01 - particles[i].vx*dt;
                        particles[new_i].y = particles[i].y - particles[i].vy*t01 - particles[i].vy*dt;
                        particles[new_i].z = particles[i].z - particles[i].vz*t01 - particles[i].vz*dt;
                }
                else {
                        t_min = t02;

                        particles[new_i].x = particles[i].x - particles[i].vx*t02 - particles[new_i].vx*dt;
                        particles[new_i].y = particles[i].y - particles[i].vy*t02 - particles[new_i].vy*dt;
                        particles[new_i].z = particles[i].z - particles[i].vz*t02 - particles[new_i].vz*dt;
```

```
                }

                dt_min = dt;
        }
}

/*
   В случае если координата X предполагаемого места вставки образа находится
   вне системы нам необходимо нормировать координату, отразив линию скорости
   столько раз, сколкьо потребуется чтобы координата X находилась в системе.
   При каждом отражении нам необходимо учитывать смену знака скорости образа
*/
if (abs(particles[new_i].x) > L - 1.0) {
        int f = 1;
        double LL = L - 1.0;
        double x = particles[new_i].x;

        if (particles[new_i].x < -LL) f = -1;
        x = x - f*LL;
        int m = int(x / (2.0 * LL));
        x = x - 2.0 * m * LL;

        /*
           Если число отражений линии скорости от стенок чётное,
           то скорость частицы необходимо изменить на противоположную
           (т.к. мы уже один раз "отразили" линию скорости, когда выполнили
           x = x - f*LL несколькими строками выше).
        */
        int x_wall = 1;
        if (m % 2 != 0)
                x_wall = -1;
        else
                particles[new_i].vx = -particles[new_i].vx;
        particles[new_i].x = x_wall * (f * LL - x);
}

/*
```

```
    Определяем ячейку в которую будет записан новый "образ"
*/
x_box = short((L + dL + particles[new_i].x) / dL);
y_box = short((A + dA + particles[new_i].y) / dA);
z_box = short((A + dA + particles[new_i].z) / dA);

/*
    Если новые координаты ячейки выходят за пределы системы,
    то поместить "образ" в граничную ячейку (например, если
    координата образа по Y больше, чем A+1 - такое возможно,
    если частица имеет большую скорость).
*/
if (y_box < 0) y_box = 0;
if (z_box < 0) z_box = 0;
if (y_box > K) y_box = K;
if (z_box > K) z_box = K;

// корректируем индексы ячейки, в которую попадает частица, т.к. при делении double
// мы имеем погрешность в 1.0e-13, из за чего ячейка может быть определена неверно
if ((boxes_yz[y_box][z_box][x_box].x1 > particles[new_i].x) && (x_box > 0)) x_box--;
if ((boxes_yz[y_box][z_box][x_box].x2 < particles[new_i].x) && (x_box < K2)) x_box++;
if ((boxes_yz[y_box][z_box][x_box].y1 > particles[new_i].y) && (y_box > 0)) y_box--;
if ((boxes_yz[y_box][z_box][x_box].y2 < particles[new_i].y) && (y_box < K)) y_box++;
if ((boxes_yz[y_box][z_box][x_box].z1 > particles[new_i].z) && (z_box > 0)) z_box--;
if ((boxes_yz[y_box][z_box][x_box].z2 < particles[new_i].z) && (z_box < K)) z_box++;

particles[new_i].x_box = x_box;
particles[new_i].y_box = y_box;
particles[new_i].z_box = z_box;

/* синхронизируем время образа со временем частицы */
particles[new_i].t = particles[i].t;

// проверка того что новый образ не пересекается с уже существующими частицами
bool search = false;
for (short r = x_box - 1; r < x_box + 2; ++r) {
        for (short q = y_box - 1; q < y_box + 2; ++q) {
```

```
                for (short w = z_box - 1; w < z_box + 2; ++w) {

                        if (q == -1) {
                                continue;
                        }
                        if (q == K + 1) {
                                continue;
                        }
                        if (w == -1) {
                                continue;
                        }
                        if (w == K + 1) {
                                continue;
                        }

                        /*
                           проверяем проникновение нового образа с частицами в ячейках,
                           ближайших к ячейке, куда добавляется образ.
                           Если образ проникает в какую-то частицу или другой образ,
                           то необходимо искать новое место для образа.
                        */
                        for (int s = 0; s <= boxes_yz[q][w][r].end; ++s) {
                                int n = boxes_yz[q][w][r].particles[s];
                                double temp = particles[new_i].t - particles[n].t;
                                double dx = particles[n].x + particles[n].vx * temp - particles[new_i].x;
                                double dy = particles[n].y + particles[n].vy * temp - particles[new_i].y;
                                double dz = particles[n].z + particles[n].vz * temp - particles[new_i].z;

                                if (4.0 > dx*dx + dy*dy + dz*dz) {
                                        search = true;
                                }
                        }
                }
        }
}

// если новый образ пересекается с уже существующими частицами или образами,
```

```
// необходимо найти другое место для нового образа
if (search == true) {

        find_place_for_particle(new_i);

        /*
           обновляем информацию о ячейке, в которой находится образ,
           так как после поиска нового положения для образа номер ячейки
           мог измениться.
        */
        x_box = short((L + dL + particles[new_i].x) / dL);
        y_box = short((A + dA + particles[new_i].y) / dA);
        z_box = short((A + dA + particles[new_i].z) / dA);

        /*
           Если новые координаты ячейки выходят за пределы системы,
           то поместить "образ" в граничную ячейку (например, если
           координата образа по Y больше, чем A+1 - такое возможно,
           если частица имеет большую скорость).
        */
        if (y_box < 0) y_box = 0;
        if (z_box < 0) z_box = 0;
        if (y_box > K) y_box = K;
        if (z_box > K) z_box = K;

        // корректируем индексы ячейки, в которую попадает частица, т.к. при делении double
        // мы имеем погрешность в 1.0e-13, из за чего ячейка может быть определена неверно
        if ((boxes_yz[y_box][z_box][x_box].x1 > particles[new_i].x) && (x_box > 0)) x_box--;
        if ((boxes_yz[y_box][z_box][x_box].x2 < particles[new_i].x) && (x_box < K2)) x_box++;
        if ((boxes_yz[y_box][z_box][x_box].y1 > particles[new_i].y) && (y_box > 0)) y_box--;
        if ((boxes_yz[y_box][z_box][x_box].y2 < particles[new_i].y) && (y_box < K)) y_box++;
        if ((boxes_yz[y_box][z_box][x_box].z1 > particles[new_i].z) && (z_box > 0)) z_box--;
        if ((boxes_yz[y_box][z_box][x_box].z2 < particles[new_i].z) && (z_box < K)) z_box++;

        particles[new_i].x_box = x_box;
        particles[new_i].y_box = y_box;
        particles[new_i].z_box = z_box;
```

```
                }

                /*
                    После того как подходящее место найдено мы записываем новый образ
                    в одну из ячеек системы, добавляя её номер в список частиц в данной ячейке
                    и увеличивая счётчик количества частиц в этом списке на 1.
                */
                short end = particles[new_i].box_i = ++boxes_yz[y_box][z_box][x_box].end;
                boxes_yz[y_box][z_box][x_box].particles[end] = new_i;

                /*
                    Определяем i_copy для частицы и её "образа", так, чтобы
                    они указывали друг на друга.
                */
                particles[new_i].i_copy = i;
                particles[i].i_copy = new_i;
            }
        }
```



```
        /*
            Функция загрузки информации о некоторой частице из файла.
            При загрузке системы мы последовательно считываем информацию
            о каждой частице с помощью этой функции.

            Аргументы:
              loading_file – ссылка на файл, из которого необходимо считать
                               данные о частице или образе.
        */
        void load_information_about_one_particle(FILE *loading_file) {
            double a1, a2, a3;
            int i, i_copy, x_box, y_box, z_box, end;

            // Считывем номер частицы и номер образа
            fscanf(loading_file, "%d %d\n", &i, &i_copy);
            particles[i].i_copy = i_copy;
```

```
// Считываем координаты частицы
fscanf(loading_file, "%le %le %le\n", &a1, &a2, &a3);
particles[i].x = a1 - L;
particles[i].y = a2 - A;
particles[i].z = a3 - A;

// Определяем в какую ячейку необходимо записать новую частицу
x_box = short((a1 + dL) / dL);
y_box = short((a2 + dA) / dA);
z_box = short((a3 + dA) / dA);

if (y_box < 0) y_box = 0;
if (z_box < 0) z_box = 0;
if (y_box > K) y_box = K;
if (z_box > K) z_box = K;

// корректируем индексы ячейки, в которую попадает частица, т.к. при делении double
// мы имеем погрешность в 1.0e-13, из за чего ячейка может быть определена неверно
if ((boxes_yz[y_box][z_box][x_box].x1 > particles[i].x) && (x_box > 0)) x_box--;
if ((boxes_yz[y_box][z_box][x_box].x2 < particles[i].x) && (x_box < K2)) x_box++;
if ((boxes_yz[y_box][z_box][x_box].y1 > particles[i].y) && (y_box > 0)) y_box--;
if ((boxes_yz[y_box][z_box][x_box].y2 < particles[i].y) && (y_box < K)) y_box++;
if ((boxes_yz[y_box][z_box][x_box].z1 > particles[i].z) && (z_box > 0)) z_box--;
if ((boxes_yz[y_box][z_box][x_box].z2 < particles[i].z) && (z_box < K)) z_box++;

particles[i].x_box = x_box;
particles[i].y_box = y_box;
particles[i].z_box = z_box;

/*
   Если мы загружаем данные о частице (и это не "образ") то
   проверяем что частица находится в правильной ячейке.
*/
if (i < NP) {
        if (boxes_yz[y_box][z_box][x_box].x1 <= particles[i].x &&
                boxes_yz[y_box][z_box][x_box].x2 >= particles[i].x)
                particles[i].x_box = x_box;
```

```
            else {
                    printf("Particle␣locates␣in␣incorrect␣place␣%d", i);
                    printf("\n␣a1␣=␣%.15le␣,␣x␣=␣%.15le␣,␣L␣=␣%.15le␣,␣dL␣=␣%.15le␣\n",
                            a1, particles[i].x, L, dL);
                    printf("\n␣x_box␣=%d,␣BOX␣x:␣[%.15le;␣%.15le]", x_box,
                            boxes_yz[y_box][z_box][x_box].x1, boxes_yz[y_box][z_box][x_box].x2);
                    printf("\n␣particle␣x:␣%.15le", particles[i].x);
                    throw "Particle␣locates␣in␣incorrect␣place";
            }

            if (boxes_yz[y_box][z_box][x_box].y1 <= particles[i].y &&
                    boxes_yz[y_box][z_box][x_box].y2 >= particles[i].y)
                    particles[i].y_box = y_box;
            else {
                    printf("Particle␣locates␣in␣incorrect␣place␣%d", i);
                    printf("\n␣BOX␣y:␣[%.15le;␣%.15le]",
                            boxes_yz[y_box][z_box][x_box].y1, boxes_yz[y_box][z_box][x_box].y2);
                    printf("\n␣particle␣y:␣%.15le", particles[i].y);
                    throw "Particle␣locates␣in␣incorrect␣place";
            }

            if (boxes_yz[y_box][z_box][x_box].z1 <= particles[i].z &&
                    boxes_yz[y_box][z_box][x_box].z2 >= particles[i].z)
                    particles[i].z_box = z_box;
            else {
                    printf("\n␣Particle␣%d␣locates␣in␣incorrect␣place", i);
                    printf("\n␣BOX␣z:␣[%.15le;␣%.15le]",
                            boxes_yz[y_box][z_box][x_box].z1, boxes_yz[y_box][z_box][x_box].z2);
                    printf("\n␣particle␣Z:␣%.15le", particles[i].z);
                    throw "Particle␣locates␣in␣incorrect␣place";
            }
    }

    // Считываем информацию о скоростях частицы
    fscanf(loading_file, "%le␣%le␣%le\n", &a1, &a2, &a3);
    particles[i].vx = a1;
    particles[i].vy = a2;
```

```
particles[i].vz = a3;

particles[i].t = 0.0;
particles[i].dt = 0.0;
particles[i].ti = 1;

// записываем частицу в ячейку
end = particles[i].box_i = ++boxes_yz[y_box][z_box][x_box].end;
boxes_yz[y_box][z_box][x_box].particles[particles[i].box_i] = i;

global_E += a1*a1 + a2*a2 + a3*a3;

/*
   Если в одной ячейке находится слишком много частиц,
   то завершить программу, выведя сообщение об ошибке.
*/
if (end > 7) {
        printf("Error: Too many particles in one box");
        printf("\n x_box, y_box, z_box = %d %d %d , end = %d \n", x_box, y_box, z_box, end);
        throw "Error: Too many particles in one box";
}

/*
   Проверяем что в данной ячейке нет повторяющихся номеров частиц
*/
Box b = boxes_yz[particles[i].y_box][particles[i].z_box][particles[i].x_box];
for (short t = 0; t < particles[i].box_i; ++t)
        for (short d = t + 1; d <= particles[i].box_i; ++d) {
                if (b.particles[t] == b.particles[d])
                {
                        printf("\n Duplicated particles in one box: %d \n", i);

                        for (int j = 0; j < b.end; ++j)
                                printf("%d ", b.particles[j]);

                        throw "Duplicated particles in one box";
                }
```

```
                }
}


/*
    Функция загрузки данных о системе из файла.
    Загружаются параметры объёма, координаты и скорости всех частиц.

    Аргументы:
     file_name - имя файла, содержащего все данные о системе.
*/
void load_seed(std::string file_name) {
        double a1, x, y, z;
        int i, count_of_virt_particles;

        FILE *loading_file;
        /*
            Открываем указанный файл для чтения,
            пишем на экран информацию об ошибке и выходим из программы если
            указанного файла не существует или его невозможно открыть для чтения.
        */
        try {
                loading_file = fopen(file_name.c_str(), "r");
        }
        catch (...) {
                printf("\n ERROR: Can't read file '%s' - is it exist? \n", file_name.c_str());
                exit(1);
        }

        // считываем количество частиц в системе
        fscanf(loading_file, "%i\n", &i);
        NP = i;
        // считываем количество образов в системе
        fscanf(loading_file, "%i\n", &i);
        count_of_virt_particles = i;
        // считываем значение A
        fscanf(loading_file, "%le\n", &a1);
```

```
A = a1 / 2.0;
A2 = a1;
// считываем значение L
fscanf(loading_file, "%le\n", &a1);
L = a1 / 2.0;

// EN: search for the appropriate values for dA, dL, K, K2
// RU: ищем подходящее значение для dA, dL, K, K2
dA = 2.5;
dL = 2.5;
K = short(A2 / dA) + 1;          // количество ячеек по Y и Z
K2 = short(2.0 * L / dL) + 1;    // количество ячеек по X
dA = A2 / (K - 1);               // точные размеры ячеек по X, Y и Z
dL = 2.0*L / (K2 - 1);           // выбранные так, чтобы, ячейки совпадали
                                 // с размерами объёма

/*
  После того как мы рассчитали количество ячеек и их размеры
  необходимо инициализировать данные каждой ячейки, указать
  границы каждой ячейки по X, Y, Z, т.к. эти данные будут
  использоваться для рассчёта времени до пересечения границ
  ячеек частицами, которые находятся в этих ячейках.
*/
y = z = -A - dA;
x = -L - dL;
for (int i = 0; i <= K; i++) {
        for (int j = 0; j <= K; j++) {
                for (int w = 0; w <= K2; w++) {
                        boxes_yz[i][j][w].x1 = x;
                        boxes_yz[i][j][w].x2 = x + dL;
                        boxes_yz[i][j][w].y1 = y;
                        boxes_yz[i][j][w].y2 = y + dA;
                        boxes_yz[i][j][w].z1 = z;
                        boxes_yz[i][j][w].z2 = z + dA;
                        boxes_yz[i][j][w].end = -1;

                        x += dL;
```

```
                }
                x = -L - dL;
                z += dA;
            }
            y += dA;
            z = -A - dA;
    }

    /*
        Обнуляем все данные для всех частиц и образов перед считыванием
        файла с данными.
        Для образов, которые не описаны в файле сохранения, останутся эти
        начальные значения.
    */
    for (int i = 0; i < NP * 2; i++) {
            particles[i].x_box = 0;
            particles[i].y_box = 0;
            particles[i].z_box = 0;
            particles[i].box_i = 0;
            particles[i].i_copy = -1;
            particles[i].ti = 0;
            particles[i].dt = 0.0;
    }

/*
    Загружаем данные о всех частицах и "образах" из файла сохранения
*/
    for (int i = 0; i < NP + count_of_virt_particles; i++) {
            load_information_about_one_particle(loading_file);
    }

    fclose(loading_file);

    /*
        Инициализируем очередь событий пустыми событиями и устанавливаем
        указатель конца списка на его начало.
    */
```

```
        last = 1;
        time_queue[0].t = 0.0;
        time_queue[0].im = -1;
        for (int i = 1; i < 16384; ++i) time_queue[i].t = 1.0E+20;

        /*
           Рассчитываем новые события для всех частиц и существующих в системе "образов"
        */
        for (int i = 0; i < NP; ++i) {
                retime(i);
                if (particles[i].i_copy > 0) retime(particles[i].i_copy);
        }
}


/*
   Функция сохранения состояния системы в текстовый файл.
   перед сохранением производится синхронизация частиц по времени.

   Аргументы:
    file_name - имя файла, в который будет записана информация
*/
void save(std::string file_name) {
        int images[7000];
        int count_of_images = 0;
        double x, y, z, dt;
        double t_global = get_maximum_particle_time();

        // RU: мы должны сохранить информацию об образах частиц чтобы
        // не пересоздавать образы при каждой загрузке системы из сохраненного состояния.
        // EN: we need to save all virtual particles (images) because we
        // don't want to create all virtual particles during the load_seed().
        for (int i = 0; i < NP; ++i) {
                if (particles[i].i_copy >= NP) {
                        images[count_of_images] = particles[i].i_copy;
                        count_of_images += 1;
                }
```

```
}

FILE *save_file = fopen(file_name.c_str(), "w+");

// RU: сохраняем информацию о количестве частиц и размерах системы
// EN: save information about count of particles and size of the system
fprintf(save_file, "%d\n", NP);
fprintf(save_file, "%d\n", count_of_images);
fprintf(save_file, "%.15le\n", A * 2.0);
fprintf(save_file, "%.15le\n", L * 2.0);

// RU: сохраняем координаты всех частиц и их скорости: x, y, z, vx, vy, vz
// EN: we need to save all coordinates of particles: x, y, z, vx, vy, vz
for (int i = 0; i < NP; ++i) {
        fprintf(save_file, "%d %d\n", i, particles[i].i_copy);

        /*
            Синхронизируем частицу i с глобальным временем системы,
            в итоге данные всех частиц будут записаны для одного момента времени,
            который совпадает со временем последнего произошедшего в системе события.
        */
        dt = t_global - particles[i].t;

        /*
            При сохранении состояния системы мы указываем
            абсолютные координаты частиц и параметры системы, так, чтобы
            центр системы был в точке (L/2, A/2), а не в (0, 0).
        */
        x = L + particles[i].x + particles[i].vx * dt;
        y = A + particles[i].y + particles[i].vy * dt;
        z = A + particles[i].z + particles[i].vz * dt;
        fprintf(save_file, "%.15le %.15le %.15le\n", x, y, z);
        fprintf(save_file, "%.15le %.15le %.15le\n", particles[i].vx,
                    particles[i].vy, particles[i].vz);
}

// RU: мы так же сохраняем координаты всех виртуальных частиц (образов)
```

```
        // EN: we also need to save all coordinates of virtual particles (images)
        for (int i = 0; i < count_of_images; ++i) {
                int m = images[i];
                fprintf(save_file, "%d %d\n", m, particles[m].i_copy);

                /*
                   Синхронизируем частицу i с глобальным временем системы,
                   в итоге данные всех частиц будут записаны для одного момента времени,
                   который совпадает со временем последнего произошедшего в системе события.
                */
                dt = t_global - particles[m].t;

                /*
                   При сохранении состояния системы мы указываем
                   абсолютные координаты частиц и параметры системы, так, чтобы
                   центр системы был в точке (L/2, A/2), а не в (0, 0).
                */
                x = L + particles[m].x + particles[m].vx * dt;
                y = A + particles[m].y + particles[m].vy * dt;
                z = A + particles[m].z + particles[m].vz * dt;
                fprintf(save_file, "%.15le %.15le %.15le\n", x, y, z);
                fprintf(save_file, "%.15le %.15le %.15le\n",
                                particles[m].vx, particles[m].vy, particles[m].vz);
        }

        fclose(save_file);
}


/*
   Процедура нового "посева" системы.

   Аргументы:
    NN - число частиц в ребре объёмо центрированного кристалла, на основе
         которого делается изначальный посев
    etta - начальная плотность, которую необходимо задать в системе
*/
```

```
void new_seed(int NN, double etta) {
        int KZ = 2;

        double axy, axz, v0x, v0y, v0z, v1x, v1y, v1z, v2x, v2y, v2z, v3x, v3y, v3z;

        // расстояние между двумя слоями
        double rb = 2.0 * sqrt(2.0);
        double rbp = sqrt(2.0);

        // рассчет параметров начального объема
        double A0 = rb * (NN - 1.0) + 2.0;
        double L0 = rb * (2.0 * NN - 1.0) + 2.0;

        double mL0 = L0;

        double Betta = 0.0;

        double XC, YC, ZC; // координаты центра объема
        int NA; // количество посеянных частиц
        double ax, ay, az, vx, vy, vz;

        NP = 2.0 * NN * (NN * NN + (NN - 1.0) * (NN - 1.0)); //  ожидаемое число частиц
        NP = NP + (2.0 * NN - 1.0) * 2.0 * (NN - 1.0) * NN;  //

        // рассчёт первоначальной плотности
        double etta0 = (4.0 * PI * NP) / (3.0 * A0 * A0 * (L0 - 2.0));

        // Начинаем рассчёт коэффициента расширения системы
        double Alpha = etta0 / etta;
        double sk = L0 / (2.0 * Alpha * (L0 - 2.0));

        /*
           Betta - коэффициент расширения системы из начальной объёмоцентрированной
                   упаковки к частицам, распределённым по всему объёму системы
        */
        Betta = exp((1.0 / 3.0) * log(L0 / (2.0 * Alpha * (L0 - 2.0)) + sk));
        //Betta = Betta + exp( (1.0 / 3.0) * log( -L0 / (2.0 * Alpha * (L0-2.0)) - sk) );
```

```
Betta = 1.0 / Betta;

XC = L0 / 2.0; //
YC = A0 / 2.0; //  вычисление центра объёма.
ZC = A0 / 2.0; //

L0 = L0*KZ;
L = L0 * Betta;  // множитель Betta - это коэффициент расширения
A = A0 * Betta;  // на него умножаются все координаты и параметры объёма

NA = 0;

/*
   "Перемешиваем" генератор случайных чисел, чтобы при каждом запуске приложения
   получаемые псевдослучайные числа были новыми
*/
srand(time(NULL));

for (int i = 0; i < 2 * NN; i++)
        for (int j = 0; j < NN; j++)
                for (int k = 0; k < NN / 2; k++) {
                        /*
                             Задаем скорость случайным образом, она будет одинакова
                                 для нескольких частиц сразу(с небольшим отклонением),
                                 чтобы был равен нулю импульс и момент импульса системы.
                        */
                        vx = double(double(rand() % 1000 + 1) / (1000.0 + double(rand() % 100))
                                        - double(rand() % 1000 + 1) / (1000.0 + double(rand() % 100)));
                        vy = double(double(rand() % 1000 + 1) / (1000.0 + double(rand() % 100))
                                        - double(rand() % 1000 + 1) / (1000.0 + double(rand() % 100)));
                        vz = double(double(rand() % 1000 + 1) / (1000.0 + double(rand() % 100))
                                        - double(rand() % 1000 + 1) / (1000.0 + double(rand() % 100)));

                        for (int ii = -1; ii < 2; ii += 2) {
                                if ((i == 0) && (ii > 0)) continue;
```

```
ax = XC + i * ii*rbp + 1.0e-7;

for (int jj = -1; jj < 2; jj += 2) {
        if ((j == 0) && (jj > 0)) continue;

        ay = YC + j * jj*rbp + 1.0e-7;

        for (int kk = -1; kk < 2; kk += 2) {
                if (((i % 2 == 0) && (j % 2 == 0)) ||
                        ((i % 2 != 0) && (j % 2 != 0)))
                        az = ZC + kk * (rbp + k * rb) + 1.0e-7;
                else {
                        if ((k == 0) && (kk > 0)) continue;
                        az = ZC + kk * k*rb + 1.0e-7;
                }

                // назначаем координаты для новой молекулы
                particles[NA].x = ax;
                particles[NA].y = ay;
                particles[NA].z = az;

                if ((j == 0) && (fabs(ZC - az) < 0.1E-15L)
                        && ((i / 2) * 2 != i))
                        if (((i + 1) / 4) * 4 != (i + 1)) {
                                axy = vy;
                                axz = vz;
                                if (ii > 0) {
                                        vy = -vy;
                                        vz = -vz;
                                }
                                vy = ii*vy;
                                vz = ii*vz;
                        }
                        else {
                                vy = axy;
                                vz = axz;
                                vy = ii*vy;
```

```
        vz = ii*vz;
    }
    if (i == 0) {
        if ((j == 0) && (kk < 0)) {
            switch (k) {
            case 0:
                v0x = vx;
                v0y = vy;
                v0z = vz;
                break;
            case 1:
                v1x = vx;
                v1y = vy;
                v1z = vz;
                break;
            case 2:
                v2x = vx;
                v2y = vy;
                v2z = vz;
                break;
            case 3:
                v3x = vx;
                v3y = vy;
                v3z = vz;
                break;
            }
        }
        else {
            switch (k) {
            case 0:
                vx = -v0x;
                vy = -v0y;
                vz = -v0z;
                break;
            case 1:
                vx = -v1x;
                vy = -v1y;
```

```
            vz = -v1z;
            break;
        case 2:
            vx = -v2x;
            vy = -v2y;
            vz = -v2z;
            break;
        case 3:
            vx = -v3x;
            vy = -v3y;
            vz = -v3z;
            break;
        }
    }
    if ((k == 0) && ((j / 2) * 2 != j)) {
        switch (j) {
        case 1:
            vx = jj*v0x;
            vy = jj*v0y;
            vz = jj*v0z;
            break;
        case 3:
            vx = jj*v1x;
            vy = jj*v1y;
            vz = jj*v1z;
            break;
        case 5:
            vx = jj*v2x;
            vy = jj*v2y;
            vz = jj*v2z;
            break;
        case 7:
            vx = jj*v3x;
            vy = jj*v3y;
            vz = jj*v3z;
            break;
```

```
                                                                    }
                                                                }
                                                            }

                                                            particles[NA].vx = vx * ii * jj * kk;
                                                            particles[NA].vy = vy * ii * jj * kk;
                                                            particles[NA].vz = vz * ii * jj * kk;

                                                            NA++;
                                                    }
                                                }
                                            }
                                        }

    // копируем систему столько раз, сколько потребуется.
    for (int jj = 1; jj < KZ; jj++) {
            for (int ii = 0; ii < NP; ii++) {
                    particles[ii + jj * NP].x = particles[ii].x + jj*mL0;
                    particles[ii + jj * NP].y = particles[ii].y;
                    particles[ii + jj * NP].z = particles[ii].z;

                    particles[ii + jj * NP].vx = particles[ii].vx;
                    particles[ii + jj * NP].vy = particles[ii].vy;
                    particles[ii + jj * NP].vz = particles[ii].vz;
            }
    }

    NP = NP*KZ;

    /*
       "расширяем" систему до необходимой плотности.
       Множитель Betta - это коэффициент расширения,
       на него умножаются все координаты и параметры объёма.
    */
    for (int ii = 0; ii < NP; ii++) {
            particles[ii].x = particles[ii].x*Betta;
            particles[ii].y = particles[ii].y*Betta;
```

```
        particles[ii].z = particles[ii].z*Betta;
}

// Рассчитываем и выводим момент импульса Lx системы после посева:
double Lx = 0.0;
for (int i = 0; i < NP; i++) {
        Lx += (particles[i].y + A)*particles[i].vz - (particles[i].z + A)*particles[i].vy;
}
printf("\n Lx = %.15le\n", Lx);

/*
A2 = A;
dA = 2.5;
dL = 2.5;
K = short(A / dA) + 1;
K2 = short(L / dL) + 1;
dA = A / (K - 1);
dL = L / (K2 - 1);
*/

A = A / 2.0;
L = L / 2.0;
for (int i = 0; i < NP; i++) {
        particles[i].t = 0.0;
        particles[i].dt = 0.0;
        particles[i].ti = 0;

        particles[i].x -= L;
        particles[i].y -= A;
        particles[i].z -= A;

        particles[i].i_copy = -1;
}

// поправка для задания точного значения начальной плотности
L = ((PI * NP) / etta) / (6.0 * A * A) + 1.0;
```

```
        /*
            Сохраняем и загружаем систему из файла чтобы инициализировать
            все необходимые переменные.
        */
        save("new.txt");
        load_seed("new.txt");
}


/*
    Функция изменения состояния частиц в соответствии с наступившим
    в системе событием.

    Аргументы:
     im - номер частицы
     jm - номер второй частицы или номер границы
*/
bool reform(int &im, int &jm) {
        particle p1 = particles[im];
        double q1, q2, z, dx, dy, dz;
        bool need_create_virt_particle = false;

        /*
            Если jm >= 0 то наступившее событие - это соударение двух
            частиц или образов.
        */
        if (jm >= 0) {
                particle p2 = particles[jm];

                double dt = p2.dt;

                p2.x += p2.vx * p2.dt;
                p2.y += p2.vy * p2.dt;
                p2.z += p2.vz * p2.dt;
                p2.t = p1.t;
                p2.dt = 0.0;
                dx = p1.x - p2.x;
```

```
dy = p1.y - p2.y;
dz = p1.z - p2.z;

if (im >= NP) {
        int m = im - NP;

        p1.vy = particles[m].vy;
        p1.vz = particles[m].vz;

        if (p1.vx * particles[m].vx < 0) {
                p1.vx = -particles[m].vx;
        }
        else {
                p1.vx = particles[m].vx;
        }
}
if (jm >= NP) {
        int m = jm - NP;

        p2.vy = particles[m].vy;
        p2.vz = particles[m].vz;

        if (p2.vx * particles[m].vx < 0) {
                p2.vx = -particles[m].vx;
        }
        else {
                p2.vx = particles[m].vx;
        }
}

q1 = (dx * p1.vx + dy * p1.vy + dz * p1.vz) / 4.0;
q2 = (dx * p2.vx + dy * p2.vy + dz * p2.vz) / 4.0;
z = q2 - q1;
p1.vx += dx*z;
p1.vy += dy*z;
p1.vz += dz*z;
z = q1 - q2;
```

```
p2.vx += dx*z;
p2.vy += dy*z;
p2.vz += dz*z;

int e1 = im;
if (im >= NP) {
        e1 = im - NP;
        particles[e1].vx = p1.vx;
        particles[e1].vy = p1.vy;
        particles[e1].vz = p1.vz;
}
else
        particles[im] = p1;

int e2 = jm;
if (jm >= NP) {
        e2 = jm - NP;

        double delta = p2.t - particles[e2].t;
        particles[e2].x += particles[e2].vx * delta;
        particles[e2].y += particles[e2].vy * delta;
        particles[e2].z += particles[e2].vz * delta;
        particles[e2].t = p2.t;

        particles[e2].vx = p2.vx;
        particles[e2].vy = p2.vy;
        particles[e2].vz = p2.vz;
}
else
        particles[jm] = p2;

if (particles[im].i_copy > 0) {
        int m = im;
        if (particles[im].i_copy < NP)
                m = m - NP;
        destroy_virt_particle(m);
        p1.i_copy = -1;
```

```
                }
                if (particles[jm].i_copy > 0) {
                        int m = jm;
                        if (particles[jm].i_copy < NP)
                                m = m - NP;
                        destroy_virt_particle(m);
                        p2.i_copy = -1;
                }

                create_virt_particle(e1);
                create_virt_particle(e2);
        }
        else
                // соударение с идеальной стенкой
                if (jm == -1) {
                        p1.vx = -p1.vx;
                        particles[im] = p1;
                }
                else {
                        if (jm != -100) {
                                short end = boxes_yz[p1.y_box][p1.z_box][p1.x_box].end;
                                Box p1_box = boxes_yz[p1.y_box][p1.z_box][p1.x_box];

                                /*
                                    Стираем частицу из ячейки в которой она находилась,
                                    вместо неё вставляем частицу из конца списка частиц в ячейке и
                                    уменьшаем число частиц в ячейке на 1.
                                */
                                p1_box.particles[p1.box_i] = p1_box.particles[end];
                                particles[p1_box.particles[end]].box_i = p1.box_i;
                                --p1_box.end;
                                boxes_yz[p1.y_box][p1.z_box][p1.x_box] = p1_box;

                                if (jm == -2) {
                                        --p1.x_box;
                                        p1.x = boxes_yz[p1.y_box][p1.z_box][p1.x_box].x2;
                                }
```

```
if (jm == -4) {
        ++p1.x_box;
        p1.x = boxes_yz[p1.y_box][p1.z_box][p1.x_box].x1;
}
if (jm == -5) {
        --p1.y_box;
        p1.y = boxes_yz[p1.y_box][p1.z_box][p1.x_box].y2;
        if ((p1.y_box == 0) && (im < NP)) {
                change_with_virt_particles(im, jm);
                p1 = particles[im];
                need_create_virt_particle = true;
        }
}
if (jm == -6) {
        ++p1.y_box;
        p1.y = boxes_yz[p1.y_box][p1.z_box][p1.x_box].y1;
        if ((p1.y_box == K) && (im < NP)) {
                change_with_virt_particles(im, jm);
                p1 = particles[im];
                need_create_virt_particle = true;
        }
}
if (jm == -7) {
        --p1.z_box;
        p1.z = boxes_yz[p1.y_box][p1.z_box][p1.x_box].z2;
        if ((p1.z_box == 0) && (im < NP)) {
                change_with_virt_particles(im, jm);
                p1 = particles[im];
                need_create_virt_particle = true;
        }
}
if (jm == -8) {
        ++p1.z_box;
        p1.z = boxes_yz[p1.y_box][p1.z_box][p1.x_box].z1;
        if ((p1.z_box == K) && (im < NP)) {
                change_with_virt_particles(im, jm);
                p1 = particles[im];
```

```
                                need_create_virt_particle = true;
                        }
                }
                if (jm < -10) {
                        /*
                           Все события, индекс которых меньше 10ти, но не равен -100,
                           это события создания "образа" для частицы.
                           Если наступает такое событие, необходимо создать образ частицы
                           не проверяя координаты частицы (т.к. мы уже знаем что частица
                           переходит в область, где у неё должен существовать "образ".
                        */
                        create_virt_particle(im, false);
                        p1 = particles[im];
                }

                /*
                   После того, как иденкс ячейки для частицы был изменён,
                   добавляем частицу в новую ячейку (или возвращаем её в старую
                   ячейку, если индекс ячейки не изменился).
                */
                p1.box_i = ++boxes_yz[p1.y_box][p1.z_box][p1.x_box].end;
                boxes_yz[p1.y_box][p1.z_box][p1.x_box].particles[p1.box_i] = im;
                particles[im] = p1;
        }
}

/*
   Если произошло события столкновения с идеальной стекой
   частицы, у которой есть образ, или образ соударяется с
   идеальной стенкой, то необходимо изменить скорость частицы
   и пересоздать образ или удалить его.
*/
if ((p1.i_copy > -1) && (jm == -1))
{
        int e = im;
        if (im >= NP) {
                e = im - NP;
```

```
                        particles[e].vx = -particles[e].vx;
                }
                clear_particle_events(e);
                create_virt_particle(e);
        }

        /*
           Если необходимо пересоздать "образ" для частицы
           после произошедшего с ней события, то вызываем функцию
           по созданию "образов".
        */
        if (need_create_virt_particle == true) {
                int e = im;
                if (im >= NP)
                        e = im - NP;
                clear_particle_events(e);
                create_virt_particle(e);
        }

        /*
          Возвращаем флаг создания образа, чтобы перерасчитать
          события для нового образа.
        */
        return need_create_virt_particle;
}


/*
   Функция "шаг", основной цикл программы,
   производит рассчёт динамики системы в течении 1 соударения
   на каждую частицу - (NP/2) столкновений в системе.
*/
void step() {
        particle p1;
        int i, im, jm;
        double time = 0.0;
        bool need_virt_particle_retime;
```

```
COLL_COUNT = 0;
jm = 0;

while (COLL_COUNT < NP / 2 || jm != -100) {
        // считываем первое событие из линейки событий
        im = time_queue[1].im;
        jm = time_queue[1].jm;

        // удаляем первое событие
        delete_event(1);

        p1 = particles[im];

        /*
           Обнуляем указатель на событие частиц, учавствующих в этом
           событии, приравнивая их -1, так, чтобы они не указывали на
           существующие события.
        */
        p1.ti = -1;
        if (jm >= 0) particles[jm].ti = -1;

        /*
           Если наступило событие -100, значит просто
           синхронизируем частицу или "образ" с которым произошло
           данное событие с глобальным временем системы.
        */
        if (jm == -100) {
                p1.dt = time - p1.t;
        }

        p1.x += p1.vx * p1.dt;
        p1.y += p1.vy * p1.dt;
        p1.z += p1.vz * p1.dt;
        p1.t += p1.dt;
        p1.dt = 0.0;
        time = p1.t;
```

```
/*
   Если у частицы есть образ и произошло событие столкновения
   с идеальной стенкой или другой частицей, то необходимо
   синронизовать образ этой частицы или частицу, которой принадлежит
   образ, с которым произошло текущее событие.
*/
if ((p1.i_copy > -1) && (jm > -2)) {
        double delta = p1.t - particles[p1.i_copy].t;
        particles[p1.i_copy].x += particles[p1.i_copy].vx * delta;
        particles[p1.i_copy].y += particles[p1.i_copy].vy * delta;
        particles[p1.i_copy].z += particles[p1.i_copy].vz * delta;
        particles[p1.i_copy].t = p1.t;
        particles[p1.i_copy].dt = 0.0;
}

particles[im] = p1;

// Производим изменения в системе согласно произошедшему событию
need_virt_particle_retime = reform(im, jm);

if (im >= NP) {
        if (particles[im].i_copy > -1)
                retime(im);
        if (jm > -2) {
                int e = im - NP;
                retime(e);
        }
}
else {
        retime(im);
        if ((particles[im].i_copy > -1) && ((need_virt_particle_retime == true)
                || (jm > -2) || ((jm < -10) && (jm != -100)))) {
                retime(particles[im].i_copy);
        }
}
```

```
                /*
                    Если jm > 0, то это значит, что произошло
                    соударение частиц в системе, увеличиваем
                    счётчик соударений
                */
                if (jm >= 0) {
                        ++COLL_COUNT;

                        if (jm >= NP) {
                                if (particles[jm].i_copy > -1)
                                        retime(jm);
                                int e = jm - NP;
                                retime(e);
                        }
                        else {
                                retime(jm);
                                if (particles[jm].i_copy > -1)
                                        retime(particles[jm].i_copy);
                        }
                }
        }

        /*
            Запускаем глобальную синхронизацию частиц по собственному времени,
            это нужно чтобы глобальное время системы стало равно нулю.

            После этой процедуры собственное время частицы может быть меньше нуля,
            что означает, что её необходимо передвинуть на |t|, чтобы она находилась
            в том же времени. Отрицательное собственное время частицы означает что
            время её ближайшего события больше, чем текущее глобальное время системы.
*/
        particle *p = particles;
        for (i = 0; i < NP * 2; ++i, ++p)
                (*p).t -= time;
        Event *t = time_queue;
        ++t;
        for (i = 1; i < last; ++i, ++t)
```

```
                (*t).t -= time;
        time = 0.0;
}


/*
  Функция получения профиля плотности системы

  Перед снятием характеристик производится синхронизация всех частиц
  с глобальным временем системы.

  Аргументы:
  steps - число замеров плотности для построения данных о профиле плотности
  accuracy - число ячеек по X на которые разбивается профиль плотности
             и в которых считается количество частиц
  file_name - имя файла для сохранения данных
*/
void image(int steps, short accuracy, std::string file_name) {
        // число ячеек, в которых будем рассчитывать количество частиц
        const int W = (int(L) + 1) * 2 * accuracy;

        // массив, в который будем собирать данные по количеству частиц
        int img[10000];
        double t_global, dt;

        printf("INFO: Image started for %d steps with accuracy %d\n", steps, accuracy);

        // заполняем массив нулями, прежде чем считать количество частиц
        for (short g = 0; g < 10000; ++g) img[g] = 0;

        for (short h = 0; h < steps; ++h) {
                step();  // далем одно соударение на частицу между замерами

                // очистка экрана и вывод информации о прогрессе
                printf("\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b");
                printf("%d / %d", h + 1, steps);
```

```
                /*
                    Рассчитываем глобальное время системы, которое равно
                    самому большому собственному времени частиц в системе.
                */
                t_global = get_maximum_particle_time();

                for (int i = 0; i < NP; ++i) {
                        /*
                            Синхронизируем частицу с глобальным временем системы
                        */
                        dt = t_global - particles[i].t;

                        /*
                            Определяем в какую из ячеек по X попадает данная частица
                            после синхронизации её по времени с глобальным временем системы
                        */
                        int m = int((L + particles[i].x + particles[i].vx * dt) * 10.0 * accuracy) / 10;
                        ++img[m];  // увеличиваем количество частиц в ячейке на 1.
                }
        }

        double x = 0.0, delta_x = 1.0 / accuracy;
        FILE *profile_file = fopen(file_name.c_str(), "w+");
        for (int f = 0; f < W; ++f) {
                /*
                    Сохраняем начальную координату X для ячейки и число частиц,
                    которые находились в данном "слое" во время измерений
                */
                fprintf(profile_file, "%f %d\n", x, img[f]);
                x += delta_x;
        }
        fclose(profile_file);

        printf("\n INFO: Image completed. Information saved to file: %s \n", file_name.c_str());
}
```

```
/*
    Функция создания разреза системы в пике,
    позволяет просматривать расположение частиц в слое,
    параллельном идеальной стенке.

    Аргументы:
     x1 - начальная X координата "среза"
     x2 - конечная X координата "среза"
     dots_per_particle - количество снятий данных через равное количество соударений в системе
     steps - число соударений между двумя снятиями данных о положении частиц в выбранном слое
     file_name - имя файла для сохранения данных
*/
void profile(double x1, double x2, int dots_per_particle, int steps, std::string file_name) {
        double x, y, z, dt;
        int i, j;

        // открываем файл на запись
        FILE *profile_file = fopen(file_name.c_str(), "w+");

        printf("INFO: Profile started.\n");

        for (i = 0; i < dots_per_particle; ++i) {
                /*
                    Между измерениями данных о положениях частиц
                    делаем указанное количество соударений на частицу в системе
                */
                for (j = 0; j < steps; ++j)
                    step();

                double t_global = get_maximum_particle_time();

                for (j = 0; j < NP; ++j) {
                        /*
                            Синхронизируем частицу по времени с глобальным
                            временем системы и проверяем её координату X,
                            если частица находится в указанном диапазоне,
                            то сохраняем её Y и Z координаты в файл.
```

```
                        */
                        dt = t_global - particles[j].t;
                        x = L + particles[j].x + particles[j].vx * dt;

                        if (x <= x2 && x >= x1) {
                                y = A + particles[j].y + particles[j].vy * dt;
                                z = A + particles[j].z + particles[j].vz * dt;

                                // сохраняем Y и Z координаты частицы
                                fprintf(profile_file, "%.15le\n", y);
                                fprintf(profile_file, "%.15le\n", z);
                        }
                }
        }

        fclose(profile_file);

        printf("INFO: Profile completed. Information saved to file: %s\n", file_name.c_str());
}


/*
  Функция сжатия системы, позволяет сжимать или расширять систему до заданной плотности
  с заданным максимальным шагом по плотности.

  Аргументы:
   compress_to_etta - итоговая плотность
   delta_L - максимальный разрешённый шаг по L
   steps - количество соударений на одну частицу в системе между
           маленькими сжатиями по плотности
   type - тип сжатия:
           0 - пододвигать только левую стенку
               1 - пододвигать только правую стенку
               2 - пододвигать одновременно обе стенки на одинаковое расстояние
*/
void compress(double compress_to_etta, double delta_L, int steps, int type) {
        int m;
```

```
double etta = (PI * NP) / (6.0 * A * A * (L - 1.0));
double min = 1.0e+100, max = -1.0, x, dL, dx;
double t_global, dt;
double L_ideal = ((PI * NP) / compress_to_etta) / (6.0 * A * A) + 1;

printf("\n INFO: Start to change system density ... \n");

printf(" \n  Maximum delta_L = %.15le \n", delta_L);
printf("  Program will wait %d collissions per particle between each change in density. \n", steps);
printf("  Type of compression: ");
if (type == 0) printf("only position of left wall will be changed");
if (type == 1) printf("only position of right wall will be changed");
if (type == 2) printf("position of both walls will be changed \n\n");

printf("etta = %.15le , should be equal to %.15le \n", etta, compress_to_etta);

// задаём плотность с точностью в 12 знаков
while (fabs(etta - compress_to_etta) > 1.0e-12) {
        if (etta < compress_to_etta) {
                max = -100.0;
                min = 1.1e+10;

                /*
                   Смещаем все частицы в текущее время системы чтобы синхронизовать
                   все частицы и затем находим координаты частиц, наиболее близко
                   расположенных от идеальных стенок, чтобы знать на сколько можно
                пододвинуть стенку не коснувшись частиц.
                */
                t_global = get_maximum_particle_time();

                for (int i = 0; i < NP; ++i) {
                        dt = t_global - particles[i].t;
                        x = L + particles[i].x + particles[i].vx * dt;
                        if (x < min) min = x;
                        if (x > max) max = x;

                        /*
```

```
            Если у данной частицы есть образ, то синхронизируем его по времени
            со временем системы и проверяем на сколько он близко находится
            к идеальным стенкам
        */
        if (particles[i].i_copy > 0) {
                m = particles[i].i_copy;
                dt = t_global - particles[m].t;
                x = L + particles[m].x + particles[m].vx * dt;
                if (x < min) min = x;
                if (x > max) max = x;
        }
}

/*
    Рассчитываем на сколько близко частицы находятся к первой
    и второй идеальным стенкам
*/
min = min - 1.0;
max = 2.0 * L - max - 1.0;

// выбираем наименьшее и этих расстояний и сохраняем в dL
dL = min;
if (dL > max) dL = max;

// сжимаем не впритык к частицам и не слишком быстро
dL = dL / 1.1;
if (dL < 0.1e-12) dL = 0.01e-30; // не двигаем стенку если частицы близко
if (dL > delta_L) dL = delta_L;  // сдвигаем стенку не больше чем на delta_L

dx = dL / 2.0;  // рассчитываем смещение для всех частиц

/*
    Если следущее смещение стенки сожмёт систему больше
    чем требуется (т.е. относительная плотность частиц в системе etta
    будет больше / меньше той, которая была указана а аргументах),
    то необходимо задать точное значение L, чтобы плотность совпала
    с требуемой плотностью.
```

```
            */
            if (L - dL < L_ideal) {
                dL = 0.0;
                L = L_ideal;
            }
        }
        else {
            /*
               Если систему необходимо расширить, то просто берём
               наибольший разрешённый шаг для изменения координаты стенки
               и сдвигаем идеальную стенку или две стенки.
            */
            dL = L - L_ideal;
            if (dL < -delta_L) dL = -delta_L; // шаг расширения системы
            dx = dL / 2.0;
        }

        if (type == 0) {
            /*
               Двигаем все частицы влево, таким образом пододвигая только левую стенку.
               расстояние частиц до правой стенки не изменится, т.к. после перемещения частиц
               влево мы сдвигаем обе стенки на то же расстояние - в итоге мы сдвинем все
               частицы влево на dL/2.0 и пододвинем обе стенки к центру системы на dL/2.0.
            */
            for (int w = 0; w < NP; ++w) {
                particles[w].x -= dx;
                if (particles[w].i_copy >= NP) particles[particles[w].i_copy].x -= dx;
            }
        }
        if (type == 1) {
            /*
               Двигаем только правую стенку (всё то же самое что и для левой стенки, но
               в этом случае все частицы смещаются вправо на dL/2.0)
            */
            for (int w = 0; w < NP; ++w) {
                particles[w].x += dx;
                if (particles[w].i_copy >= NP) particles[particles[w].i_copy].x += dx;
```

```
                }
            }

            // изменяем систему - происходит мгновенное изменение координат двух стенок
            L -= dL;

            /*
               Сохраняем состояние системы и снова загружаем его пересчитав новые
               параметры и проведя необходимую инициализацию
            */
            save("tmp");
            load_seed("tmp");

            /*
               После каждого смещения стенки делаем указанное количество соударений
               на частицу в системе
            */
            for (short i = 0; i < steps; ++i)
                step();

            // рассчитываем плотность после сжатия
            etta = (PI * NP) / (6.0 * A * A * (L - 1.0));

            printf("etta = %.15le, should be equal to %.15le   \n", etta, compress_to_etta);
        }
        printf("\n INFO: System density was sucessfully changed to %.15le \n", etta);
}


/*
   Функция проведения эксперимента по указанному в
   текстовом файле описанию.

   Аргументы:
    file_name - имя файла с описанием шагов эксперимента для программы.
*/
void init(std::string file_name) {
```

```
using namespace std;
clock_t start, end, result;
char command[255], parameter[255];
long long int i, steps;
ifstream command_file(file_name.c_str());

while (!command_file.eof()) {
        command_file.getline(command, 255, ' ');
        string str_command = command;

        printf("\n\n<==========>\n");

        // Если необходимо создать новый посев
        if (str_command.compare("new") == 0) {
                int NN;
                double etta;

                command_file >> NN;
                command_file >> etta;
                command_file.getline(parameter, 255, '\n');  // завершить считывание строки
                new_seed(NN, etta);

                print_system_parameters();
        }
        // Если необходимо загрузить состояние системы из файла
        if (str_command.compare("load") == 0) {
                command_file.getline(parameter, 255, '\n');
                load_seed(parameter);

                printf("\n System was successfully loaded from file '%s' \n", file_name.c_str());

                print_system_parameters();
        }
        // Если необходимо рассчитать динамику системы в течении
        // указанного количества соударений
        if (str_command.compare("step") == 0) {
                command_file >> steps;
```

```
            command_file.getline(parameter, 255, '\n');  // завершить считывание строки

            printf("␣INFO:␣Step␣start.␣\n");

            start = clock();

            for (i = 0; i < steps; ++i) {
                    step();
                    printf("\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b");
                    printf("%d␣/␣%d", i + 1, steps);

                    // если мы пишем файл с историей событий,
                    // то очищать его каждые 1000 соударений на частицу,
                    // чтобы он не превышал допустимых размеров.
                    if (i % 1000 == 0) {
                            FILE *history_file = fopen("history.txt", "w+");
                            fclose(history_file);
                    }
            }

            end = clock();
            result = end - start;

            printf("\n␣INFO:␣finished␣%d␣collisions␣per␣particle␣\n", steps);
            printf("Total␣Time␣=␣%f␣seconds.␣\n", double(result / CLOCKS_PER_SEC));

            /*
               Рассчитываем момент импульса системы Lx и выводим его на экран.
            */
            double Lx = 0.0;
            for (int i = 0; i < NP; i++) {
                    Lx += (particles[i].y + A)*particles[i].vz - (particles[i].z + A)*particles[i].vy;
            }
            printf("\n␣Lx␣=␣%.15le\n", Lx);
    }
    // если необходимо собрать данные по профилю плотности в системе
    if (str_command.compare("image") == 0) {
```

```
            command_file >> steps; // число соударений на одну частицу за время измерений
            command_file >> i; // точность. число точек графика на один радиус частицы
            command_file.getline(parameter, 255, '\n');
            image(steps, i, parameter);
    }
    // если необходимо собрать данные по "разрезу" в системе
    if (str_command.compare("profile") == 0) {
            int dots_for_each_particle;
            double x1, x2;
            command_file >> x1;
            command_file >> x2;
            command_file >> dots_for_each_particle;
            command_file >> steps;
            command_file.getline(parameter, 255, '\n');
            profile(x1, x2, dots_for_each_particle, steps, parameter);
    }
    // если необходимо сохранить состояние системы
    if (str_command.compare("save") == 0) {
            command_file.getline(parameter, 255, '\n');
            save(parameter);
            printf("\n INFO: particles coordinates saved to '%s' \n", parameter);
    }
    // если необходимо изменить плотность системы
    if (str_command.compare("compress") == 0) {
            command_file.getline(parameter, 255, '\n');  // завершить считывание строки
            printf("\n Sorry, 'compress' command was deprecated, we need to use \n");
            printf("'compress_two_walls'(short form: 'compresst') instead. You can also use ");
            printf("'compress_left_wall'(short form: 'compressl') \n or 'compress_right_wall'");
            printf("(short form: 'compressr') commands to change system density.\n");
            exit(1);
    }
    // если необходимо изменить плотность системы
    // сжимать, сдвигая две идеальные стенки
    if ((str_command.compare("compress_two_walls") == 0) ||
            (str_command.compare("compresst") == 0)) {
            double etta, delta_etta;
            command_file >> etta; // требуемая плотность
```

```
                command_file >> delta_etta;  // минимальное допустимое значение изменения плотности
                command_file >> steps;  // количество соударений после каждого шага сжатия
                command_file.getline(parameter, 255, '\n');  // завершить считывание строки
                compress(etta, delta_etta, steps, 2);

                print_system_parameters();
            }
            // если необходимо изменить плотность системы
            // двигать только "левую" идеальную стеку, x = 0
            if ((str_command.compare("compress_left_wall") == 0) ||
                (str_command.compare("compressl") == 0)) {
                double etta, delta_etta;
                command_file >> etta; // требуемая плотность
                command_file >> delta_etta;  // минимальное допустимое значение изменения плотности
                command_file >> steps;  // количество соударений после каждого шага сжатия
                command_file.getline(parameter, 255, '\n');  // завершить считывание строки
                compress(etta, delta_etta, steps, 0);

                print_system_parameters();
            }
            // если необходимо изменить плотность системы
            // двигать только "правую" идеальную стеку, x = L
            if ((str_command.compare("compress_right_wall") == 0) ||
                (str_command.compare("compressr") == 0)) {
                double etta, delta_etta;
                command_file >> etta; // требуемая плотность
                command_file >> delta_etta;  // минимальное допустимое значение изменения плотности
                command_file >> steps;  // количество соударений после каждого шага сжатия
                command_file.getline(parameter, 255, '\n');  // завершить считывание строки
                compress(etta, delta_etta, steps, 1);

                print_system_parameters();
            }
            if (str_command.empty()) break;
        }
        FILE *result_flag = fopen("result", "w+");
        fclose(result_flag);
```

```
}

/*
   Стартовая функция для программы, считать файл "program.txt"
   и начать выполнять эксперимент, описанный в этом файле.
*/
int main()
{
        FILE *history_file = fopen("history.txt", "w+");
        fclose(history_file);

        init("program.txt");
        return 0;
}
```

16.3 Дополнительные материалы