САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра Системного Программирования
Группа 20.Б11-мм

Сульдин Вячеслав Романович

Влияние момента центра инерции на движение броуновских частиц

Отчет по учебной практике
в форме «Эксперимент»

Научный руководитель:
профессор кафедры ПА, д.ф.-м.н. Прозорова Э. В.

Санкт-Петербург
2022

Содержание

Введение

Знание пространственной структуры молекул позволяет понять и спрогнозировать протекание тех или иных химических процессов, объяснить свойства исследуемого вещества или даже синтезировать новое с заданными полезными свойствами, такие как наноструктуры.

В настоящее время, благодаря стремительному развитию вычислительной техники, приблизиться к решению этих задач позволяют методы компьютерного моделирования. Во многих случаях они оказываются единственным способом получения детальных количественных сведений о поведении молекулярных систем, как известных в природе, так и еще планируемыхк созданию. Сопоставляя результат вычислений с опытными данными, можно выявить наиболее важные факторы и закономерности, отвечающие за те или иные свойства реальных молекул.

С другой стороны, компьютерное моделирование часто выступает в качестве связуещего звена между теорией и физическим экспериментом. Наконец, компьютерное моделирование во многих случаях является мощным средством повышения информативности самих экспериментальных методов исследований молекул и позволяют целенаправленно выполнять эксперимент, ускоряя его проведение.

Одним из «прогнозирующих» компьютерных методов является метод молекулярной динамики (МД). Молекулы рассматриваются как система взаимодействующих классических частиц. Используются различные модели взаимодействия, методы расчёта движения.

Распространенную модель движения броуновских частиц описывает уравнение Ланжевена. Она включает в качестве дополнительного фактора нерегулярную (статистическую) силу.

Нам бы хотелось проверить и исследовать вычислительным методом решение задачи о броуновском движении, используя иную модель движения. Её главное отличие заключается в том, что сила детерменирована и обусловлена влиянием момента центра инерции на молекулы.

1 Цели и задачи

Целью данной работы является проверка корректности влияния момента центра инерции на движение броуновских частиц. Для её выполнения были поставлены следующие задачи:

1. Изучение предметной области и тонкостей при работе с непрерывными величинами в ЭВМ,
2. Внедрение необходимого количества математичиских и физических абстракций для удобной работы с объектами,
3. Реализация модели движения молекул Аргона,
4. Определение влияния момента центра инерции,
5. Создание визуализации для отладки и презентации.
6. Добавление тяжелой частицы. Исследование модели:
   - Построение графика кинетической энергии (проверка выполнения законов сохранения),
   - Расчёт вязкости системы,
   - Годограф движения броуновской частицы,
7. Анализ полученных результатов. Подведение итогов эксперимента.

2 Обзор предметной области

2.1 Классическая механика

При моделировании молекулы полагаются твердыми сферами. Значит, они подчиняются уравненяем движения. Задача заключается в определении положения и скорости каждой частицы в моменты времени, и расчёт по данной траектории их новых координат.

На каждом промежутке, который получили после дискретизации, мы рассматриваем движение как равноускоренное. Тогда можно использовать следующие формулы:

Второй закон Ньютона

$$\vec{F} = m\vec{a} \tag{1}$$

Чтобы вычислить координаты материальной точки через время t

$$x = x_0 + v_0 t + \frac{a}{2}t^2 \tag{2}$$

А скорость в свою очередь вычисляется как

$$v = v_0 + at \tag{3}$$

Тем не менее, полезными оказываются другие алгоритмы вычисления движения. Уменьшая погрешность вычислений, они в большей степени обеспечивают сохранение кинетической энергии. Далее будем верхними индексами обозначать временной шаг, или номер конфигурации.

- **Алгоритм Верле в скоростной форме**

$$x^{i+1} = x^i + v^i \Delta t + a^i \frac{\Delta t^2}{2} \tag{4}$$

$$a^i = \frac{F^i}{m} \tag{5}$$

$$v^{i+1} = v^i + \frac{(a^i + a^{i+1})\Delta t}{2} \tag{6}$$

- **Алгоритм с полушагом**

$$x^{i+1} = x^i + v^i \Delta t + a^i \frac{\Delta t^2}{2} \tag{7}$$

Промежуточное значение скорости

$$v^{\frac{i+1}{2}} = v^i + \frac{a^i \Delta t}{2} \tag{8}$$

Новое значение ускорения

$$a^{i+1} = \frac{F^{i+1}}{m} \tag{9}$$

Новое значение скорости

$$v^{i+1} = v^{\frac{i+1}{2}} + \frac{a^{i+1}\Delta t}{2} \tag{10}$$

2.2 Распределение Максвелла

Функция распределения Максвелла показывает, какова вероятность того, что скорость данной молекулы имеет значение, заключенное в единичном интервале скоростей, включающем данную скорость u, или каково относительное число молекул, скорости которых лежат в этом интервале.

Распределение значений проекции скорости v_x:

$$\Psi(v_x) = \left(\frac{m}{2\pi kT}\right)^{1/2} exp\left(\frac{mv_x^2}{2kT}\right) \tag{11}$$

- m - масса молекулы
- k - постоянная Больцмана
- T - температура, К

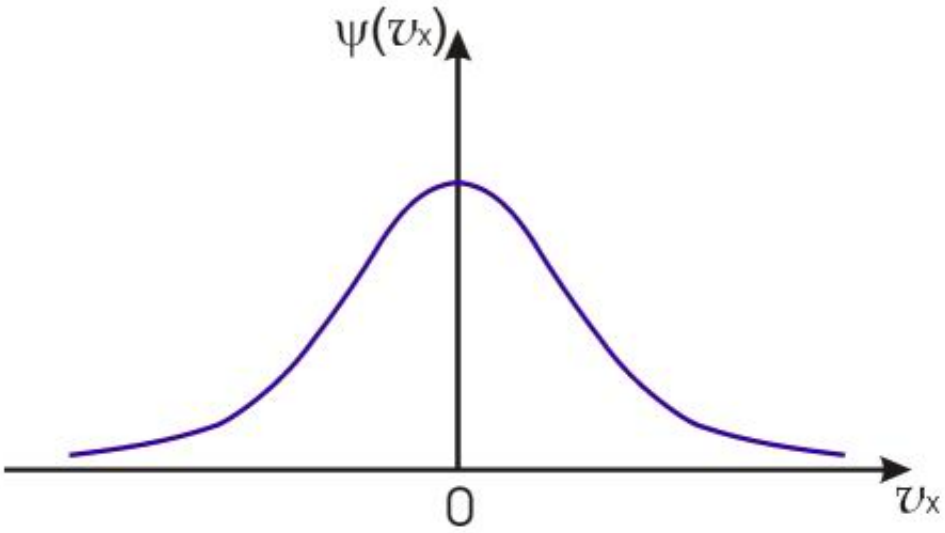

Рис. 1: График $\Psi(v_x)$

Данная функция в будущем () поможет нам задать начальные скорости молекул, не нерушая равновесности системы.

2.3 Потенциал Леннарда-Джонса

Самая распространенная модель бинарного взаимодействия молекул - потенциал Леннарда-Джонса.

$$U(r) = \varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - 2 \left(\frac{\sigma}{r} \right)^{6} \right] \tag{12}$$

- r - расстояние между молекулами
- ε - глубина потенциальной ямы
- σ - длина связи

При больших r молекулы притягиваются, что соответствует члену $\left(\frac{\sigma}{r}\right)^6$. На малых же расстояниях молекулы начинают сильно отталкиваться. $\left(\frac{\sigma}{r}\right)^{12}$

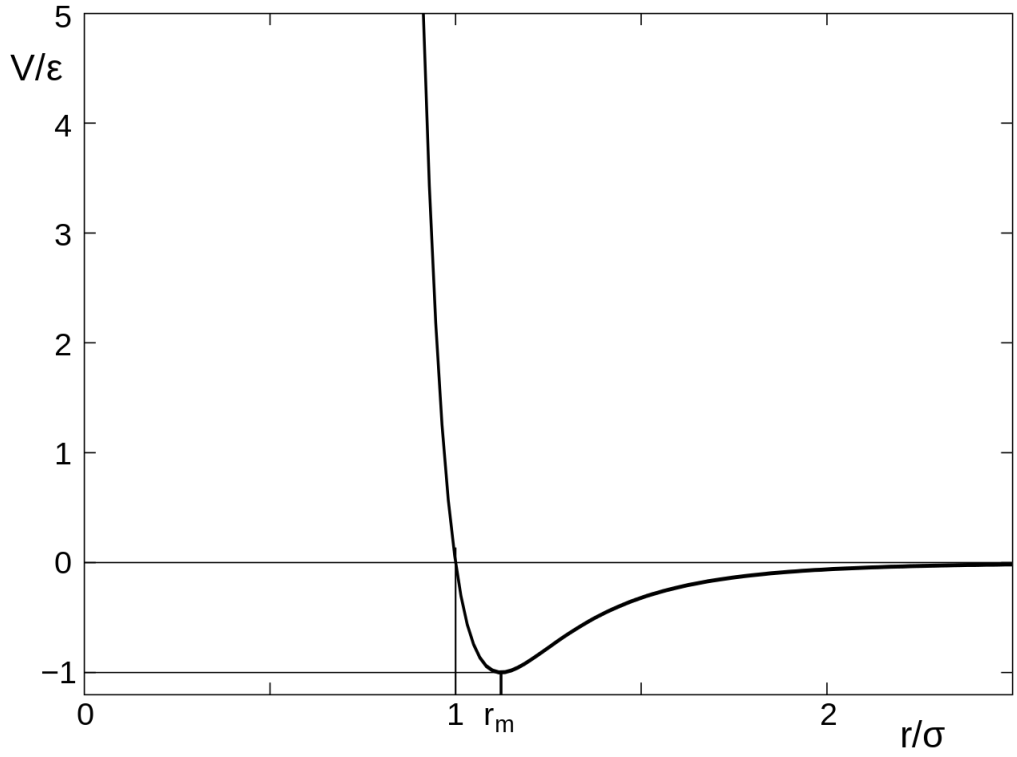

Рис. 2: График $U(r)$

Нас скорее будет интересовать сила, обусловленная этим потенциалом. Она есть ничто иное, как производная, взятая с другим знаком.

$$F(r) = -\frac{dU(r)}{dr} \tag{13}$$

И, вычисляя производную:

$$F(r) = \frac{12\varepsilon}{\sigma} \left[\left(\frac{\sigma}{r} \right)^{13} - \left(\frac{\sigma}{r} \right)^{7} \right] \tag{14}$$

Стоит отметичть, что равенство достигается при $dr \to 0$, чего невозможно обеспечить при компьютерном моделировании, но чем меньше dr, тем точнее будут результаты.

2.4 Центр инерции

Повторяясь, при исследовании методами МД мы считаем молекулы сферами, а значит у системы молекул есть центр инерции. И находиться он так же, как и в классической механике.

$$C = \frac{\sum_{i=1}^{n} x_i m_i}{\sum_{i=1}^{n} m_i} \tag{15}$$

Момент центра инерции - такой же момент силы, только расстояние до объекта отсчитывается от центра инерции.

$$M_c = F * |C - X| \tag{16}$$

Изменение момента центра инерции, обуслевленное изменение центар инерции и силы, действующей на молекулу, приводит к возникновению силы. Она вычисляется как производная по направлению вектора изменения, или градент.

$$F_c = \nabla M_c \tag{17}$$

3 Ход работы

3.1 Выбор инструментов разработки

В качестве языка разработки был выбран C++, так как при должном желании он выделяется скоростью и точностью, что особенно важно при большом количестве математических вычислений.

Для наглядности, отладки и построения графиков я выбрал SFML - простую и главное быструю библиотеку для создания оконных приложений.

3.2 Подготовка

Для начала хотелось определить единицы, с которыми мы будем активно работать. Своё отражение это желание нашло в следующих модулях.

3.2.1 Константы

Для автоматизированной работы с приложением я определил константы. Рассмотрим интерфейс.

1: Constants.h

```
// Число молелкул в стороне куба из молекул
const int MOL_SIDE;
// = MOL_SIDE^3
const int NUMBER_OF_MOLECULES;
// Временной шаг
const double DELTA_T;
// Длинна окна приложения, 1000 px
// Высота окна приложения
const int X;
const int Y;
// Макс. расстояния действия сил
const double FORCE_RADIUS;
// См. Л-Дж, = 3.54 A
const double SIGMA;
// См. Д-Дж, = 0.00801 eB
const double E;
// 6.6e-26
const double ARGON_MASS;
// Коэф. для отображения системы на приложение
const double SCALE;
// Нач. Расстояние между молекулами
const double DIST;
// Длинна стороны куба системы
const double SIDE_OF_SYSTEM;
```

```
25 // 1.92e-10
26 const double ARGON_RADIUS;
```

И сразу к реализации. Зададим зависимость от числа молекул. Определенные константы оставим в «...» для краткости.

2: Constants.cpp

```
1  // ПУсть равно некоторому n
2  const int MOL_SIDE = n;
3  // Очевидно
4  const int NUMBER_OF_MOLECULES = pow(MOL_SIDE, 3);
5  ...
6  /* С первого взгляда на график потенциала Л-Дж ясно,
7    что после 2*SIGMA сила пренебрежима мала. */
8  const double FORCE_RADIUS = 2 * SIGMA;
9  ...
10 // Растяжение
11 const double SCALE = SIDE_OF_SYSTEM / X;
12 // Пусть изначально молекулы не взаимодействуют
13 const double DIST = FORCE_RADIUS;
14 /* А теперь полученный куб из молекул уложим в систему.
15    Длину ребра легко найти */
16 const double SIDE_OF_SYSTEM = DIST * MOL_SIDE;
17 ...
```

Теперь, выбирая лишь число молекул, все остальные характеристики подтягиваются сами. Это упрощает использование. Также довольно просто переопределить расстояние между молекулами, выбирая концентрацию.

3.2.2 Класс Vector

В работе мы производим много вычислений нескалярных, то есть векторных величин. Для сокращения количества кода и более привычных записей будет удобно создать инструмент работы с векторами. Для этого перегрузим нужные операторы и добавим функции на векторах, которые могут пригодиться.

3: Vector.h

```
class Vector
{
public:
    Vector();
    Vector(double x, double y, double z);
    double x, y, z;

    // Сложение векторов - покоординатно
    Vector operator+(Vector v);
    Vector operator-(Vector v);

    // Умножение на элемент поля
    Vector operator*(double l);
    Vector operator/(double l);

    // Лаконично суммировать.
    Vector& operator+=(Vector v);
    // Унарный минус
    Vector& operator-();

    void print();
    // Получить направление
    Vector normalize();
    // Длинна вектора
    double length();
};

double distance(Vector v, Vector u);
// Получить нулевой вектор.
Vector null();
```

Реализация класса тривиальна.

3.2.3 Класс Molecule

Самым важным объектом исследования является молекула. Но для начала введем класс-помощник, несущий в себе два вектора

4: Molecule.h

```
class Delta
{
public:
    Delta(Vector prev, Vector cur);
    // Previous - предыдущее значение.
    Vector prev;
    // Current - текущее значение
    Vector cur;
    // Разница между ними. (Векторный смысл)
    Vector delta();
};
```

Мотивация в его создании следует из алгоритмов, в явном виде использующих пред. и тек. значения, а также в необходимости обоих значений сразу при вычисления момента инерции.

Перейдем к Молекуле.

5: Molecule.h

```
class Molecule
{
public:
    Molecule(Vector coor, Vector vel);

    // Три вида алгоритмов расчёта движения
    void base();
    void semiStep();
    void verlet();

    // Проверка на периодические условия
    void periodic();
    // Разрешающая столкновения функция
    void collide(Molecule &with);

    // Координаты: Текущиее и предшествующие
    Delta coordinates;

    // Сила: аналогично
    Delta force;

    // Текущая скорость молекулы
    Vector velocity;

```

```
25      double mass = ARGON_MASS;
26      double radius = ARGON_RADIUS;
27  };
```

Забежим вперёд. Функция periodic проверяет, не вылетела ли молекула из системы, в противном случае возвращает её обратно. Её реализации я выделил свой раздел 3.5. Аналогично collide проверяет и решает ситуацию столкновения 3.6.

А пока, разберём одну из функций расчета движений. Остальные идеологически похожи. Вспомним формулы из 2.1. Уточню, что $i + 1$ соответсвует текущий момент, а i предыдущий. Нетрудно это переписать.

6: Molecule.cpp

```
1  void Molecule::verlet()
2  {
3      // Старые принимают значения текущих
4      coordinates.prev = coordinates.cur;
5      // Найдем ускорение до действия силы (acceleration)
6      Vector acc = force.prev / mass;
7      // Новые координаты
8      coordinates.cur += (velocity * DELTA_T) +
9          acc * (pow(DELTA_T, 2) / 2);
10     // Текущее ускорение
11     Vector accN = force.cur / mass;
12     // И посчитаем скорость для текущего момента
13     velocity += (acc + accN) * (DELTA_T / 2);
14     // Проверяем координаты молекулы
15     periodic();
16 }
```

Уже такой небольшой аппарат позволяет, добавив силы, моделировать движение молекул. Но пока мы имеем информацию только в числах.

3.2.4 Визуализация. SFML

Данный раздел познакомит с самым базовым арсеналом библиотеки SFML на примере решения задачи визуализации процесса движения молекул. В нём вы можете увидеть как изобразить трёх-мерное пространство на экране, как сделать много окон, как можно строить графики в реальном времени.

Подключение библиотеки:

- sudo apt-get install libsfml-dev
- # include <SFML/Graphics.hpp>
- g++ main.o -o sfml-app -lsfml-graphics -lsfml-window -lsfml-system

Ключевые конструкции:

7: Some.cpp

```
// Создание окна X на Y с именем "title"
sf::RenderWindow window(sf::VideoMode(200, 200), "title");
// Создание фигуры (круга) радиуса 100.f
sf::CircleShape shape(100.f);
// Присвоение ей цвета Green
shape.setFillColor(sf::Color::Green);
// Удаление всех объектов с окна
window.clear();
// Отобразить фигуру на окне
window.draw(shape);
// Обновить окно
window.display();
```

Этого нам хватит, чтобы визуализировать все вышеперечисленное. Запуская нашу симуляцию, нам бы хотелось видеть сразу несколько окон, (сами молекулы, график a, график b...). Но нельзя забывать об удобстве. Чтобы создать и поддерживать новое окно, нужно проделывать однотипные вышеперечисленные действия. Занести их все в массив - тоже не вариант: с ростом их количества растёт вероятность ошибки. Чтобы автоматизировать этот процесс, было принято решение создать класс, содержащий в себе все окна, и обеспечивающий удобную работу с ними.

Я выбрал путь через X-Макросы. Они не сложны для понимания, но очень универсальны в использовании, и я нашел в них выход. Вот несколько полезных материалов (). Тем временем, вот тот самый класс, разберём его.

8: Visualization.h

```
/* Для того, чтобы добавить новое окно, нам
   потребуется лишь дописать его название в
   список, остальное сделает за нас препроцессор */
#define LIST_OF_WINDOW \
    X(main)            \
    X(kinetic)         \
    X(force)

class App
{
public:
    /* Похоже на main, и не случайно: не все окна
       потребуются при каждом запуске. Нужные будем
       передавать в качестве аргументов, например
       ./sim main force                          */
    App(int argc, char *argv[]);

    /* Проще показать, на что заменятся, возможно
       пугающие, строки 25-27:
     sf::RenderWindow main;
     sf::RenderWindow kinetic;
     sf::RenderWindow force;
      чудо! Для каждого из списка мы создали по
      полю, сделав работу с окнами более строгой. */
#define X(name) sf::RenderWindow name;
    LIST_OF_WINDOW
#undef X

    void add_window(sf::RenderWindow *window, sf::String
       title);
    // Проверка на желание закрыть окно
    void is_close();
    // display для всех окон
    void display();
    // Сравнение с аргументом и добавление окна
    void add_if(const char *name, char *cmd, sf::
       RenderWindow *window);
};
```

Осталось еще организовать своеобразный дескриптор комманд, распознающий нужный список. Рассмотрим функции по отдельности.

9: Visualization.cpp

```
/* Дело в эквивалентности выражений:
sf::RenderWindow window(sf::VideoMode(X, Y), title)
и
```

```
4 sf::RenderWindow window;
5 window.create(sf::VideoMode(X, Y), title);
6 */
7 void App::add_window(sf::RenderWindow *window, sf::String
      title)
8 {
9     window->create(sf::VideoMode(X, Y), title);
10 }
```

10: Visualization.cpp

```
1 /* По полученному имени и строки из аргументов,
2    взависимости от результата сравнения добавляем
3    или нет окно.                                    */
4 void App::add_if(const char *name, char *cmd, sf::
      RenderWindow *window)
5 {
6     if (!strcmp(name, cmd))
7     {
8         add_window(window, (sf::String)name);
9     }
10 }
```

11: Visualization.cpp

```
1 /* Создавая объект, инициализируем требуемые окна
2    на основе нашего списка. Для каждого слова в
3    аргументах мы запустим add_if.          */
4 App::App(int argc, char *argv[])
5 {
6     for (int i = 0; i < argc; i++)
7     {
8 #define X(name) add_if(#name, argv[i], &name);
9         LIST_OF_WINDOW
10 #undef X
11     }
12 }
```

12: Visualization.cpp

```
1 // Обновляем все окна
2 void App::display()
3 {
4 #define X(name) name.display();
5     LIST_OF_WINDOW
6 #undef X
7 }
```

13: Visualization.cpp

```
/* Здесь спрятана проверка на закрытие для каждого
   из окон. В противном случае окно очищается    */
void App::is_close()
{
    sf::Event event;
#define X(name)                                    \
    {                                              \
        sf::Event event;                           \
        while (name.pollEvent(event))              \
        {                                          \
            if (event.type == sf::Event::Closed)   \
                name.close();                      \
        }                                          \
        name.clear();                              \
    }
    LIST_OF_WINDOW
#undef X
}
```

Подводя итог, чтобы добавить новое окно нужно:

- - добавить его в LIST_OF_WINDOW
- - упомянуть при запуске приложения в аргументах

Всю остальную работу с окном берёт на себя класс. Это практично.

Построим графики по новоприбывающим значениям. Основная из преследуемых целей: сравнивать эти графики. Это говорит о необходимости размещать их на одном окне. Поэтому по аналогии с фигурами, будем разделять их от окон. Класс графика:

14: Graph.h

```
class Graph
{
public:
    Graph(sf::Color color);
    // Добавление нового значения в график
    void update_graph(sf::RenderWindow *window, double
        new_value);
    // Отображаемые значения
    std::vector<double> values;
    // Верхнее значение
    double top = 0;
    // Цвет графика
    sf::Color color;
};
```

```
14 void draw(sf::RenderWindow *window, float radius, sf::Color
       color, double x, double y, double scale);
```

Наибольший интерес представляет функция update_graph. Напоминаю, что рисуем на окне X на Y. За ось ординат принято Y. Рисуем по пикселям, значений столько, сколько X.

15: Graph.cpp

```
1  void Graph::update_graph(sf::RenderWindow *window, double
       new_value)
2  {
3    // Если значений в массиве еще недостаточно, просто добав
       ляем
4      if (values.size() < X)
5      {
6          values.push_back(new_value);
7      }
8      // Иначе циклически сдвинуться, забыв последнее значени
         е и добавив новое
9      else
10     {
11         for (int i = 0; i < values.size(); i++)
12         {
13             values[i] = values[i + 1];
14         }
15         values[values.size()] = new_value;
16     }
17     // Обновляем top
18     top = new_value > top ? new_value : top;
19     // Рисуем все точки
20     for (int i = values.size(); i > 1; i--)
21     {
22           // аргументы draw: куда, толщина, цвет, X, Y,
                scale
23         draw(window, 2, color,
24              i,
25              (values[i] / top*0.75) * Y, 1);
26     }
27 }
```

Полученный график будет занимать достигать не более 3/4 от высоты экрана, и иногда растягиваться по вертикали, так он будет всегда в поле зрения для анализа. Для ускорения можно использовать циклические структуры данных, менять верхниюю границу (зафиксировать, сделать для каждого графа общую). Напомню, что левый верхний угол имеет координаты (0,0), а правый нижний (X,Y), тоесть большему значению соответствует более низкая точка.

Наконец, нарисуем молекулы. Конечно нам хочется изобразить трехмерность среды. Есть много способов. Например использовать вместо SFML OpenGL. Но, на мой взгляд, это слишком мощный инструмент, тем более который может съесть львиную долю производительности, делая акцент на трехмерную графику. Нам же хватит двумерной иллюзии. Снова пригодиться функция draw.

16: Main.cpp

```
1       // "Само" появившееся .main, рисуем в него
2  draw(&visualization.main,
3       10 + ((X * ARGON_RADIUS / (2 * MOL_SIDE * SIGMA)) /
           SIDE_OF_SYSTEM) * molecules[i].coordinates.cur.z,
4       // Цвет
5       sf::Color::Cyan,
6       // X и Y берём исходные.
7       molecules[i].coordinates.cur.x,
8       molecules[i].coordinates.cur.y,
9       // Поскольку передаём значения не в пикселях, нужно от
           масштабировать.
10      SCALE);
```

Подробнее про второй аргумент. Достигать объёмности будем за счёт изменения размера. Чем ближе - больше молекула, и наоборот. За это отвечает z координата. Но . Хотим, чтобы размер, тот что в () обращался в 0, когда $z = 0$ (далеко), и пропорционально увеличивался до максимального значения при $z \to$ SIDE_OF_SYSTEM. Максимально значение есть

$$\frac{X * ARGON\_RADIUS}{2 * MOL\_SIDE * SIGMA}$$

Тут скрыта еще одна идея - добиться реальных пропорций (выполнялось

$$\frac{SIDE\_OF\_SYSTEM}{X} = \frac{ARGON\_RADIUS}{2ndArg}$$

Добавим еще draw, но уже без 10 во втором аргументе. Получим меньшие круги поверх больших, это позволит различать молекулы при наложении.

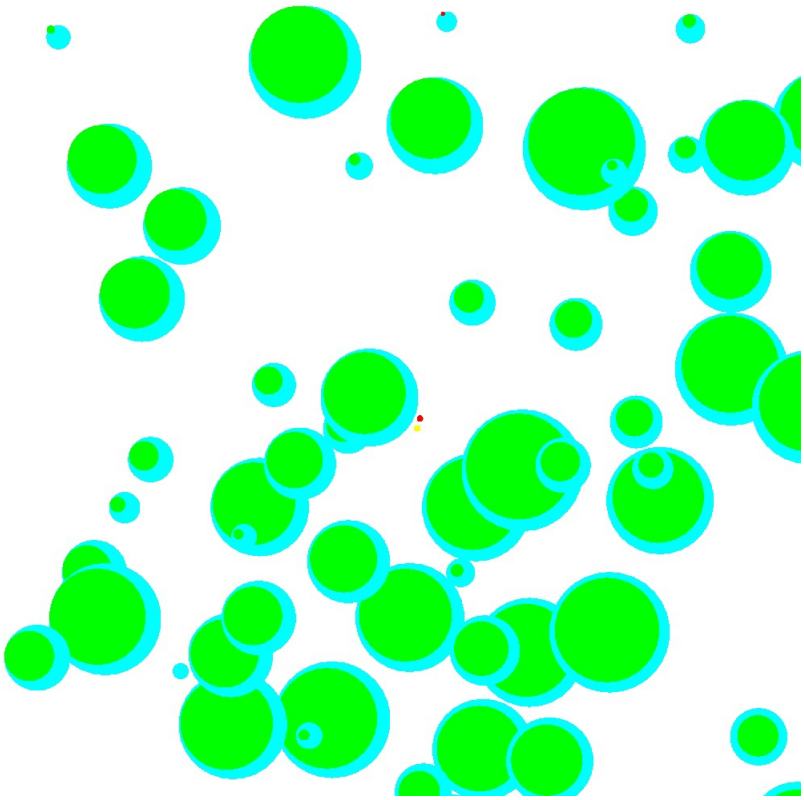

Рис. 3: Вот что получилось!

Остается одна проблема - поскольку отрисовка происходит в порядке положения в массиве - далекие молекулы могут казаться ближе, отрисовываясь первее и, соответсвенно, «на» больших. Можно добавить сортировку по z, но, на мой взгяд, это избыточно.

3.3 Начальное распределение молекул

Рассмотрим начальный момент времени. Для кооректности необходимо задать те условия, в которых могла оказаться произвольная система. Взяв нереальное расположение молекул мы не получим полезных результатов.

3.3.1 Координаты

Про координаты уже было сказано в разделе про константы. Выбранно расположение в виде куба, поскольку мы рассматриваем лишь часть - ячейку крупной системы. Это можно сравнить с приближением площади маленькими квадратами. А еще это удобно упаковать в цикл. Формируем куб:

17: Main.cpp

```
void setup_positions(vector<Molecule> &molecules)
{
    for (int i = 1; i < MOL_SIDE + 1; i++)
    {
        for (int j = 1; j < MOL_SIDE + 1; j++)
        {
            for (int k = 1; k < MOL_SIDE + 1; k++)
            {
    // Создаем и помещаем молекулу (координаты, скорость)
                Molecule to_add =
                        Molecule(
                            Vector((i + 0.5) * DIST,
                                   (j + 0.5) * DIST,
                                   (k + 0.5) * DIST),
                            get_maxwell_vector());

                molecules.push_back(to_add);
            }
        }
    }
}
```

Функция get_maxwell_vector() по запросу выдает вектор скорости.

3.3.2 Скорости

Подробнее про начальные скорости. Вышеупомянутая функция - функция распределения Максвела. См 2.2. Наша задача - построить гистограмму, создать группы скоростей. Чем больше в группе экземпляров, тем чаще встречается молекула. Устремляя количество групп к бесконечности, получаем все более точную функцию распределения.

18: Physics.cpp

```cpp
// Для начала сама функция распределения. Температуру прини
    маем нормальной.
double maxwell(double V)
{
    double m = ARGON_MASS;
    int T = 300;
    double k = 1.38 * pow(10, -23);
    double A1 = (1 / sqrt(M_PI)) * sqrt(m / (2 * k * T));
    return A1 * exp(-((m * pow(V, 2)) / (2 * k * T)));
}
```

19: Physics.cpp

```cpp
double get_maxwell_dist()
{
    double sum = 0;
    std::vector<double> vel;

    /* Для каждого значения скорости, которому соответсвует
     индекс массива, вычисляем вероятность, и записываем
     в этот индекс. Максимальное значение скорости 1000 */
    for (int i = 0; i < 1000; i++)
    {
        double prob = maxwell(i);
        vel.push_back(prob);
        sum += prob;
    }
    // Берём "случайный" процент
    double b = rand() % 100;

    int i = 0;
    double cur = 0;

    /* Ищем, чему этот процент соответствует. Чем больше
     было вероятность i-ого значения скорости, тем с
     большей вероятностью мы перешагнём и остановимся
     на нём. Либо массив кончится.                      */
    for (i = 0; i < vel.size() && b > cur * 100; i++)
    {
        cur += vel[i] / sum;
    }
    // Возвращаем либо +, либо - i.
    return (rand() % 2 == 0 ? -1 : 1) * i;
}
```

Функция get_maxwell_vector() просто возвращает вектор из трех значений функции get_maxwell_dist(). Взятая с такими скоростями система

удолетворяет распределению скоростей максвела по проекциям.

3.4 Итерация симуляции

Имея начальные значения, мы можем перейти к последубщей конфигурации. Продолжать этот процесс можно до тех пор, пока накопленная погрешность не сделает вычисления бессмысленными. Не взаимодействующие молекулы ведут себя довольно скучно - куда изначально двигались, туда и двигаются. В действительности между молекулами возникают силы, а также они могут столкнуться. Или в нашем случае вылететь из системы. Все эти отношения нужно смоделировать.

3.4.1 Расчёт сил

Мы будем использовать наиболее расрпостраненный метод нахождения сил взаимодействия молекул - потенциал Леннарда-Джонса 2.3. Константы для неё вычислялись экспериментальным метода, и уникальны для каждого вида молекул. Переписываем формулу.

20: Physics.cpp

```
// Для начала сама функция распределения. Температуру прини
    маем нормальной.
double lennard_jones(double r)
{
    return 12 * (E / SIGMA) * (pow(SIGMA / r, 13) - pow(
        SIGMA / r, 7));
}
```

Несколько больше интересно её применение. Наивно - необходимо для каждой молекулы найти ей пару, посчитать расстояние между ними, прибавить силу, забыть её, попасть на вторую молекулу, найдя ей в пару первую - снова посчитать расстояние и силу. Забегая вперёд, рассчёт расстояния выполняется не за $O(1)$, поэтому это может сыграть свою роль. Куда более успешным решением будет рассмотреть все пары. Их уже не n^2, а $\binom{n}{2}$, что всяко лучше. Вспоминая сортировку пузырьком:

21: Physics.cpp

```
for (int i = 0; i < molecules.size(); i++)
    {
        Vector force_sum = Vector(0, 0, 0);
    // начинаем не с 0, а с i+1. Так рассмотрим все пары и
        только их
        for (int k = i + 1; k < molecules.size(); k++)
        {
              // находим численное значение расстояние, и н
                  аправление взамодействия
```

```
8               pair<double, Vector> dist_dir =
                    calc_periodic_dist(molecules[i], molecules[k
                    ]);
9
10              // Функция возвращает dist = 0, если молекулы с
                    лишком далеко друг от друга.
11              if (dist_dir.first != 0)
12              {
13                  // Высчитываем силу как направление *
14                  Vector force = dist_dir.second *
                        lennard_jones(dist_dir.first);
15
16                  // Собираем для i-ой
17                  force_sum += force;
18                  // Для каждой k-ой собираем по отдельности
19                  molecules[k].force.cur += -force;
20              }
21          }
22          // То, что собрали с последующих прибавляем.
23          molecules[i].force.cur += force_sum;
24      }
```

Таким образом для i-ой молекулы мы $(size() - i)$ раз прибавляем силы с другими (в конце) и еще i сил она получает, потмоу что до этого ровно i молекул отдали ей свою. Вот, например, для 4-ех молекул. A→ B означает: сила B += сила от молекулы A. Сверху вниз - по i. Стрелки снизу означают 23 строку кода.

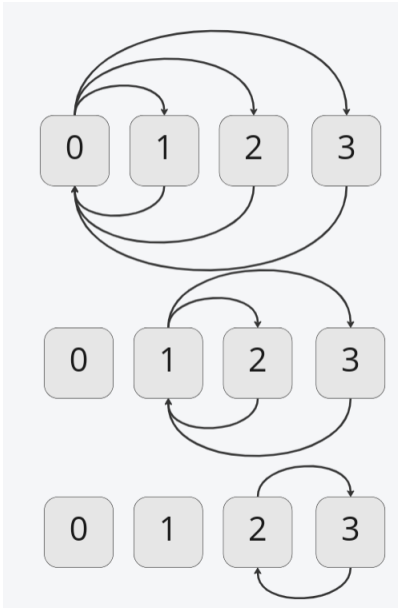

Рис. 4: Расчет сил для 4-ёх молекул

3.4.2 Получение новых положений молекул

Заполнив массив свежими значениями сил, мы готовы совершить шаг. Выполняем его одновременно для всех.

22: Main.cpp

```
for (int i = 0; i < molecules.size(); i++)
        {
            molecules[i].semiStep();
        }
```

Далее осталось лишь продолжать данные вычисления. Причиной остановки может служить как закрытие главного окна, так и наступление определенного момента времени. В таком случае понадобиться счётчик. Я использую терминацию main.

23: Main.cpp

```
// Пока окно открыто
while (visualization.main.isOpen())
    {
      // Проверить, может некоторые окна закрыли
        visualization.is_close();

        // Отрисовываем молекулы
        for (int i = 0; i < molecules.size(); i++)
        {
            draw(..);
            draw(..);
        }

      // Шаг
        for (int i = 0; i < molecules.size(); i++)
        {
            molecules[i].semiStep();
        }
        // Обновляем окна
        visualization.display();
    }
```

Теперь на этот «скелет» мы будем наслаивать мышцы: перехватывать значения и добавлять их в графики, корректировать силы до совершения шага. Но сначала разберемся с долгами.

3.5 Периодические граничные условия

Наша система ограничена. Поэтомы случай, при котором молекула достигнет границу неизбежен. Известны две модели решения этой задачи.

- Жесткие граничные условия

  Молекулы просто отскакивают от стенок. Угол отражения равен углу падения. Скорость не меняется по модулю. Простая модель.

- Периодические граничные условия

  Наша система окружена своими копиями. Молекула, вылетая из неё, пораждает свою копию, влетающую с другой. Вот картинка для двумерного случая.

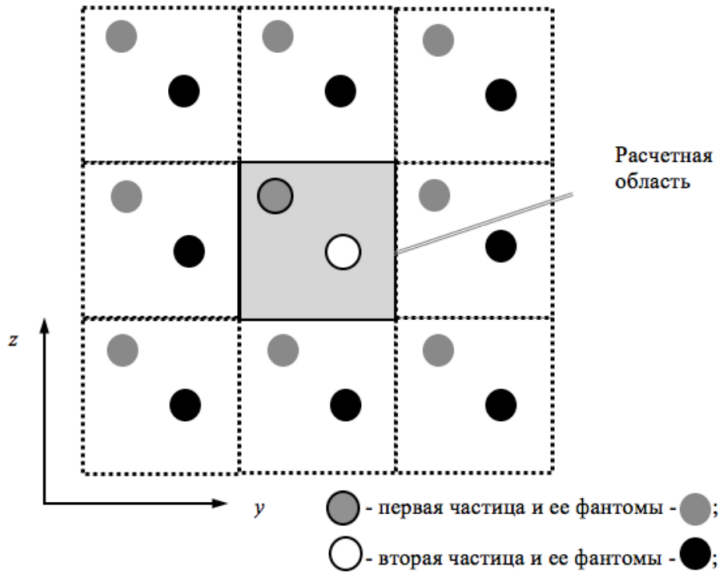

Рис. 5: Иллюстрация периодических граничных условий

Я выбрал второую модель, поскольку она, в некотором смысле, позволяет смоделировать бесконечно много частиц.

Начнём с простого. Обработаем случай вылета за границу системы.

24: Molecule.cpp

```
// Проверка и возврат в систему по проекции
void Check(double &cor)
{
  // Вылетела равносильно условию
    if (cor >= SIDE_OF_SYSTEM || cor <= 0)
    {
          // Достаточно сместиться на side, но в какую стор
             ону
        cor += (cor <= 0 ? 1 : -1) * SIDE_OF_SYSTEM;
    }
}

// Проверяем для всех проекций
void Molecule::periodic()
{
    Check(coordinates.cur.x);
    Check(coordinates.cur.y);
    Check(coordinates.cur.z);
}
```

Теперь молекулы в бесконечном отеле. Но есть некоторое противоречие. Предположим, мы рассматриваем две молекулы. Они находятся на расстоянии больше действия сил, но так, что фантом одной из них - близко ко второй. Логично было бы считать, что они взаимодействуют. Причем в противоположном направлении, нежели оригиналы. Я посчитал нужным учитывать данный случай. То есть, для того, чтобы найти расстояние между частицами, нам необходимо еще и высчитать расстояние до их копий.

Довольно просто показать, что, при данном выборе констант, если мы нашли достаточно малое расстояние между молекулой и какой-то копией второй, то дальше мы не найдем не только более влиятельного фантома, но и вовсе в радиусе действия силы. (Все копии находятся на равном расстоянии друг от друга, и это расстояние = стороне системы. Поэтому уже в случае 8 молекул (0 и 1 очевидно) мы не сможем предъявить конфигурацию, где у молекулы в радиусе действия силы лежит два и более фантомов другой частицы.) Я реализовал это следующим образом.

25: Molecule.cpp

```
// Получаем две молекулы.
pair<double, Vector> calc_periodic_dist(Molecule tarMol,
   Molecule copMol)
{ //Для удобство возьмём их точки
    Vector target = tarMol.coordinates.cur;
    Vector to_copy = copMol.coordinates.cur;

```

```
7       /* Заведем пустышку, которую будем изменять, получая из
            центральной
8          молекулы её копии (включая центральную) */
9       Vector dummy = to_copy;
10      /* По всем координатам будем смещаться на SIDE. Для
11         этого переберем все случаи. -1 - влево/вниз/глубже;
12         0 - на месте, 1 - вправо/вверх/ближе */
13      for (int n = -1; n <= 1; n++)
14      {
15          for (int m = -1; m <= 1; m++)
16          {
17              for (int k = -1; k <= 1; k++)
18              {
19                  // Смещаемся, получая из исходной копию в д
                       ругой ячейке
20                  dummy.x += m * SIDE_OF_SYSTEM;
21                  dummy.y += n * SIDE_OF_SYSTEM;
22                  dummy.z += k * SIDE_OF_SYSTEM;
23                  // Находим расстояние между ними в привычно
                       м смысле
24                  double d = distance(dummy, target);
25                  // Проверяем, не мало ли оно
26                  if (d < FORCE_RADIUS)
27                  {
28                      // Нашли одно - других не будет. + Напр
                           авление важно
29                      return pair<double, Vector>(d,
30                          Vector(target.x - dummy.x,
31                                 target.y - dummy.y,
32                                 target.z - dummy.z).
                                    normalize());
33                  }
34                  // Обновляем
35                  dummy = to_copy;
36              }
37          }
38      }
39      // Если ничего не нашли, возвращаем 0
40      return pair<double, Vector>(0, Vector(0, 0, 0));
41  }
```

Это помогает нам не только детектировать сложные ситуации, но правильно расчитывать силу, считая в таких случаях взаимодействия с фантомами, а не с оригиналами, что придавало бы силе неправильно направление. Вот пример такой ситуации

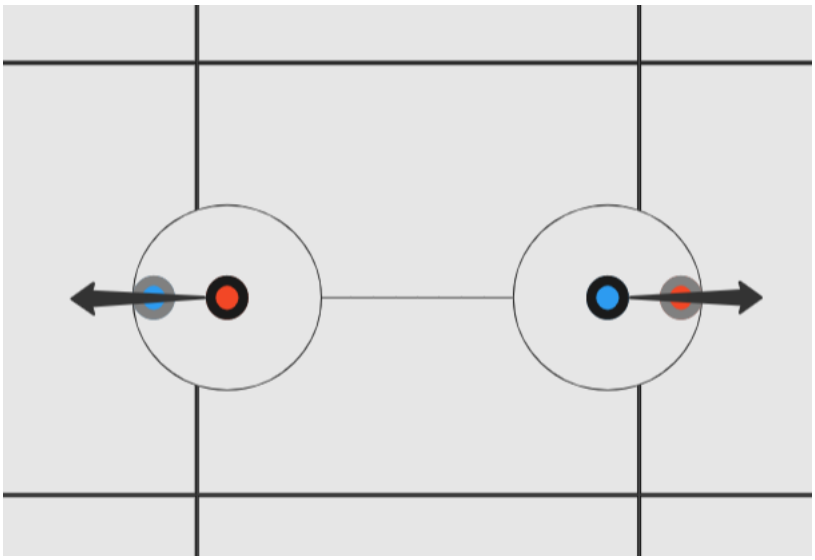

Рис. 6: Взаимодействие на границе

3.6 Модель нецентральных столкновений

TODO =(

3.7 Проверка стабильности системы

Проверим, что на текущем моменте у нас довольно небольшое расхождение. Для этого добавим график кинетической энергии.

26: Main.cpp

```
...
Graph kinetic_e = Graph(sf::Color::Blue);
...
while (...)
{
   ...
     double velocity = 0;
     for (int i = 0; i < molecules.size(); i++)
     {
          velocity += pow(molecules[i].velocity.x, 2) +
                      pow(molecules[i].velocity.y, 2);
     }
     kinetic_e.update_graph(&visualization.kinetic, velocity
        );
}
```

Для начала возьмём две молекулы, а скорости зададим только по одной координате - навстречу друг другу.

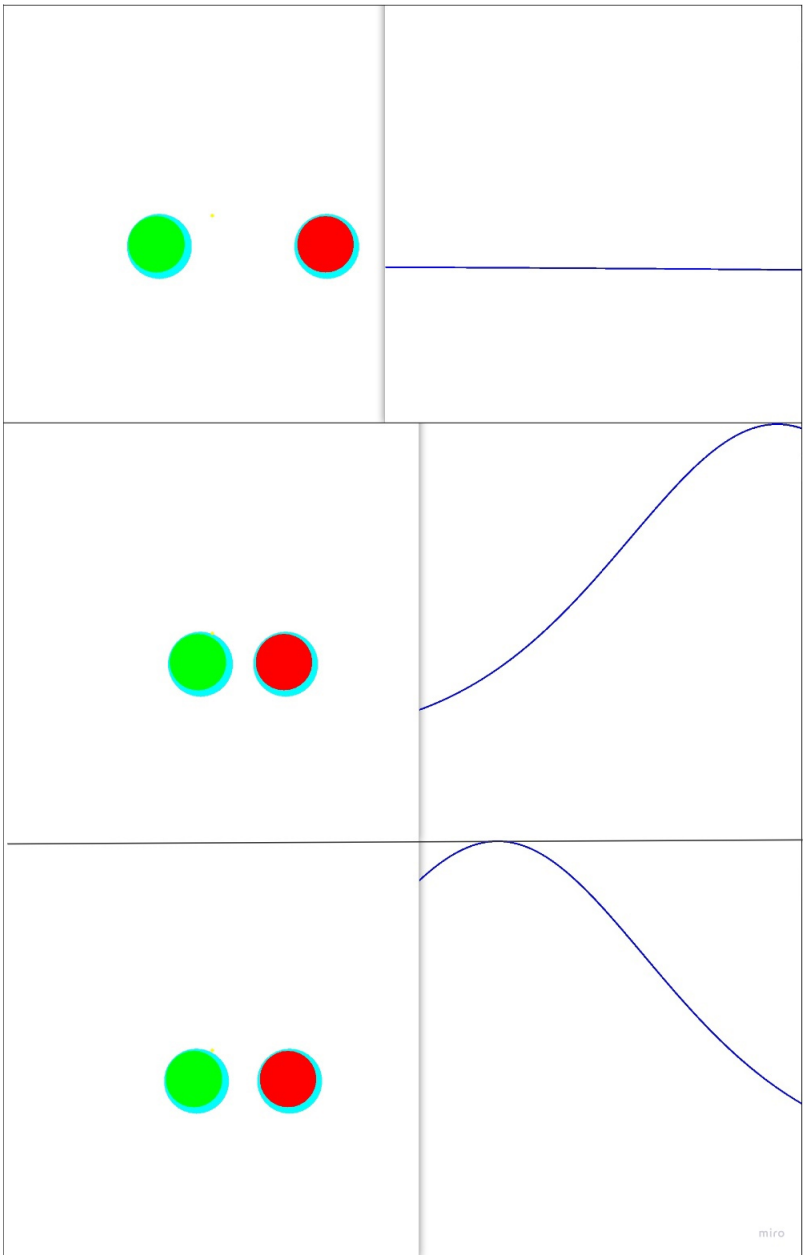

Рис. 7: Две молекулы $\Delta t = 10^{15}$

Проанализируем его. Очень хорошо видно, что побуждает изменение кинетической энергии. Это приближения молекул друг к другу, и, как следствие, влияние на них сил, ускоряющих их.

- Сначала они ускоряются - график под наклоном вниз.
- Потом они начинают отталкиваться, вплоть до скоростей, равных 0 (Вершина графика)

• График возвращается на прежние значения до столкновения.

Полезным может оказаться рассмотреть более низкую точность, увеличив δt.

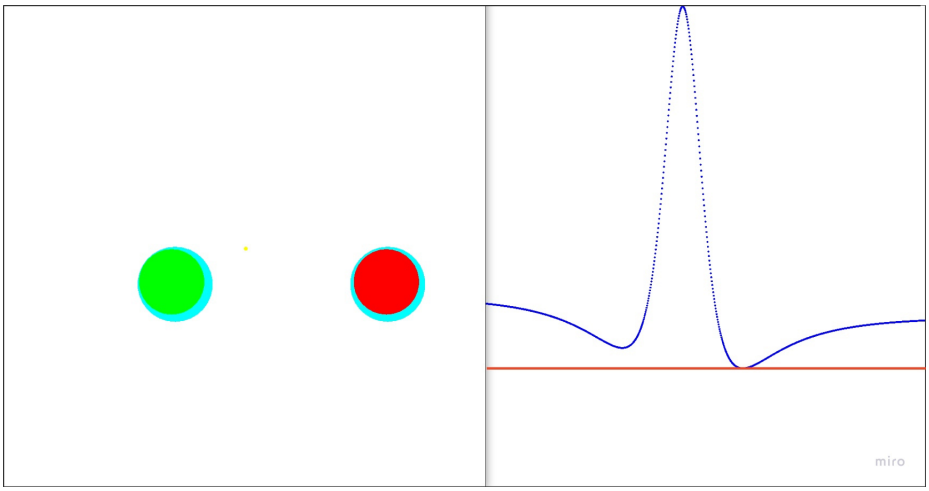

Рис. 8: Две молекулы $\Delta t = 10^{13}$

Можно видеть, что с каждым взаимодействием кинетическая энергия на очень малую величину увеличивается. При множественном взаимодействии этот эффект будет накапливаться и результаты эксперимента будут не достоверными. Поэтому с целью свести его к минимуму шаг по времени берется максимально малый.

Конечно, добавив молекул, нам будет еще сложнее проследить за каждым столкновением, но важно то, что кинетическая энергия после колебаний возвращается на прежний уровень.

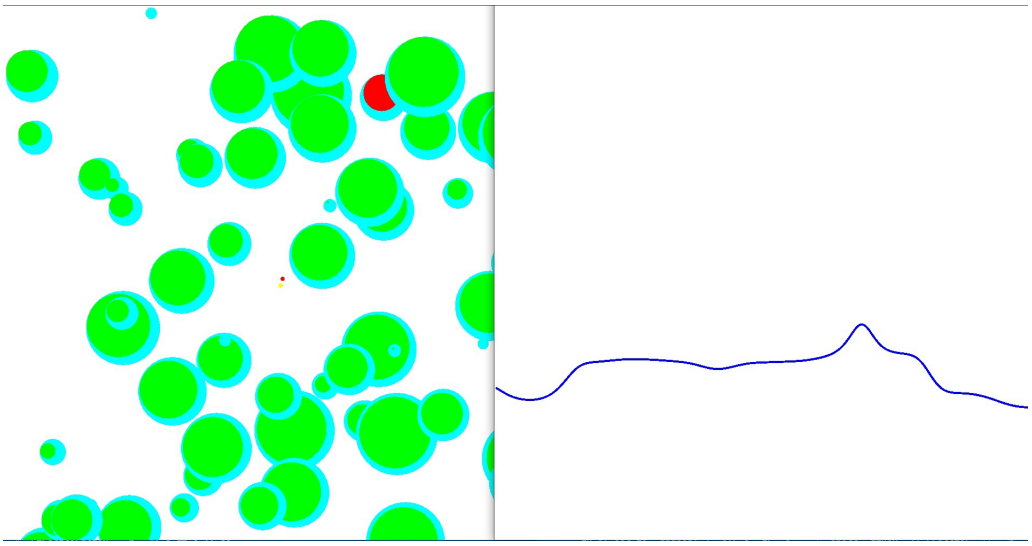

Рис. 9: 100 молекул $\Delta t = 10^{15}$

3.8 Центр инерции

Теперь мы готовы перейти к исследованию влияния моменты центра инерции.

3.8.1 Нахождение центра

Лишь взгянув на формулу 2.4, становится ясно, что центр инерции особенно легко найти в случае равновесных частиц. Однако мы впоследствии добавим тяжелую частицу, поэтому напишем функцию честно:

27: Main.cpp

```
Vector calc_inertia_center(std::vector<Molecule> molecules)
{
    Vector center = null();
    double mass = 0;
    for (int i = 0; i < molecules.size(); i++)
    {
        center += (molecules[i].coordinates.cur * molecules
            [i].mass);
        mass += molecules[i].mass;
    }
    return center / mass;
}
```

Вспомним об удобном классе Delta, позволяющий нас хранить текущее и предыдущее значение. Сентер - сущность статическая, связана с системой, поэтому изменяется после каждого шага.

28: Main.cpp

```
Delta inertion = Delta(null(), null());
while(..)
{
  ..
    inertion.prev = inertion.cur;
    inertion.cur = calc_inertia_center(molecules);

    for (int i = 0; i < molecules.size(); i++)
        {
            molecules[i].semiStep();
        }
    ..
}
```

Теперь на каждой итерации мы имеем необходимые для вычисления момента значения. Найдем градиент.

3.8.2 Момент центра инерции. Сила

29: Physics.cpp

```
1  void calc_iner_force(std::vector<Molecule> first, Delta
       iner)
2  {
3      for (int i = 0; i < first.size(); i++)
4      {   // Находим разницу моментов.
5          Vector M = first[i].force.cur * distance(iner.cur,
               first[i].coordinates.cur) -
6                      first[i].force.prev * distance(iner.prev
                           , first[i].coordinates.prev);
7          // Считаем градиент, сразу суммируем.
8          Delta coor = first[i].coordinates;
9          first[i].force.cur += Vector(M.x / (coor.prev.x -
10                                            coor.cur.x),
11                                     M.y / (coor.prev.y -
12                                            coor.cur.y),
13                                     M.z / (coor.prev.z -
14                                            coor.cur.z));
15     }
16 }
```

3.9 Исследование новой системы

4 Заключение

В результате работы над учебной практикой в течение осеннего семестра были выполнены следующие задачи:

1. Смоделирована движение броуновских частиц,
2. Вычислен момент центра инерции,
3. Добавлена тяжелая частица,
4. Исследованы на графиках:
    1) Кинетическая энергия,
    2) Вязкость,
    3) Влияние центра инерции
5. Построена визуализация движения.

Список литературы

[1] Потенциал Леннарда-Джонса

https://thesaurus.rusnano.com/wiki/article1565 http://http://tm.spbstu.ru/ https://mipt.ru/upload/medialibrary/138/b03_909_gordeev_modelirovanie-chastits-v-potentsiale.pdf

[2] Справочные материалы. Значения констант для Аргона

https://en.wikipedia.org/wiki/Argon

[3] Функция Максвелла

https://es.wikipedia.org/wiki/Distribuci%C3%B3n_de_Maxwell-Boltzmann http://fn.bmstu.ru/data-physics/library/physbook/tom2/ch5/texthtml/ch5_4.htm https://physics.spbstu.ru/userfiles/files/molec2-03.pdf

[4] Центр инерции

http://www.ablov.ru/Physics/8%20klass/0427(14).htm

[5] Молекулярная Динамика

http://komp-model.narod.ru/Komp-mod9.pdf

https://phys.vsu.ru/me/downloads/m13-178.pdf

https://portal.tpu.ru/SHARED/b/BGA/bio/masters/Tab3/5_MD.pdf

https://www.researchgate.net/publication/261081523_Molekularnaa_dinamika_na_personalnom_komputere_ucebnoe_posobie https://elar.urfu.ru/handle/10995/28065

http://test.kirensky.ru/master/articles/monogr/Book/Chapter_1_11.htm

https://www.cambridge.org/core/books/art-of-molecular-dynamics-simulation/57D40C5ECE9B7EA17C0E77E7754F5874

https://books.ifmo.ru/file/pdf/2363.pdf

[6] Периодические граничные условия

https://elcut.ru/advanced/perio1_r.htm

https://ru.frwiki.wiki/wiki/Condition_p%C3%A9riodique_aux_limites

https://studfile.net/preview/596304/page:4/

https://www.mathnet.ru/links/7676b4ca8f2c25d29232cfd6b9583876/zvmmf9995.pdf

https://mpei.ru/diss/Lists/FilesDissertations/347-%D0%94%D0%B8%D1%81%D1%81%D0%B5%D1%80%D1%82%D0%B0%D1%86%D0%B8%D1%8F.pdf

[7] Законы сохранения https://www.bsuir.by/m/12_100229_1_112767.pdf

[8] Нецентральные столкновения https://zftsh.online/articles/4801

[9] X-Macros https://habr.com/ru/post/475162/

https://temofeev.ru/info/articles/primenenie-x-macro-v-modernovom-c-kode/

https://stackoverflow.com/questions/6635851/real-world-use-of-x-macros

https://quuxplusone.github.io/blog/2021/02/01/x-macros/

[10] SFML

https://www.sfml-dev.org/