**МІНІСТЕРСТВО ОСВІТИ Й НАУКИ УКРАЇНИ**
**СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ**

Кафедра прикладної математики та моделювання складних систем

Допущено до захисту

Завідувач кафедри ПМ та МСС

_______________ Коплик І. В
(підпис)

«__» _____________________ 2020р.

**КВАЛІФІКАЦІЙНА РОБОТА**
на здобуття освітнього ступеня «магістр»

спеціальність 113 «Прикладна математика»

освітньо-професійна програма «Прикладна математика»

тема роботи «**ДОСЛІДЖЕННЯ МОЖЛИВОСТЕЙ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ МОДЕЛЮВАННЯ БАГАТОЧАСТИНКОВИХ СИСТЕМ МЕТОДОМ МОЛЕКУЛЯРНОЇ ДИНАМІКИ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ CUDA**»

**Виконавець**
студент факультету ЕлІТ
Арнаутов Олексій Ігорович ______
(прізвище, ім'я, по батькові) (підпис)

**Науковий керівник**
к. ф. – м.н., доцент
(науковий ступінь, вчене звання)

Князь Ігор Олександрович ______
(прізвище, ім'я, по батькові) (підпис)

Суми – 2020

# РЕФЕРАТ

**Кваліфікаційна робота:** 152 с., 17 рисунків, 85 формул, 2 таблиці 14 джерел.

**Мета роботи:** пошук можливостей прискорення розрахунків систем взаємодіючих частинок методом молекулярної динаміки.

**Об'єкт дослідження:** системи взаємодіючих частинок та їх комплексів.

**Предмет дослідження:** моделі взаємодіючих елементів багаточастинкових систем.

Побудовано математичні моделі взаємодіючих елементів та їх комплексів. Описано взаємодію між окремими моделями молекул; за допомогою потенціалів зв'язку та кутового потенціалу описано механізми взаємодії між моделями атомів у межах окремих моделей молекул.

Проведено серію комп'ютерних експериментів з метою визначення оптимального значення радіусу відсікання для системи гнучких куль за допомогою технології CUDA та визначено, що аналітична оцінка, наведена в літературних джерелах, є заниженою.

Ключові слова: МОЛЕКУЛЯРНА ДИНАМІКА, МОДЕЛЬ МОЛЕКУЛИ ВОДИ, ПОТЕНЦІАЛИ ВЗАЄМОДІЇ, CUDA, МІЖМОЛЕКУЛЯРНА ВЗАЄМОДІЯ ВНУТРІШНЬО МОЛЕКУЛЯРНА ВЗАЄМОДІЯ, РІВНЯННЯ РУХУ НЬЮТОНА, МЕТОД ВЕРЛЕ.

# ЗМІСТ

ВСТУП........................................................................................................ 8

1 АНАЛІТИЧНИЙ ОГЛЯД ................................................................................ 10

1.1 Найпростіша модель – частинки як пружні кулі. Потенціали взаємодії частинок................................................................................................................ 10

1.2 Ускладнення моделі – молекула з «жорсткими» зв’язками .................... 20

1.3 Наступний крок – конфігурація «гнучких» взаємозв’язків у середині молекули. Потенціали внутрішньої взаємодії..................................................... 21

1.4 Дискретизація руху частинок. Чисельний метод розв’язання рівнянь руху частинок ........................................................................................................ 25

2 ВИКОНАНІ САМОСТІЙНО ДОСЛІДНИЦЬКІ РОБОТИ........................... 27

2.1 Отримання необхідних рівнянь та виразів ............................................... 27

2.2 Перехід до умовних величин ..................................................................... 35

2.3 Обчислення модельних константних значень у безрозмірних величинах 47

3 РЕАЛІЗАЦІЯ КОМП’ЮТЕРНОГО ЕКСПЕРИМЕНТУ ............................... 54

3.1 Методика чисельного експерименту......................................................... 54

3.1.1 Загальний алгоритм проведення комп’ютерного моделювання динаміки системи моделей молекул................................................................... 54

3.1.2 Етап визначення розмірів віртуального обмежуючого контейнеру модельованої області ............................................................................................. 54

3.1.3 Етап визначення кількості моделей молекул ....................................... 55

3.1.4 Етап обчислення початкових значень координат моделей молекул ... 55

3.1.5 Етап обчислення початкових значень швидкостей моделей молекул 59

3.1.6 Етап безпосереднього моделювання динаміки системи моделей молекул води ........................................................................................ 59

3.2 Алгоритм розрахунку значення радіусу відсікання ................................ 60

3.3 Результати розрахунків.............................................................................. 62

ВИСНОВКИ..................................................................................................... 66

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ ....................................................... 67

ДОДАТОК А ЛІСТИНГ ПРОГРАМИ, ЩО ДОЗВОЛЯЄ БУДУВАТИ ДВОВИМІРНІ ГРАФІКИ ТА ЗБЕРІГАТИ ЇХ У ВИГЛЯДІ ФАЙЛІВ ЗОБРАЖЕНЬ .PNG, МОВОЮ C++. ТЕКСТ ФАЙЛУ MAIN.CPP , ЩО МІСТИТЬ ФУНКЦІЮ MAIN..................................................................... 69

ДОДАТОК Б ЛІСТИНГ ПРОГРАМИ, ЩО ДОЗВОЛЯЄ БУДУВАТИ ДВОВИМІРНІ ГРАФІКИ ТА ЗБЕРІГАТИ ЇХ У ВИГЛЯДІ ФАЙЛІВ ЗОБРАЖЕНЬ *.PNG*, МОВОЮ C++. ТЕКСТ ФАЙЛУ *USER_CHART_WRAPPER.H*, ЩО МІСТИТЬ ОПИС КЛАСУ *USER_CHART_WRAPPER*, ЗА ДОПОМОГОЮ ЯКОГО БУДУЮТЬСЯ ГРАФІКИ...................................................................................................... 79

ДОДАТОК В ЛІСТИНГ ПРОГРАМИ, ЩО ДОЗВОЛЯЄ БУДУВАТИ ДВОВИМІРНІ ГРАФІКИ ТА ЗБЕРІГАТИ ЇХ У ВИГЛЯДІ ФАЙЛІВ ЗОБРАЖЕНЬ *.PNG*, МОВОЮ C++. ТЕКСТ ФАЙЛУ *USER_CHART_WRAPPER.CPP*, ЩО МІСТИТЬ РЕАЛІЗАЦІЮ КЛАСУ *USER_CHART_WRAPPER*, ЗА ДОПОМОГОЮ ЯКОГО БУДУЮТЬСЯ ГРАФІКИ...................................................................................................... 82

ДОДАТОК Г ЛІСТИНГ ПРОГРАМИ, ЩО РОЗРАХОВУЄ ЧАСТИННІ ПОХІДНІ ДЛЯ ВИРАЗУ СИЛИ КУТОВОГО ПОТЕНЦІАЛУ МОВОЮ ПРОГРАМУВАННЯ PYTHON ........ 86

ДОДАТОК Д ЛІСТИНГ ПРОГРАМИ ДЛЯ ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ РОЗРАХУНКУ РАДІУСУ ВІДСІКАННЯ МОВОЮ ПРОГРАМУВАННЯ С++. ФАЙЛ *MAIN.CU* ........ 88

ДОДАТОК Е ЛІСТИНГ ПРОГРАМИ ДЛЯ ПРОВЕДЕННЯ МОДЕЛЮВАННЯ ДИНАМІКИ СИСТЕМИ ВЗАЄМОДІЮЧИХ ЧАСТИНОК З ПОБУДОВОЮ ГРАФІКУ ЗАЛЕЖНОСТІ ЕНЕРГІЇ ВІД ЧАСУ МОДЕЛЮВАННЯ МОВОЮ ПРОГРАМУВАННЯ С++. ФАЙЛ *MAIN.CU* ........ 90

ДОДАТОК Ж ЛІСТИНГ ОГОЛОШЕННЯ КЛАСУ «ATOM» ДЛЯ ПОБУДОВИ ОБ’ЄКТІВ МОДЕЛЕЙ АТОМІВ МОВОЮ ПРОГРАМУВАННЯ С++. ФАЙЛ *ATOM.H* ........ 91

ДОДАТОК И ЛІСТИНГ ОГОЛОШЕННЯ КЛАСУ «DYNAMIC_CHART» ДЛЯ ПОБУДОВИ ОБ’ЄКТІВ, ЩО ДОЗВОЛЯЮТЬ БУДУВАТИ ГРАФІКИ МОВОЮ ПРОГРАМУВАННЯ С++. ФАЙЛ *DYNAMIC_CHART.H* ........ 93

ДОДАТОК К ЛІСТИНГ ОГОЛОШЕННЯ КЛАСУ «MD_EXPERIMENT» ДЛЯ ПОБУДОВИ ОБ’ЄКТІВ, ЩО МІСТЯТЬ ФУНКЦІЇ ТА ДАНІ ДЛЯ ПРОВЕДЕННЯ КОМП’ЮТЕРНОГО ЕКСПЕРИМЕНТУ МОВОЮ ПРОГРАМУВАННЯ С++. ФАЙЛ *MD_EXPERIMENT.CUH* ........ 95

ДОДАТОК Л ЛІСТИНГ ІНТЕРФЕЙСУ ФУНКЦІЇ ІНІЦІАЛІЗАЦІЇ ЕКСПЕРИМЕНТУ МОВОЮ ПРОГРАМУВАННЯ С++. ФАЙЛ *MD_EXPERIMENT_INIT_1.CUH* ........ 101

ДОДАТОК М ЛІСТИНГ ОГОЛОШЕННЯ КЛАСУ «WATER_MOLECULE» ДЛЯ ПОБУДОВИ ОБ'ЄКТІВ МОДЕЛЕЙ МОЛЕКУЛИ ВОДИ МОВОЮ ПРОГРАМУВАННЯ C++. ФАЙЛ *WATER_MOLECULE.H* ................................ 102

ДОДАТОК Н ЛІСТИНГ РЕАЛІЗАЦІЇ КЛАСУ «ATOM» МОВОЮ ПРОГРАМУВАННЯ C++. ФАЙЛ *ATOM.CPP* ................................................... 105

ДОДАТОК П ЛІСТИНГ РЕАЛІЗАЦІЇ КЛАСУ «DYNAMIC_CHART» МОВОЮ ПРОГРАМУВАННЯ C++. ФАЙЛ *DYNAMIC_CHART.CPP* ............................... 106

ДОДАТОК Р ЛІСТИНГ РЕАЛІЗАЦІЇ ЧАСТИНИ ФУНКЦІЙ ДЛЯ КЛАСУ «MD_EXPERIMENT» МОВОЮ ПРОГРАМУВАННЯ C++. ФАЙЛ *MD_EQUATIONS.CPP* ......................................................................... 111

ДОДАТОК С ЛІСТИНГ РЕАЛІЗАЦІЇ КЛАСУ «MD_EXPERIMENT» МОВОЮ ПРОГРАМУВАННЯ C++. ФАЙЛ *MD_EXPERIMENT.CU* ................................ 120

ДОДАТОК Т ЛІСТИНГ РЕАЛІЗАЦІЇ АЛГОРИТМУ МОДЕЛЮВАННЯ ДИНАМІКИ СИСТЕМИ БЕЗ РАДІУСУ ВІДСІКАННЯ МОВОЮ ПРОГРАМУВАННЯ C++. ФАЙЛ *MD_EXPERIMENT_BRUTE_FORCE_ITERATION.CU* ....................................... 127

ДОДАТОК У ЛІСТИНГ РЕАЛІЗАЦІЇ АЛГОРИТМУ МОДЕЛЮВАННЯ ДИНАМІКИ СИСТЕМИ З РАДІУСОМ ВІДСІКАННЯ МОВОЮ ПРОГРАМУВАННЯ C++. ФАЙЛ *MD_EXPERIMENT_BRUTE_FORCE_ITERATION_WITH_R-CUT.CPP* ............. 135

ДОДАТОК Ф ЛІСТИНГ РЕАЛІЗАЦІЇ АЛГОРИТМУ ФУНКЦІЇ ІНІЦІАЛІЗАЦІЇ МОВОЮ ПРОГРАМУВАННЯ C++. ФАЙЛ *MD_EXPERIMENT_BRUTE_FORCE_ITERATION_WITH_R-CUT.CPP* ............. 143

ДОДАТОК Х ТАБЛИЦЯ РЕЗУЛЬТАТІВ КОМП'ЮТЕРНОГО ЕКСПЕРИМЕНТУ З ПАРАМЕТРАМИ: 100 МОДЕЛЕЙ МОЛЕКУЛ ТА РОЗМІРАМИ КОНТЕЙНЕРУ $100 \times 100 \times 100$ ..................................................... 145

ДОДАТОК Ц ТАБЛИЦЯ РЕЗУЛЬТАТІВ КОМП'ЮТЕРНОГО ЕКСПЕРИМЕНТУ З ПАРАМЕТРАМИ: 200 МОДЕЛЕЙ МОЛЕКУЛ ТА РОЗМІРАМИ КОНТЕЙНЕРУ $100 \times 100 \times 100$ ..................................................... 148

ДОДАТОК Ш ГРАФІКИ РЕЗУЛЬТАТІВ МОДЕЛЮВАННЯ СИСТЕМИ ЧАСТИНОК ЯК ПРУЖНИХ КУЛЬ З ВИКОРИСТАННЯМ ГРАФІЧНОГО ПРИСКОРЮВАЧА (ТЕХНОЛОГІЯ CUDA)........................................................ 151

# ВСТУП

**Метою** даної кваліфікаційної роботи є розрахунок оптимального значення радіусу відсікання, який часто використовується для оптимізації обчислень під час проведення комп'ютерної симуляції динаміки системи багатьох взаємодіючих частинок.

**Актуальність** теми полягає в тому, що моделювання методами молекулярної динаміки є розповсюдженим підходом для перевірки теоретичних та експериментальних результатів. Через постійно наростаючу складність досліджуваних об'єктів існує необхідність у розробці методів, які дозволять зменшити загальну обчислювальну складність проведення моделювання. Визначаються значення радіусу відсікання, сутність котрого полягає в тому, що він дозволяє зменшити кількість обчислень шляхом відкидання таких взаємодії, які вважаються відсутніми (тобто сила зв'язку між елементами системи близька до нуля).

У даній роботі було визначено, що для системи взаємодіючих частинок, яка являє собою множину куль оцінка радіусу відсікання, що пропонується в літературних джерелах ($2\sigma$, де $\sigma$ це рівноважна відстань) є недостатньою, а слід спиратися на діапазон значень $[4\sigma;6\sigma]$. Також у даній роботі побудовано варіант системи взаємодіючих частинок, що складається з моделей молекули води (тобто множина куль, які у свою чергу складають невеликі структури) і відповідно визначено всі необхідні рівняння для руху системи молекул. За час, що виділено на написання кваліфікаційної роботи було обрано чисельний метод для розрахунку координат та швидкостей частинок, визначено вид рівняння руху Ньютона з урахуванням двох типів зв'язку, що передбачені моделлю молекули: міжмолекулярного – введено потенціали взаємодії Леннард-Джонса та Кулона, а також внутрішньо молекулярного – введено потенціали: зв'язку (*англійською* ***bond potential***), який враховує наявність парного зв'язку в межах однієї молекули та потенціал кута (*англійською* ***angle potential***), який дозволяє ввести механізм

врахування додаткової взаємодії між моделями трьох атомів в межах моделі молекули, що утворюють кут попарними зв'язками. Також було визначено співвідношення для отримання безрозмірних варіантів рівнянь руху моделей молекул та розраховано всі необхідні значення.

Розроблено комплекс програм, які дозволять провести комп'ютерні експерименти для розрахунку значення радіусу відсікання з використанням системи взаємодіючих моделей молекул води.

Проведено серію комп'ютерних експериментів для визначення конкретних значень параметрів, за яких значення енергії системи не має швидкого росту через накопичення помилки комп'ютерних обчислень та використання чисельного методу для розрахунку координат та швидкостей.

Проведено комп'ютерні експерименти для визначення значення радіусу відсікання за алгоритмом, що запропоновано автором роботи: визначити середнє значення енергії системи на одну модель атому, просумувати дані значення за деякий модельний проміжок часу та усереднити за кількістю кроків за часом, що були зроблені на протязі цього часу моделювання. Потім обчислити такі ж «контрольні» значення для кожного значення радіусу відсікання з деякого діапазону. У кінці обрати оптимальним те значення радіусу відсікання, якому буде відповідати максимально близьке значення «контрольного» параметру.

Шляхом аналізу результатів комп'ютерних експериментів визначено, що оцінка радіусу відсікання, яка пропонується в літературних джерелах є дуже заниженою у випадку моделювання системи молекул, що запропонована в даній роботі. Можливою причиною такого результату є наявність у моделі сили Кулона, яка відноситься до сил «довгого радіусу дії», тобто значення сили Кулона можна вважати рівною 0 тільки на значних відстанях, що перевищують оцінку з літературних джерел ($2\sigma$, де $\sigma$ – рівноважна відстань).

# 1 АНАЛІТИЧНИЙ ОГЛЯД

Сучасні технології дозволяють реалізовувати комп'ютерні експерименти різної складності. Одним з багатьох можливих напрямків, у яких може бути використано комп'ютерну техніку для дослідження різних явищ та закономірностей, є вивчення навколишніх об'єктів засобами молекулярної динаміки. [1,2]

Будь-який об'єкт у нашому світі складається зі структур різного степеню малості. Молекулярна динаміка (або більш коротко МД) у свою чергу дозволяє зазирнути на рівень, що недоступний неозброєному оку. Комбінація різних методів та підходів, що відносяться до МД та сучасної обчислювальної техніки дозволяють з високою точністю відтворювати та підтверджувати різні теоретичні й лабораторні дослідження, а також моделювати нові структури зі наперед заданими властивостями та характеристиками. Також безсумнівною перевагою та особливістю використання обчислювальної техніки в якості віртуальних лабораторій є факт можливості проведення різних симуляцій за майже будь-яких умов включно з, наприклад, дуже великими або ж малими значеннями температури або ж тиску без будь-якої шкоди для експериментаторів. [1,3]

## 1.1 Найпростіша модель – частинки як пружні кулі. Потенціали взаємодії частинок

Історично склалося, що однією з перших моделей, за допомогою якої вивчали властивості речовини, була модель системи, що складалася (у двовимірному випадку) з набору взаємодіючих дисків та з набору куль або ж сфер (у тривимірному випадку відповідно). У межах цієї моделі рух елементів системи описується за допомогою другого закону Ньютона (1.1):

$$F = m \cdot a, \tag{1.1}$$

де $F$ – сума сил що діють на частинку, Н;

$m$ – величина маси частинки, на яку діють сили, кг;

$a$ – в величина прискорення, з яким рухається частинка, $\text{м/с}^2$.

А їх взаємозв’язок між собою виражається за допомогою так званих потенціалів взаємодії. [2,4,5]

***Парний потенціал Леннард-Джонса.*** Саме цю залежність було використано під час моделювання властивостей газу Аргону. Даному потенціалу відповідає наступна формула (1.2):

$$U(r_{i,j}) = 4\varepsilon_{i,j}\left[\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{12} - \left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{6}\right], \quad (1.2)$$

де $\varepsilon_{i,j}$ – це енергія зв’язку між частинками з номерами *i* та *j* , Дж;

$\sigma_{i,j}$ – рівноважна відстань між кожною парою частинок системи (тобто потенціальна енергія між частинками рівна нулю $\left(U(\sigma_{i,j}) = 0\right)$), м;

$r_{i,j}$ – відстань між парою частинок, м. [2,4–6]

А також відповідна сила (1.3):

$$F(r_{i,j}) = 24\frac{\varepsilon_{i,j}}{\sigma_{i,j}}\left[2\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{13} - \left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{7}\right], \quad (1.3)$$

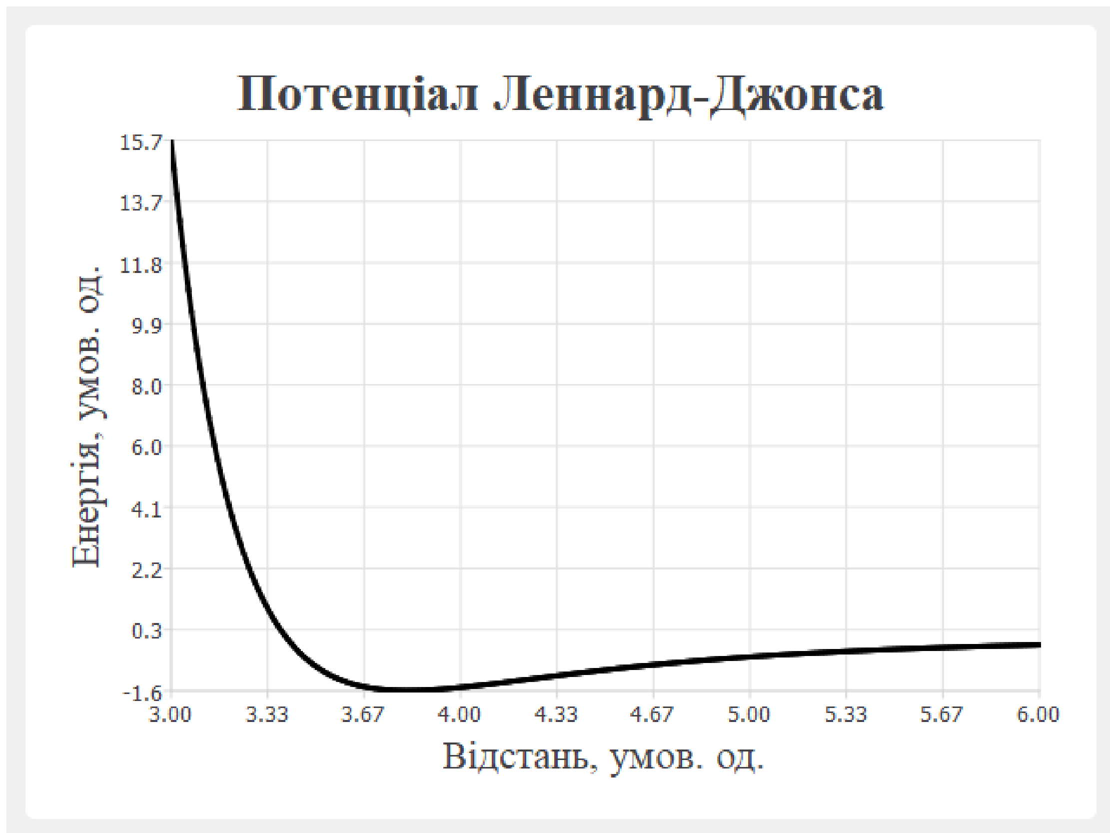
де $F$ – сила, що діє на частинку, Н;

$r_{i,j}$ – відстань між частинками з номерами *i* та *j*, м;

$\varepsilon_{i,j}$ – енергія зв’язку між частинками, Дж;

$\sigma_{i,j}$ – рівноважна відстань між кожною парою частинок, м. [5–9]

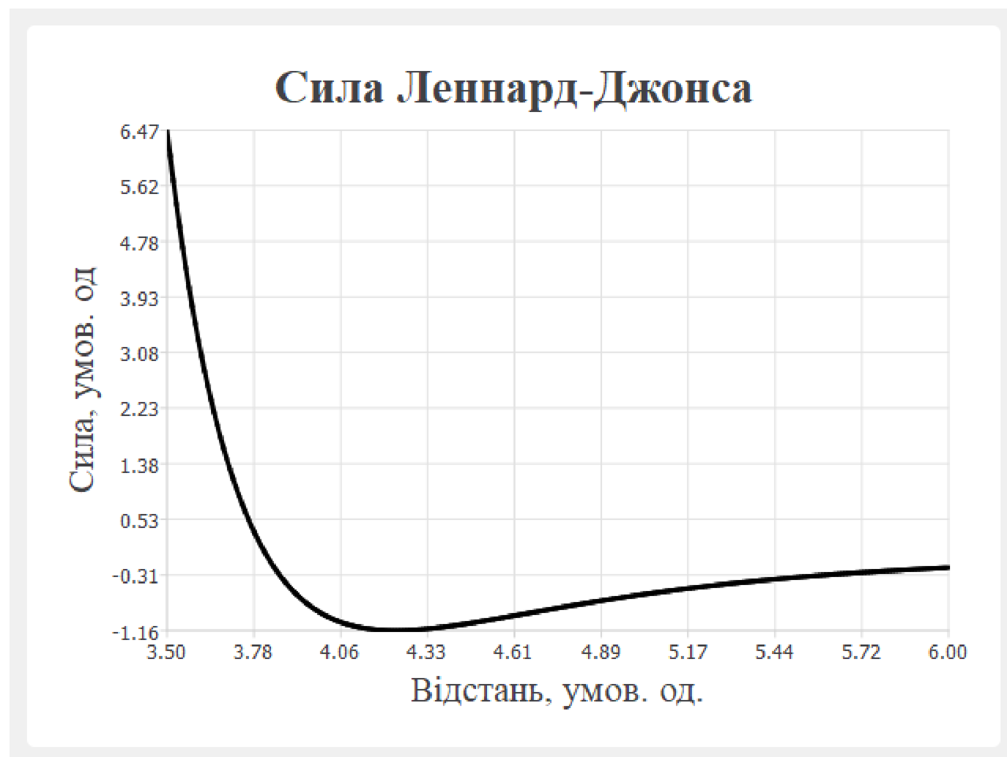
Однією з основних причин частого використання для моделювання систем взаємодіючих частинок саме потенціалу Леннард-Джонса є той факт, що в залежність енергії від відстані не входять вирази, що є відносно складними з точки зору комп’ютерних обчислень (наприклад, логарифм або тригонометричні функції). Можна побачити наочно форму залежності енергії від відстані відповідно до рівняння (1.2) на рисунку 1.1.




*Дану залежність побудовано на основі наступних параметрів (усі значення в умовних одиницях): значення **відстані** взято з діапазону:* $r_{i,j} \in [3;6]$ *з кроком* $0.01$*;* ***рівноважна відстань*** $\sigma_{i,j} = 3.4$*;* ***енергія зв'язку*** $\varepsilon_{i,j} = 1.65$. *Значення рівноважної відстані та енергії зв'язку отримано шляхом відкидання дуже малих множників у значень реальних параметрів газу Аргону:* $\left(\varepsilon_{Аргон} = 1.65 \cdot 10^{-21},\ \text{Дж}\right)$ *та* $\left(\sigma_{Аргон} = 3.4 \cdot 10^{-10},\ \text{м}\right)$. *Діапазон відстаней обрано спеціально таким чином, щоб можна було наочно побачити «потенціальну яму» на графіку.*

*Рисунок1.1 – Графік залежності потенціальної енергії від відстані за виразом Леннард-Джонса*

А також на рисунку 1.2 можна побачити залежність сили від відстані відповідно до рівняння (1.3).




*Дана залежність побудована за допомогою наступних параметрів (значення в умовних одиницях): значення* ***відстані*** *взято з діапазону:* $r_{i,j} \in [3.5;6]$ *з кроком* $0.01$*;* ***рівноважна відстань*** $\sigma_{i,j} = 3.4$*;* ***енергія зв'язку*** $\varepsilon_{i,j} = 1.65$*. Значення рівноважної відстані та енергії зв'язку отримано шляхом відкидання дуже малих множників у значень реальних параметрів газу Аргону:* $\left(\varepsilon_{Аргон} = 1.65 \cdot 10^{-21},\ \text{Дж}\right)$ *та* $\left(\sigma_{Аргон} = 3.4 \cdot 10^{-10},\ \text{м}\right)$*. Діапазон значень відстані було обрано спеціально таким чином, щоб можна було наочно побачити «яму сили» на графіку.*

*Рисунок1.2 – Графік залежності сили від відстані за потенціалом Леннард-Джонса*

***Потенціал Кулона.*** У випадку присутності електричних зарядів у моделі до рівнянь, що описують взаємодію між частинками, додається наступна залежність (1.4):

$$U\left(r_{i,j}\right) = k\frac{q_i \cdot q_j}{r_{i,j}}, \qquad (1.4)$$

де $U$ – значення потенціальної енергії взаємодії між зарядженими частинками, Дж;

$r_{i,j}$ – значення відстані між парою частинок з номерами *i* та *j*, м;

$k$ – коефіцієнт пропорційності, значення якого рівне: $9 \cdot 10^9$, $\text{Н} \cdot \text{м}^2 / \text{Кл}^2$;

$q_i$ – значення заряду частинки з номером *i*, Кл. [5,9–11]

А також відповідний вираз для сили (1.5):

$$F\left(r_{i,j}\right) = k\frac{q_i \cdot q_j}{r_{i,j}^2}, \tag{1.5}$$

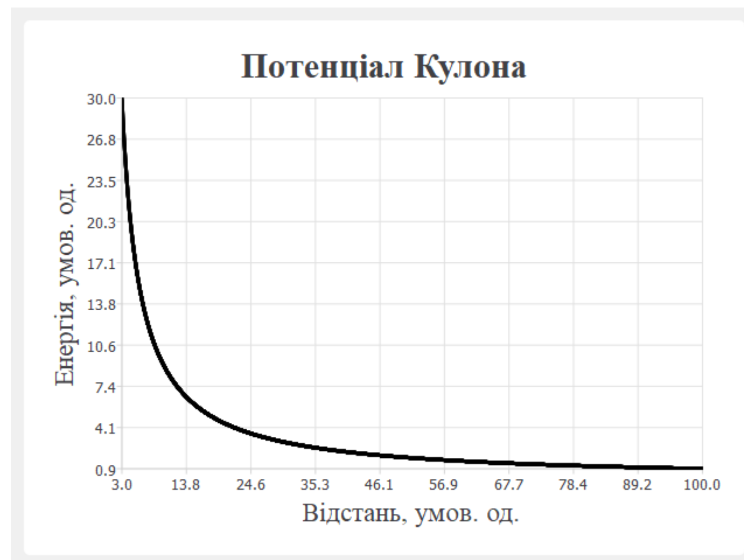
де $F$ – значення сили взаємодії між зарядженими частинками, Н;

$r_{i,j}$ – відстань між парою частинок з номерами *i* та *j*, м;

$k$ – коефіцієнт пропорційності, значення якого рівне: $9 \cdot 10^9$, $\text{Н} \cdot \text{м}^2 / \text{Кл}^2$;

$q_i$ – значення заряду частинки з номером *i*, Кл. [5,6,9,10]

Форму залежності потенціалу Кулона від відстані між частинками можна побачити, якщо поглянути на рисунок 1.3 нижче.

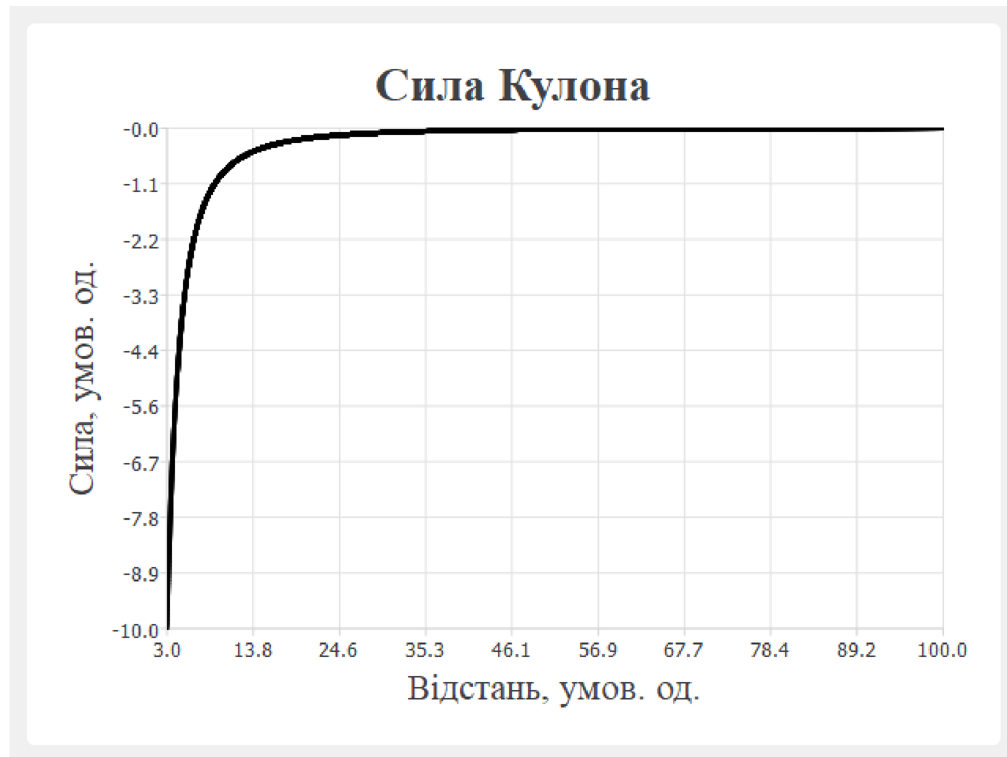



*Даний графік побудовано за допомогою наступних параметрів (значення використані в умовних одиницях):* ***відстань*** *між парою частинок:* $r_{i,j} \in [3;100]$, *з кроком* $0.01$*; значення* ***заряду*** *частинки з номером* ***i****:* $q_i = +1 \cdot 10^{-4}$*; значення*

***заряду** частинки з номером **j**:* $q_i = -1 \cdot 10^{-4}$*; коефіцієнт* ***k<sub>i,j</sub>****:* $k_{i,j} = 9 \cdot 10^{9}$*. Діапазон відстаней обрано довільним чином з урахуванням умови:* $r_{i,j} \geq 0$*.*

*Рисунок1.3 – Графік залежності потенціальної енергії від відстані за виразом потенціалу Кулона*

А також відповідна форма залежності сили Кулона від відстані на рисунку 1.4.




*Даний графік побудовано за наступних параметрів (значення використані в умовних одиницях):* ***відстань*** *між парою частинок:* $r_{i,j} \in [3;100]$*, з кроком* $0.01$ *; значення* ***заряду*** *частинки з номером* ***i****:* $q_i = +1 \cdot 10^{-4}$*; значення* ***заряду*** *частинки з номером* ***j****:* $q_i = -1 \cdot 10^{-4}$*; коефіцієнт* ***k<sub>i,j</sub>****:* $k_{i,j} = 9 \cdot 10^{9}$*. Діапазон відстаней обрано довільним чином з урахуванням умови:* $r_{i,j} \geq 0$*.*

*Рисунок1.4 – Графік залежності сили від відстані за виразом сили Кулона*

***Гармонічний потенціал.*** Даний потенціал найчастіше використовуються для моделювання динаміки одноатомних газів. А також його використовують для опису внутрішніх молекулярних взаємозв’язків, якщо моделюється речовина, що складається з молекул, які в свою чергу містять у собі деяку структуру з декількох атомів. Потенціальна залежність має вигляд (1.6):

$$U\left(r_{i,j}\right)=\frac{1}{2}k_{i,j}\left(r_{i,j}-r_{i,j}^{0}\right)^{2}, \qquad (1.6)$$

де $U$ – значення потенціальної енергії між взаємодіючими частинками, Дж;

$r_{i,j}$ – відстань між парою взаємодіючих частинок з номерами *i* та *j*, м;

$k_{i,j}$ – деякий коефіцієнт пружності зв'язку, кг/с$^2$;

$r_{i,j}^{0}$ – рівноважне значення відстані $\left(U\left(r_{i,j}^{0}\right)=0\right)$, м. [9–11]

Відповідний вираз для обчислення сили зв'язку має наступну форму (1.7):

$$F\left(r_{i,j}\right)=-k_{i,j}\left(r_{i,j}-r_{i,j}^{0}\right), \qquad (1.7)$$

де $F$ – сила взаємодії між парою частинок, Н;

$r_{i,j}$ – значення відстані між парою частинок з номерами *i* та *j*, м;

$k_{i,j}$ – деякий коефіцієнт пружності зв'язку, кг/с$^2$;

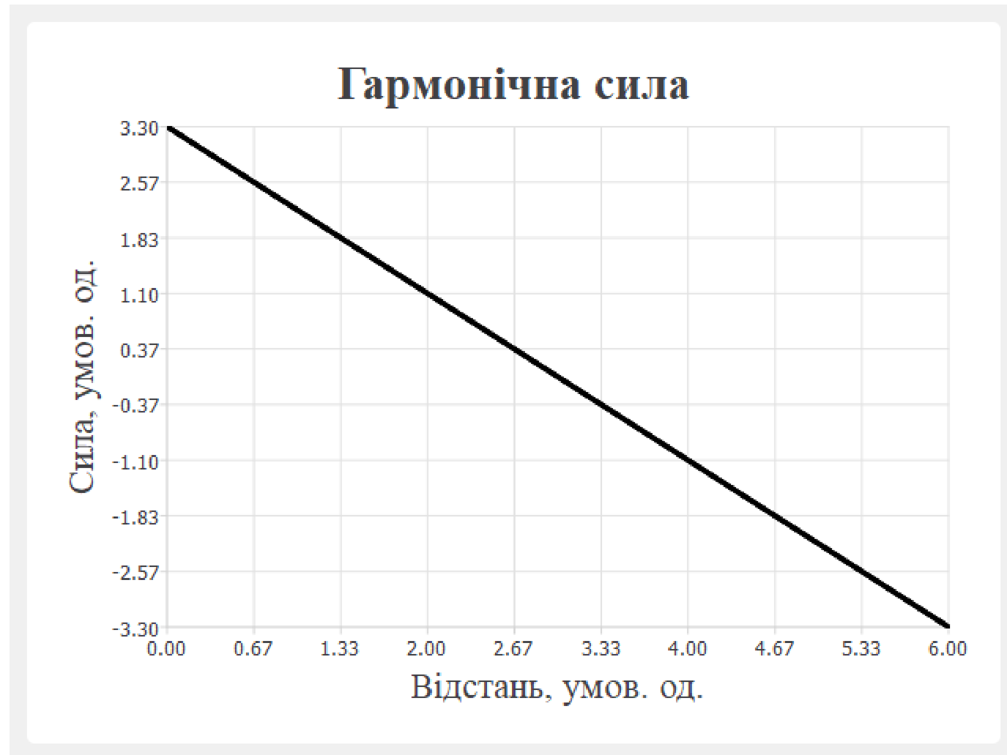
$r_{i,j}^{0}$ – рівноважне значення відстані $\left(F\left(r_{i,j}^{0}\right)=0\right)$, м. [9–11]

Форму залежності енергії від відстані зображено на рисунку 1.5.

*Даний графік побудовано за наступних параметрів (значення використані в умовних одиницях): **відстань** між парою частинок: $r_{i,j}\in\left[0;6\right]$, з кроком $0.01$; значення **рівноважної відстані** між парою частинок: $r_{i,j}^{0}=3$; значення **коефіцієнту пружності зв'язку** між частинками: $k_{i,j}=1.1$. Діапазон відстаней обрано довільним чином з урахуванням умови: $r_{i,j}\geq 0$.*

*Рисунок1.5 – Графік залежності потенціальної енергії від відстані відповідно до виразу пружного гармонічного потенціалу*

Залежність значення сили від відстані (відповідно до рівняння (1.7)) можна побачити на рисунку 1.6




*Графік побудовано за наступних параметрів (значення використані в умовних одиницях):* ***відстань*** *між парою частинок:* $r_{i,j} \in [0;6]$, *з кроком* $0.01$; *значення* ***рівноважної відстані*** *між парою частинок:* $r^0_{i,j} = 3$; *значення* ***коефіцієнту пружності зв'язку*** *між частинками:* $k_{i,j} = 1.1$. *Діапазон відстаней обрано довільним чином з урахуванням умови:* $r_{i,j} \geq 0$.

*Рисунок1.6 – Графік залежності сили від відстані відповідно до виразу пружного гармонічного потенціалу*

***Потенціал Мі.*** Даний потенціал є близьким до *потенціалу Леннард-Джонса*, але на відміну від потенціалу Леннард-Джонса має чотири параметри. Більша кількість параметрів дозволяє більш детально налаштовувати взаємодію між частинками в межах моделі. Даний потенціал має наступну форму у вигляді математичного виразу (1.8):

$$U\left(r_{i,j}\right) = \frac{\varepsilon_{i,j}}{n-m}\left[m\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{n} - n\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{m}\right], \tag{1.8}$$

де $U$ – потенціальна енергія між парою взаємодіючих частинок, Дж;

$r_{i,j}$ – відстань між парою взаємодіючих частинок, м;

$\varepsilon_{i,j}$ – енергія зв’язку між частинками, Дж;

$n$ – деякий параметр, умовні одиниці;

$m$ – деякий параметр, умовні одиниці;

$\sigma_{i,j}$ – рівноважна відстань між парою частинок з номерами *i* та *j*, м. [2]

Відповідний вираз для сили має вигляд (1.9):

$$F\left(r_{i,j}\right)=\frac{n\cdot m\cdot\varepsilon_{i,j}}{\left(n-m\right)\cdot\sigma_{i,j}}\left[\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{n+1}-\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{m+1}\right], \qquad (1.9)$$

де $F$ – сила взаємодії між парою частинок, Н;

$r_{i,j}$ – відстань між частинками, що взаємодіють, з номерами *i* та *j*, м;

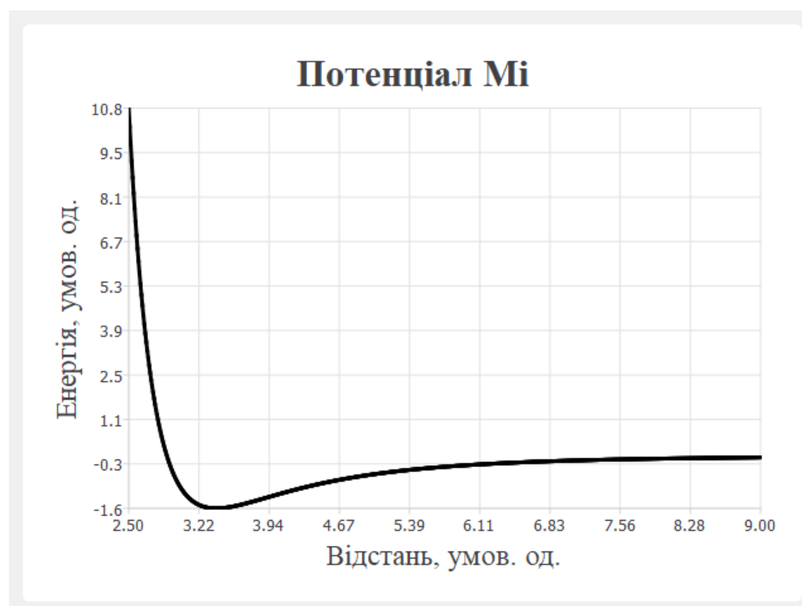
$n$ – деякий додатковий параметр, умовні одиниці;

$m$ – деякий додатковий параметр, умовні одиниці;

$\varepsilon_{i,j}$ – енергія зв’язку між частинками, Дж;

$\sigma_{i,j}$ – рівноважна відстань між парою частинок з номерами *i* та *j*, м. [2]

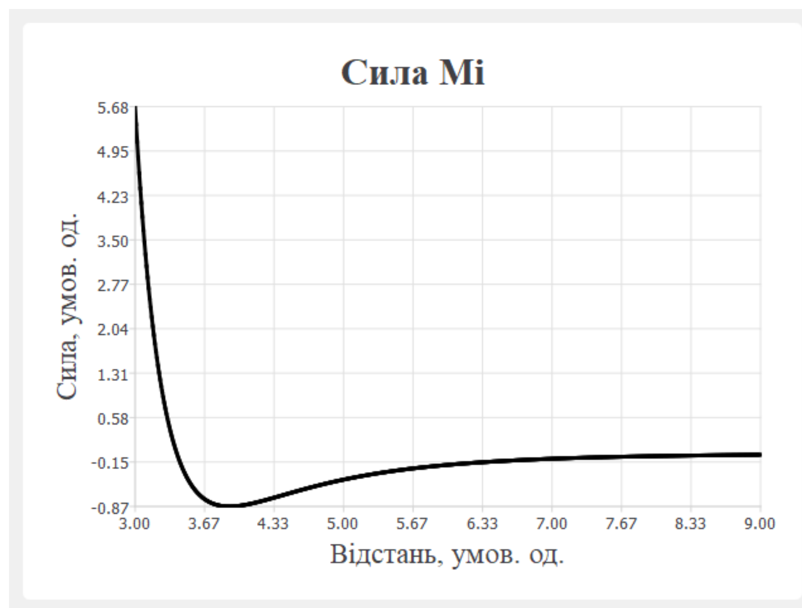
Відповідне графічне представлення форм залежності потенціальної енергії від відстані зображено на рисунку 1.7.



*Графік залежності побудовано на основі таких параметрів:* ***відстань*** *в інтервалі:* $r_{i,j} \in [2.5;9]$*, з кроком* $0.01$*;* ***рівноважна відстань*** $\sigma_{i,j} = 3.4$*;* ***енергія зв'язку*** $\varepsilon_{i,j} = 1.65$*; параметри* ***n*** *та* ***m*** *обрано довільним чином:* $n = 9$, $m = 4$. *Значення рівноважної відстані та енергії зв'язку отримано шляхом відкидання дуже малих множників у значень реальних параметрів газу Аргону:* $\left(\varepsilon_{Аргон} = 1.65 \cdot 10^{-21}, \text{ Дж}\right)$ *та* $\left(\sigma_{Аргон} = 3.4 \cdot 10^{-10}, \text{ м}\right)$. *Діапазон значень відстані було обрано спеціально таким чином, щоб можна було наочно побачити «потенціальну яму» на графіку.*

*Рисунок1.7 – Графік залежності потенціальної енергії від відстані відповідно до виразу потенціалу Мі*

Вид залежності значення сили від відстані (відповідно до формули (1.9)) подано на рисунку 1.8.




*Графік побудовано на основі таких параметрів:* ***відстань*** *в інтервалі:* $r_{i,j} \in [3;9]$*, з кроком* $0.01$*;* ***рівноважна відстань*** $\sigma_{i,j} = 3.4$*;* ***енергія зв'язку*** $\varepsilon_{i,j} = 1.65$*; параметри* ***n*** *та* ***m*** *обрано довільним чином:* $n = 9$, $m = 4$. *Значення рівноважної відстані та енергії зв'язку отримано шляхом відкидання дуже малих множників у значень реальних параметрів газу Аргону:* $\left(\varepsilon_{Аргон} = 1.65 \cdot 10^{-21}, \text{ Дж}\right)$ *та* $\left(\sigma_{Аргон} = 3.4 \cdot 10^{-10}, \text{ м}\right)$. *Діапазон значень відстані*

*було обрано спеціально таким чином, щоб можна було наочно побачити «яму сили» на графіку.*

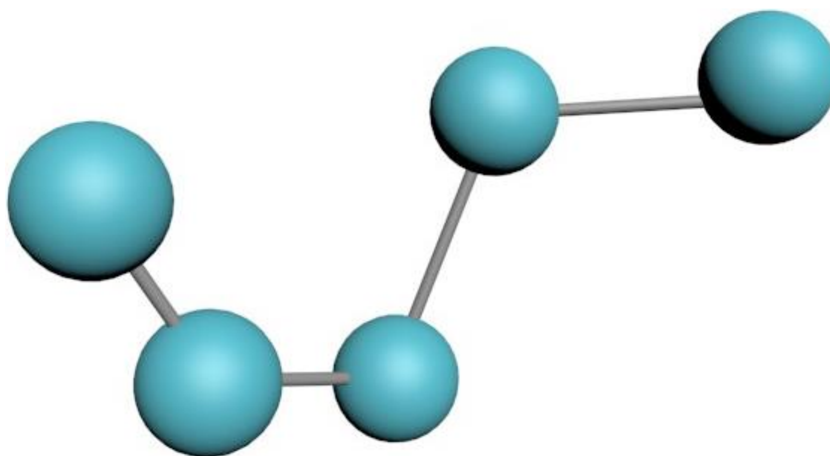
*Рисунок1.8 – Графік залежності значення сили від значення відстані відповідно до виразу потенціалу Мі*

Недоліком даної моделі є її обмеженість, очевидно, що за допомогою пружних куль можна досліджувати властивості лише тих речовин, що являються одноатомними. Результати моделювання можна з використанням CUDA можна побачити в розділі ДОДАТОК Ш.

## 1.2 Ускладнення моделі – молекула з «жорсткими» зв’язками

У попередньому підрозділі йшла мова про найпростішу модель речовини – набір «гнучких» сфер. Взаємодія кожної пари частинок такої моделі описується лише за допомогою парних потенціалів взаємодії, а рух у просторі складається лише з поступальної лінійної компоненти.

Наступним можливим ускладненням моделі для розширення можливостей симуляції різних речовин є формування молекули як набору гнучких сфер, що з’єднуються деяким чином. Новим припущенням для такої моделі є факт формування зв’язку між окремими «атомами» у межах кожної моделі молекули, а також збереження відстаней в межах кожної моделі молекули між атомами, які мають зв’язок, незмінними на протязі всього часу моделювання. Можливий варіант представлення такої моделі в тривимірному віртуальному просторі можна побачити, якщо поглянути на рисунок 1.9 нижче.



*На даному зображенні наведено приклад конфігурації тривимірної моделі молекули з п’яти «атомів» (кулі) та зв’язку між парами (лінії між кулями).*

*Рисунок1.9 – Приклад тривимірної віртуальної моделі з жорсткими зв’язками*

У випадку використання такої моделі рух модельованих частинок речовини потрібно представляти як комбінацію поступального руху та обертального руху. У якості можливого рішення можна представити рух кожної молекули як поступальний рух центру мас кожної молекули та обертальний рух «атомів» навколо центра мас кожної окремої моделі молекули.

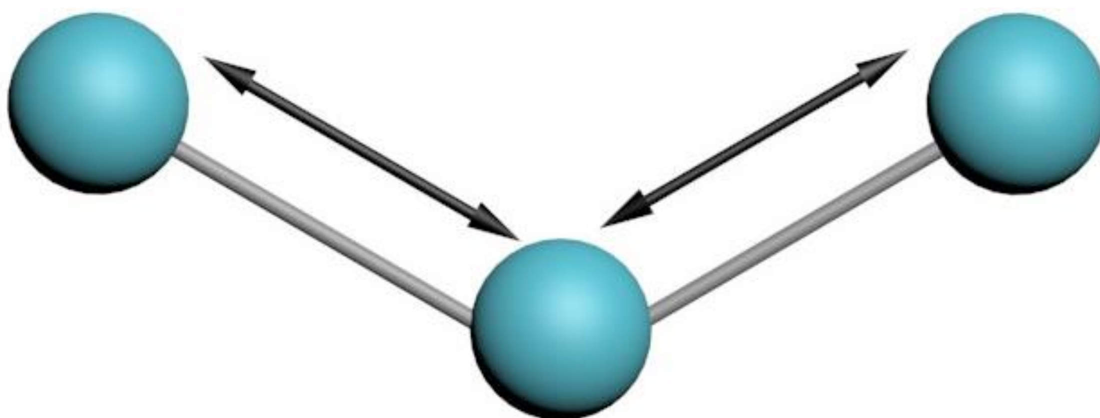
Одним з недоліків даної моделі є відсутність можливості опису взаємодії речовин, які можуть вступати в хімічну реакцію та утворювати нові сполуки в своєї наслідок взаємодії. Причина такого недоліку криється в основному припущенні моделі: незмінні «жорсткі» зв’язки між парами атомів в межах кожної молекули.

## 1.3 Наступний крок – конфігурація «гнучких» взаємозв’язків у середині молекули. Потенціали внутрішньої взаємодії

У системах взаємодіючих молекул можна виділити два окремі типи взаємодії. Перший тип – це взаємодія між самими молекулами, а другий – це вплив кожного окремого атому на всі сусідні в межах молекули. Також слід відмітити, що зв’язки, якими поєднуються атоми всередині молекули відрізняються за силою від тих типів зв’язку, що діють усередині молекул.

У загальному випадку внутрішньо молекулярні зв'язки описуються за допомогою трьох різних функцій потенціальної енергії. У літературних джерелах ці потенціали мають наступні назви:

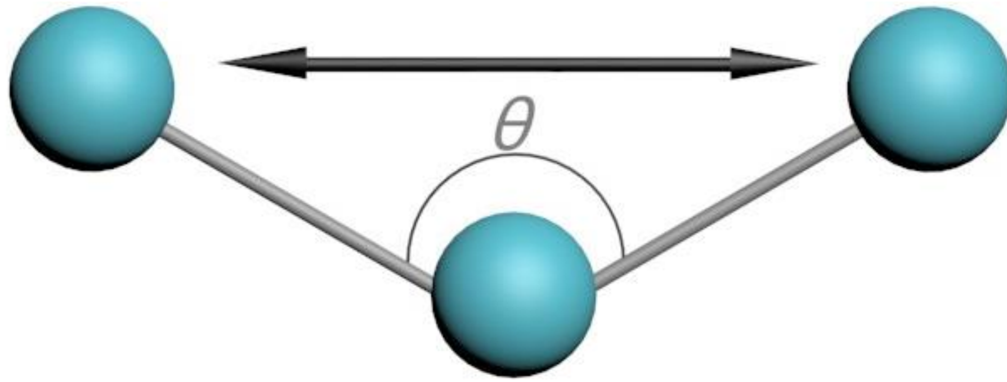
1. *Англійською* ***the bond potential*** (або приблизний переклад автора: *зв'язковий потенціал*). Цією залежністю описується механізм взаємодії між тими парами атомів у межах молекули, між якими існує сильний зв'язок див. рисунок 1.10. [9,11]



*Кулі відповідають атомам; лінії між кулями – зв'язкам між ними; стрілки вказують, між якими саме «атомами» буде відбувається взаємодія.*

*Рисунок1.10 – Схематичне зображення до пояснення дії зв'язкового (bond) потенціалу*

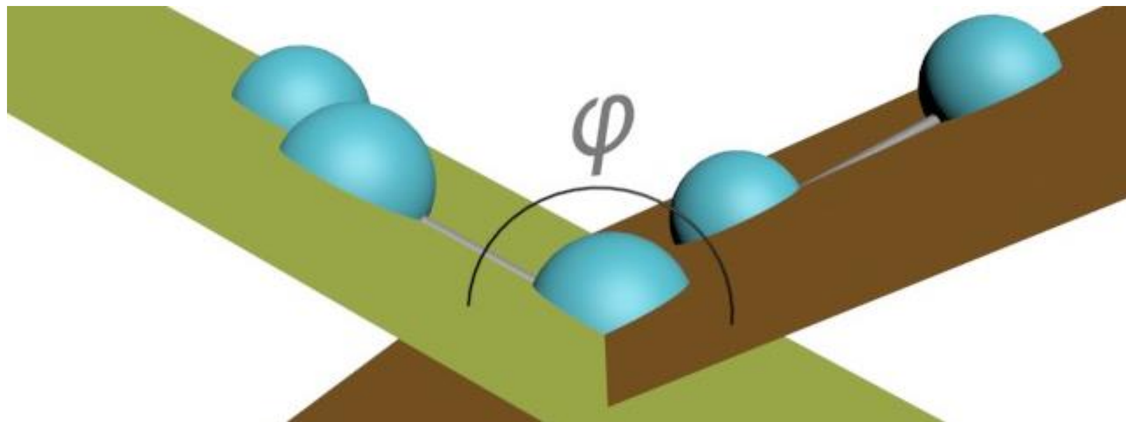
2. *Англійською* ***the angle potential*** (або в приблизному перекладі автора: *кутовий потенціал*). Дана функціональна залежність описує взаємодію між кожними двома атомами, які знаходяться на кінцях двох послідовних пар, наприклад, нехай існує молекула з трьох атомів з наступними зв'язками: атом номер 1 пов'язаний з атомом номер 2, і останній у свою чергу має ще один зв'язок із атомом номер 3, тоді кутовий потенціал буде описувати взаємодію між атомами 1 та 3 див. рисунок 1.11.




*Кулі відповідають атомам; лінії між кулями – зв'язкам між ними; стрілка вказує, між якими саме атомами буде відбувається взаємодія, а дуга відмічає кут (θ), величина якого впливає на значення кутового потенціалу.*

*Рисунок1.11 – Схематичне зображення до пояснення дії кутового (angle) потенціалу*

3. *Англійською* ***the torsion potential*** (або приблизний переклад автора: *обертовий потенціал*). Цей потенціал описує взаємодію в ланцюгах мінімум із чотирьох атомів у межах однієї молекули див. рисунок 1.12.




*Кулі відповідають атомам; дуга відмічає кут, який впливає на значення обертового потенціалу, літера (φ) відповідає значенню кута.*

*Рисунок1.12 – Схематичне зображення до пояснення дії обертового (torsion) потенціалу*

У якості математичного представлення потенціалу зв'язку (***bond potential***) зазвичай приймають гармонічний потенціал (формула (1.6)) у вигляді (1.10):

$$U_{зв}\left(r_{i,j}\right)=\frac{1}{2}k_{зв}\left(r_{i,j}-r_0\right)^2, \tag{1.10}$$

де $U_{зв}$ – потенціальна енергія, Дж;

$r_{i,j}$ – довжина зв'язку між атомами з номерами *i* та *j*, м;

$k_{зв}$ – константа пружності зв'язку, кг/с$^2$;

$r_0$ – рівноважна відстань між атомами з номерами *i* та *j* $\left(U_{зв}\left(r_0\right)=0\right)$, м. [9,11]

Кутовий потенціал (***angle potential***) описує зв'язок між ланцюжками з трьох атомів, і на величину значення цього потенціалу впливає кут, що утворює послідовна трійка з'єднаних атомів у молекулі. У даному випадку потенціальна залежність може прийняти одну з можливих форм (1.11) або (1.12):

$$U_{кут}\left(\theta\right)=-k_{\theta}\left(\cos\left(\theta-\theta_0\right)-1\right), \tag{1.11}$$

де $U_{кут}$ – значення енергії кутового потенціалу, Дж;

$\theta$ – кут, що утворюють частинки, градуси;

$k_{\theta}$ – константна величина, що являє собою силу зв'язку, Н;

$\theta_0$ – рівноважне значення кута, градуси.

У разі, якщо відхилення від рівноважного значення кута $\theta_0$ є невеликими можна використати кутовий потенціал у наступному вигляді (1.12):

$$U_{кут}\left(\theta\right)\approx\frac{1}{2}k_{\theta}\left(\theta-\theta_0\right)^2, \tag{1.12}$$

де $U_{кут}$ – значення енергії кутового потенціалу, Дж;

$\theta$ – кут, що утворюють частинки, градуси;

$k_{\theta}$ – константна величина, що являє собою силу зв'язку, Н;

$\theta_0$ – рівноважне значення кута, градуси.

Обертовий потенціал описує взаємодію в межах ланцюжків із чотирьох атомів у молекулі. У формі математичного виразу цей потенціал має вигляд (1.13):

$$U_{обер}\left(\phi_{i,j,k,l}\right)=\sum_{n=1}^{3}k_{\phi_n}\left(\cos\left(n\phi_{i,j,k,l}-\delta_n\right)+1\right), \tag{1.13}$$

де $U_{обер}$ – значення потенціальної енергії обертового потенціалу, Дж;

$\phi_{i,j,k,l}$ – кут між площиною, що задається координатами атомів з номерами *i*, *j*, *k* та площиною, що будується на основі координат атомів з номерами *j*, *k*, *l*, градуси;

$k_{\phi_n}$ – енергія зв'язку, Дж;

$\delta_n$ – фазовий зсув, градуси.

У якості альтернативи можна використати в наступному вигляді (1.14):

$$U_{обер}\left(\phi_{i,j,k,l}\right)=\sum_{n=1}^{3}k_{\phi_n}\cos^n\left(\phi_{i,j,k,l}\right), \qquad (1.14)$$

де $U_{обер}$ – значення потенціальної енергії обертового потенціалу, Дж;

$\phi_{i,j,k,l}$ – кут між площиною, що задається координатами атомів з номерами *i*, *j*, *k* та площиною, що будується на основі координат атомів з номерами *j*, *k*, *l*, градуси;

$k_{\phi_n}$ – енергія зв'язку, Дж. [9,11]

Рух такої моделі можна описати за допомогою другого закону Ньютона (див. формулу (1.1)) та набору потенціалів взаємодії (що були представлені в даному розділі вище та в розділі 1.1). Розв'язання рівняння Ньютона для кожної частинки дасть можливість отримувати їх координати та швидкості, які можна використати для подальшого аналізу.

У подальшому при виконанні дипломної роботи будуть використані потенціали *зв'язку* (***bond potential***) та *кутовий* (***angle potential***) для побудови моделі молекули.

## 1.4 Дискретизація руху частинок. Чисельний метод розв'язання рівнянь руху частинок

Для розв'язання рівнянь руху частинок використаємо метод, що найчастіше описується в літературних джерелах – ***схема Верле***. Відповідно до

даного підходу рівняння для розрахунку нових компонент векторів координат виглядають наступним чином (1.15):

$$x(t_{n+1}) = x(t_n) + v_x(t_n) \cdot \Delta t + \frac{1}{2} a_x(t_n) \cdot (\Delta t)^2, \qquad (1.15)$$

де $x(t_{n+1})$ – $OX$-компонента вектору координат деякої частинки в момент часу $t_{n+1}$, м або умовні модельні одиниці;

$n$ – поточний номер кроку (ітерації) алгоритму моделювання, $n \in [1, 2, ..., N]$;

$N$ – номер останнього кроку (ітерації) алгоритму;

$x(t_n)$ – $OX$-компонента вектору координат деякої частинки в момент часу $t_n$, м або умовні модельні одиниці відстані;

$v_x(t_n)$ – $OX$-компонента вектору швидкості деякої частинки в момент часу $t_n$, м/с або умовні модельні одиниці швидкості;

$\Delta t$ – крок за часом, деяка мала константа, с або умовні модельні одиниці часу;

$a_x(t_n)$ – $OX$-компонента вектору прискорення деякої частинки в момент часу $t_n$, м/с$^2$ або умовні модельні одиниці прискорення. [5,11]

Відповідні рівняння для розрахунку компонент вектору швидкостей (1.16):

$$v_x(t_{n+1}) = v_x(t_n) + \frac{1}{2}\left(a_x(t_{n+1}) + a_x(t_n)\right)\Delta t, \qquad (1.16)$$

де $v_x(t_{n+1})$ – $OX$-компонента вектору швидкості деякої частинки в момент часу $t_{n+1}$, м/с або умовні модельні одиниці швидкості;

$n$ – поточний номер кроку (ітерації) алгоритму моделювання, $n \in [1, 2, ..., N]$;

$N$ – номер останнього кроку (ітерації) алгоритму;

$a_x(t_{n+1})$ – $OX$-компонента вектору прискорення деякої частинки в момент часу $t_{n+1}$, м/с$^2$ або умовні модельні одиниці прискорення;

$\Delta t$ – крок за часом, деяка мала константа, с або умовні модельні одиниці часу.

Формули для знаходження компонент $y$ та $z$ радіус-вектору, а також вектору швидкості частинки можна отримати з формул (1.15) та (1.16) просто заміною усіх $x$ відповідно на $y$ та $z$.

# 2 ВИКОНАНІ САМОСТІЙНО ДОСЛІДНИЦЬКІ РОБОТИ

## 2.1 Отримання необхідних рівнянь та виразів

Для того щоб описати математичну модель молекули з гнучкими зв'язками між атомами в межах кожної молекули, потрібно зібрати всі необхідні рівняння разом.

Почнемо з рівняння Ньютона (формула (1.1)). Дане рівняння не потребує модифікацій. Наступним кроком опишемо сили, що діють у межах такої моделі. Відповідно до розділу 1.3 взаємодії між молекулами та атомами описуються за допомогою набору потенціалів, а також відомо, що між потенціальною енергією та силою існує взаємозв'язок (2.1):

$$\vec{F} = -\vec{\nabla} U , \tag{2.1}$$

де $\vec{F}$ – вектор діючої сили;

$\vec{\nabla}$ – набла-вектор;

$U$ – потенціальна енергія.

У свою чергу набла-вектор має вигляд (2.2):

$$\vec{\nabla} = \vec{e}_x \frac{\partial}{\partial x} + \vec{e}_y \frac{\partial}{\partial y} + \vec{e}_z \frac{\partial}{\partial z} , \tag{2.2}$$

де $\vec{e}_x$ – одиничний вектор тривимірного простору осі $OX$;

$\frac{\partial}{\partial x}$ – частинна похідна за $X$-компонентою;

$\vec{e}_y$ – одиничний вектор тривимірного простору осі $OY$;

$\frac{\partial}{\partial y}$ – частинна похідна за $Y$-компонентою;

$\vec{e}_z$ – одиничний вектор тривимірного простору осі *OZ*;

$\frac{\partial}{\partial z}$ – частинна похідна за *Z*-компонентою.

Окремі проекції на відповідні вісі будуть виглядати так (2.3):

$$F_x = -\frac{\partial}{\partial x} U\left(r_x\right), \tag{2.3}$$

де $F_x$ – довжина проекції вектору $\vec{F}$ на вісь *OX*, Н або умовні модельні одиниці;

$\frac{\partial}{\partial x}$ – частинна похідна за *X*-компонентою;

$U\left(\cdot\right)$ – значення потенціальної енергії, Дж або умовні модельні одиниці;

$r_x$ – довжина проекції вектору $\vec{r}$ на вісь *OX*, м або умовні модельні одиниці.

Усі інші проекції можна отримати з формули  просто замінивши $x$ на $y$, а потім на $z$. Отже, можна отримати вирази для сил відповідних потенціалів.

Далі приймемо, що $r_{i,j}$ – це значення евклідової відстані між парою частинок.

Тепер отримаємо з виразу потенціалу Леннард-Джонса (формула (1.2)) відповідну силу (формула (1.3)):

$$U\left(r_{i,j}\right) = 4\varepsilon_{i,j}\left[\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{12} - \left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{6}\right],$$

тоді вираз силу можна знайти так,

$$F\left(r_{i,j}\right) = -\left(U\left(r_{i,j}\right)\right)'_{r_{i,j}},$$

Відповідно до виразу вище можемо отримати:

$$F\left(r_{i,j}\right) = -\left(4\varepsilon_{i,j}\left[\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{12} - \left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{6}\right]\right)'_{r_{i,j}} \Rightarrow$$

$$F\left(r_{i,j}\right) = -\left(4\varepsilon_{i,j}\left[\sigma^{12} r^{-12} - \sigma^{6} r^{-6}\right]\right)'_{r_{i,j}} \Rightarrow$$

$$F\left(r_{i,j}\right)=-4\varepsilon_{i,j}\left[-12\sigma^{12}r^{(-12-1)}-(-6)\sigma^{6}r^{(-6-1)}\right]\Rightarrow$$

$$F\left(r_{i,j}\right)=-4\varepsilon_{i,j}(-6)\sigma^{-1}\left[2\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{13}-\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{7}\right]\Rightarrow$$

$$F\left(r_{i,j}\right)=24\frac{\varepsilon}{\sigma}\left[2\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{13}-\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{7}\right].$$

Вираз сили з потенціалу Леннард-Джонса отримано.

Отримаємо вираз для сили з *гармонічного пружного потенціалу* (1.6):

$$U\left(r_{i,j}\right)=\frac{1}{2}k_{i,j}\left(r_{i,j}-r_{i,j}^{0}\right)^{2},$$

отже, формулу для сили можна отримати наступним чином:

$$F\left(r_{i,j}\right)=-\left(\frac{1}{2}k_{i,j}\left(r_{i,j}-r_{i,j}^{0}\right)^{2}\right)'_{r_{i,j}}\Rightarrow$$

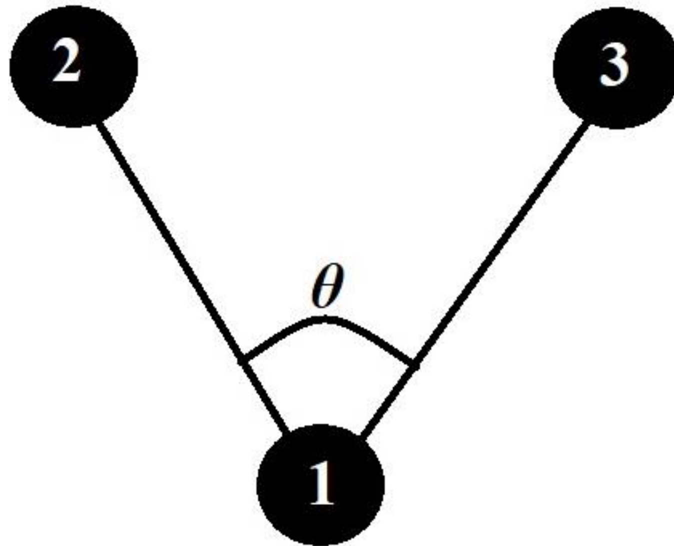
$$F\left(r_{i,j}\right)=-\left(\frac{1}{2}k_{i,j}2\left(r_{i,j}-r_{i,j}^{0}\right)\cdot 1\right)\Rightarrow$$

$$F\left(r_{i,j}\right)=-k_{i,j}\left(r_{i,j}-r_{i,j}^{0}\right).$$

Вираз для сили з пружного гармонічного потенціалу отримано.

Для визначення виразу сили з кутового потенціалу (формула (1.11)) потрібно розглянути невеликий приклад, так як випадок дещо складніший за рахунок участі 3 частинок (атомів) у потенціалі.

Нехай маємо наступну конфігурацію частинок див. рисунок 2.1:




Диски представляють собою частинки, які для зручності пронумеровано; лінії відображають зв’язки між частинками; кутовий потенціал буде описувати взаємодію між частинками з номерами 2 та 3.

*Рисунок2.1 – Схематичне зображення конфігурації молекули з трьох атомів*

Приймемо наступні позначення відповідно до конфігурації, що зображена вище:

$$\vec{r}_1 = \left( x_1 ; y_1 ; z_1 \right), \tag{2.4}$$

де $\vec{r}_1$ – радіус-вектор частинки з номером 1;

$x_1$ – компонента радіус-вектору частинки з номером 1, що відповідає вісі *OX*;

$y_1$ – компонента радіус-вектору частинки з номером 1, що відповідає вісі *OY*;

$z_1$ – компонента радіус-вектору частинки з номером 1, що відповідає вісі *OZ*.

$$\vec{r}_2 = \left( x_2 ; y_2 ; z_2 \right), \tag{2.5}$$

де $\vec{r}_2$ – радіус-вектор частинки з номером 2;

$x_2$ – компонента радіус-вектору частинки з номером 2, що відповідає вісі *OX*;

$y_2$ – компонента радіус-вектору частинки з номером 2, що відповідає вісі *OY*;

$z_2$ – компонента радіус-вектору частинки з номером 2, що відповідає вісі *OZ*.

$$\vec{r}_3 = \left( x_3 ; y_3 ; z_3 \right), \tag{2.6}$$

де $\vec{r}_3$ – радіус-вектор частинки з номером 3;

$x_3$ – компонента радіус-вектору частинки з номером 3, що відповідає вісі *OX*;

$y_3$ – компонента радіус-вектору частинки з номером 3, що відповідає вісі *OY*;

$z_3$ – компонента радіус-вектору частинки з номером 3, що відповідає вісі *OZ*.

$$\vec{v}_{1,2} = \left( x_2 - x_1; y_2 - y_1; z_2 - z_1 \right), \qquad (2.7)$$

де $\vec{v}_{1,2}$ – вектор, що проведено від частинки 1 до частинки 2.

$$\vec{v}_{1,3} = \left( x_3 - x_1; y_3 - y_1; z_3 - z_1 \right), \qquad (2.8)$$

де $\vec{v}_{1,3}$ – вектор, що проведено від частинки 1 до частинки 3.

Відповідно до означення можна ввести довжини цих векторів:

$$\left|\vec{v}_{1,2}\right| = \sqrt{\left(x_2 - x_1\right)^2 + \left(y_2 - y_1\right) + \left(z_2 - z_1\right)}, \qquad (2.9)$$

де $\left|\vec{v}_{1,2}\right|$ – довжина вектору $\vec{v}_{1,2}$.

$$\left|\vec{v}_{1,3}\right| = \sqrt{\left(x_3 - x_1\right)^2 + \left(y_3 - y_1\right) + \left(z_3 - z_1\right)}, \qquad (2.10)$$

де $\left|\vec{v}_{1,3}\right|$ – довжина вектору $\vec{v}_{1,3}$.

Тоді вираз для проекції сили на вісь *OX* для частинки з номером 2 (відповідно до Рисунку 2.1) можна отримати так з використанням формул (2.1) та (1.11):

$$F_{x_2} = -\frac{\partial U}{\partial x_2} \Rightarrow F_{x_2} = -\frac{\partial}{\partial x_2}\left[-k_{кут} \cdot \left(\cos\left(\theta - \theta_0\right) - 1\right)\right] \Rightarrow$$

$$F_{x_2} = -\frac{\partial}{\partial x_2} \cdot \frac{\partial \theta}{\partial \theta}\left[-k_{кут} \cdot \left(\cos\left(\theta - \theta_0\right) - 1\right)\right] \Rightarrow$$

$$F_{x_2} = -\frac{\partial}{\partial \theta}\left[-k_{кут} \cdot \left(\cos\left(\theta - \theta_0\right) - 1\right)\right] \cdot \frac{\partial \theta}{\partial x_2}.$$

Останній вираз відмітимо як окрему формулу (2.11):

$$F_{x_2} = -\frac{\partial}{\partial \theta}\left[-k_{кут} \cdot \left(\cos\left(\theta - \theta_0\right) - 1\right)\right] \cdot \frac{\partial \theta}{\partial x_2}.. \qquad (2.11)$$

Отримавши формулу (2.11) можна продовжити визначення загального виразу шляхом обчислення двох окремих частин $-\frac{\partial}{\partial \theta}\left[-k_{кут} \cdot \left(\cos\left(\theta - \theta_0\right) - 1\right)\right]$ та $\frac{\partial \theta}{\partial x_2}$.

Почнемо з першої результат позначимо як формула (2.12):

$$-\frac{\partial}{\partial\theta}\left[-k_{кут}\cdot\left(\cos\left(\theta-\theta_0\right)-1\right)\right]=-k_{кут}\cdot\sin\left(\theta-\theta_0\right). \qquad (2.12)$$

Для отримання виразу $\partial\theta/\partial x_2$ потрібно попередньо визначити вираз для $\theta$. Тоді спираючись на введені сутності можна отримати значення кута $\theta$ скориставшись означенням скалярного добутку векторів :(2.13)

$$\cos\left(\theta\right)=\frac{\left(\vec{v}_{1,2};\vec{v}_{1,3}\right)}{|\vec{v}_{1,2}|\cdot|\vec{v}_{1,3}|}\Rightarrow\theta=\arccos\left(\frac{\left(\vec{v}_{1,2};\vec{v}_{1,3}\right)}{|\vec{v}_{1,2}|\cdot|\vec{v}_{1,3}|}\right). \qquad (2.13)$$

Якщо записати з використанням координат частинок, то отримаємо :

$$\theta=\arccos\Big(\left[\left(x_2-x_1\right)\left(x_3-x_1\right)+\left(y_2-y_1\right)\left(y_3-y_1\right)+\left(z_2-z_1\right)\left(z_3-z_1\right)\right]\times$$
$$\times\Big[\sqrt{\left(x_2-x_1\right)^2+\left(y_2-y_1\right)^2+\left(z_2-z_1\right)^2}\times$$
$$\sqrt{\left(x_3-x_1\right)^2+\left(y_3-y_1\right)^2+\left(z_3-z_1\right)^2}\Big]^{-1}\Big). \qquad (2.14)$$

Отже, вираз для $\partial\theta/\partial x_2$ можна записати тепер таким чином (2.15):

$$\frac{\partial\theta}{\partial x_x}=\arccos\Big(\left[\left(x_2-x_1\right)\left(x_3-x_1\right)+\left(y_2-y_1\right)\left(y_3-y_1\right)+\left(z_2-z_1\right)\left(z_3-z_1\right)\right]\times$$
$$\times\Big[\sqrt{\left(x_2-x_1\right)^2+\left(y_2-y_1\right)^2+\left(z_2-z_1\right)^2}\times$$
$$\times\sqrt{\left(x_3-x_1\right)^2+\left(y_3-y_1\right)^2+\left(z_3-z_1\right)^2}\Big]^{-1}\Big)'_{x_2}. \qquad (2.15)$$

Позначимо окремо вираз, що знаходиться в круглих дужках (2.16):

$$\left[\left(x_2-x_1\right)\left(x_3-x_1\right)+\left(y_2-y_1\right)\left(y_3-y_1\right)+\left(z_2-z_1\right)\left(z_3-z_1\right)\right]\times$$
$$\times\Big[\sqrt{\left(x_2-x_1\right)^2+\left(y_2-y_1\right)^2+\left(z_2-z_1\right)^2}\times$$
$$\times\sqrt{\left(x_3-x_1\right)^2+\left(y_3-y_1\right)^2+\left(z_3-z_1\right)^2}\Big]^{-1}=A. \qquad (2.16)$$

Тоді формулу (2.15) можна переписати наступним чином (2.17):

$$\left(\frac{\partial\theta}{\partial x_2}\right)=-\frac{1}{\sqrt{1-A^2}}\cdot A'_{x_2}. \qquad (2.17)$$

За допомогою програми визначимо вирази для множників з формули (2.17). Відповідний код програми можна побачити перейшовши в ДОДАТОК Г.

У результаті для множнику $\left(-1/\sqrt{1-A^2}\right)$ було отримано наступний вираз

$$-1/\sqrt{1-A^2}=$$
$$=-1/\Big[1-\left(x_1^2-x_1x_2-x_1x_3+x_2x_3+y_1^2-y_1y_2-y_1y_3+y_2y_3+z_1^2-z_1z_2-z_1z_3+z_2z_3\right)^2\times$$
$$\times\left(\left((-x_1+x_2)^2+(-y_1+y_2)^2+(-z_1+z_2)^2\right)\cdot\left((-x_1+x_3)^2+(y_1+y_3)^2+(z_1+z_3)^2\right)\right)^{-1}\Big]^{1/2}$$

Або використовуючи введені позначення в формулах (2.4)-(2.10) можна перевисати рівність вище у векторному вигляді (2.18):

$$-1/\sqrt{1-A^2}=$$
$$=-1/\Big[1-\left(\left((\vec{r}_1;\vec{r}_1)-(\vec{r}_1;\vec{r}_2)-(\vec{r}_1;\vec{r}_3)+(\vec{r}_2;\vec{r}_3)\right)^2\times\right.$$
$$\left.\times\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{-1}\right)\Big]^{1/2}. \qquad (2.18)$$

Слід зазначити, що множники (2.12) та (2.18) будуть однаковими для атомів 2 та 3 в усіх трьох проекціях.

Множник $A'_{x_2}$ з формули (2.17) має вигляд:

$$A'_{x_2}=\Big[(-x_1+x_3)\cdot\left((x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2\right)+$$
$$+(x_1-x_2)\cdot\left(x_1^2-x_1x_2-x_1x_3+x_2x_3+y_1^2-y_1y_2-y_1y_3+\right.$$
$$\left.+y_2y_3+z_1^2-z_1z_2-z_1z_3+z_2z_3\right)\Big]\times$$
$$\times\Big[\left(\left((x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2\right)\times\right.$$
$$\left.\times\left((x_1-x_3)^2+(y_1-y_3)^2+(z_1-z_3)^2\right)\right)^{1/2}\times$$
$$\times\left((x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2\right)\Big]^{-1}.$$

У з використанням введених векторів множник можна переписати так (2.19):

$$A'_{x_2}=\Big[(-x_1+x_3)\cdot\left|\vec{v}_{1,2}\right|^2+(x_1-x_2)\times$$

$$\times\left(\left(\vec{r}_1;\vec{r}_1\right)-\left(\vec{r}_1;\vec{r}_2\right)-\left(\vec{r}_1;\vec{r}_3\right)+\left(\vec{r}_2;\vec{r}_3\right)\right)\Big]\times$$
$$\times\left[\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{1/2}\cdot\left|\vec{v}_{1,2}\right|^2\right]^{-1}. \qquad (2.19)$$

Отже, було отримано всі необхідні вирази, значить проекцію сили на вісь *OX* можна розрахувати за допомогою формул (2.11), (2.12), (2.16)-(2.19).

Як зазначалося раніше більшість множників виразу для сили будуть константними для всіх проекцій, отже, потрібно лише знайти необхідні частинні похідні від формули (2.16).

Отримаємо вирази, що будуть відрізнятися:

1. Для частинки 2 та проекції на вісь *OY* (2.20):

$$A'_{y_2}=\Big[\left(-y_1+y_3\right)\cdot\left|\vec{v}_{1,2}\right|^2+\left(y_1-y_2\right)\times$$
$$\times\left(\left(\vec{r}_1;\vec{r}_1\right)-\left(\vec{r}_1;\vec{r}_2\right)-\left(\vec{r}_1;\vec{r}_3\right)+\left(\vec{r}_2;\vec{r}_3\right)\right)\Big]\times$$
$$\times\left[\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{1/2}\cdot\left|\vec{v}_{1,2}\right|^2\right]^{-1}. \qquad (2.20)$$

2. Для частинки 2 та проекції на вісь *OZ* (2.21):

$$A'_{z_2}=\Big[\left(-z_1+z_3\right)\cdot\left|\vec{v}_{1,2}\right|^2+\left(z_1-z_2\right)\times$$
$$\times\left(\left(\vec{r}_1;\vec{r}_1\right)-\left(\vec{r}_1;\vec{r}_2\right)-\left(\vec{r}_1;\vec{r}_3\right)+\left(\vec{r}_2;\vec{r}_3\right)\right)\Big]\times$$
$$\times\left[\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{1/2}\cdot\left|\vec{v}_{1,2}\right|^2\right]^{-1}. \qquad (2.21)$$

3. Для частинки 3 та проекції на вісь *OX* (2.22):

$$A'_{x_3}=\Big[\left(-x_1+x_2\right)\cdot\left|\vec{v}_{1,3}\right|^2+\left(x_1-x_3\right)\times$$
$$\times\left(\left(\vec{r}_1;\vec{r}_1\right)-\left(\vec{r}_1;\vec{r}_2\right)-\left(\vec{r}_1;\vec{r}_3\right)+\left(\vec{r}_2;\vec{r}_3\right)\right)\Big]\times$$
$$\times\left[\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{1/2}\cdot\left|\vec{v}_{1,3}\right|^2\right]^{-1}. \qquad (2.22)$$

4. Для частинки 3 та проекції на вісь *OY* (2.23):

$$A'_{y_3}=\Big[\left(-y_1+y_2\right)\cdot\left|\vec{v}_{1,3}\right|^2+\left(y_1-y_3\right)\times$$
$$\times\left(\left(\vec{r}_1;\vec{r}_1\right)-\left(\vec{r}_1;\vec{r}_2\right)-\left(\vec{r}_1;\vec{r}_3\right)+\left(\vec{r}_2;\vec{r}_3\right)\right)\Big]\times$$

$$\times\left[\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{1/2}\cdot\left|\vec{v}_{1,3}\right|^2\right]^{-1}. \qquad (2.23)$$

5. Для частинки 3 та проекції на вісь $OZ$ (2.24):

$$A'_{z_3}=\left[\left(-z_1+z_2\right)\cdot\left|\vec{v}_{1,3}\right|^2+\left(z_1-z_3\right)\times\right.$$
$$\left.\times\left(\left(\vec{r}_1;\vec{r}_1\right)-\left(\vec{r}_1;\vec{r}_2\right)-\left(\vec{r}_1;\vec{r}_3\right)+\left(\vec{r}_2;\vec{r}_3\right)\right)\right]\times$$
$$\times\left[\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{1/2}\cdot\left|\vec{v}_{1,3}\right|^2\right]^{-1}. \qquad (2.24)$$

Зібравши всі частини разом отримаємо вираз (2.25):

$$F_{x_2}=-k_{кут}\cdot\sin\left(\theta-\theta_0\right)\times\left(-1\right)/$$
$$/\left[1-\left(\left(\left(\vec{r}_1;\vec{r}_1\right)-\left(\vec{r}_1;\vec{r}_2\right)-\left(\vec{r}_1;\vec{r}_3\right)+\left(\vec{r}_2;\vec{r}_3\right)\right)^2\cdot\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{-1}\right)\right]^{1/2}\times$$
$$\times\left[\left(-x_1+x_3\right)\cdot\left|\vec{v}_{1,2}\right|^2+\left(x_1-x_2\right)\cdot\left(\left(\vec{r}_1;\vec{r}_1\right)-\left(\vec{r}_1;\vec{r}_2\right)-\left(\vec{r}_1;\vec{r}_3\right)+\left(\vec{r}_2;\vec{r}_3\right)\right)\right]\times$$
$$\times\left[\left(\left|\vec{v}_{1,2}\right|^2\cdot\left|\vec{v}_{1,3}\right|^2\right)^{1/2}\cdot\left|\vec{v}_{1,2}\right|^2\right]^{-1}. \qquad (2.25)$$

Було отримано вираз для розрахунку значення сили на основі кутового потенціалу.

## 2.2 Перехід до умовних величин

Перед тим як почати вести мову про безрозмірні величини потрібно безпосередньо ввести розмірні змінні. Модель, що використано в кваліфікаційній роботі, заснована на параметрах моделі молекули води, що має назву TIPS3. Англійською дане скорочення можна розшифрувати наступним чином: ***T****ransferable* ***I****ntermolecular* ***P****otential with* ***3 S****ites*, або в приблизному перекладі автора: *трьох сторонній переданий міжмолекулярний потенціал*. [9]

У моделі використовується наступний список параметрів:

- для *потенціалу зв'язку* (*англійською* ***bond potential***):
  - $r^0_{i,j}=0.957$ Å;

- $k_{зв} = 450$ кг/с$^2$.

Å – ангстрем; позасистемна одиниця довжини 1 Å = $10^{-10}$ м; кал – калорія, позасистемна одиниця енергії: $1\,\text{кал} \approx 4.2\,\text{Дж}$. Відповідно $1\,\text{ккал} \approx 4200\,\text{Дж}$. У свою чергу $\left(\text{моль}^{-1}\right)$ відповідає сталій Авогадро $N_A = 6{,}022 \cdot 10^{23}$. [10,12,13]

- для *кутового потенціалу* (*англійською* ***angle potential***):
  - $\theta_0 = 104.52$ градуси;
  - $k_{кут} = 55$ ккал/моль.
- для *потенціалу Кулона:*
  - $q_H = 0.417 \cdot q_e$ Кл;
  - $q_O = -0.834 \cdot q_e$ Кл;
  - $k_{Кулон} = 9 \cdot 10^9$ $\text{Н} \cdot \text{м}^2 / \text{Кл}^2$. [10]

$q_H$ – заряд атому гідрогену, $q_e$ – елементарний електричний заряд рівний $1.602176487(40) \cdot 10^{-19}$ Кл. [10]

- для *потенціалу Леннард-Джонса*:
  - $\varepsilon_{HH} = 0.046$ ккал/моль;
  - $\sigma_{HH} = 0.4$ Å;
  - $\varepsilon_{OO} = 0.1521$ ккал/моль;
  - $\sigma_{OO} = 3.1506$ Å;
  - $\varepsilon_{OH} = 0.0836$ ккал/моль;
  - $\sigma_{OH} = 1.7753$ Å.
- для *рівняння руху Ньютона:*
  - $m_H = 1.0080$ u;
  - $m_O = 15.9994$ u.

u – (*англійською* ***unified atomic mass unit or Dalton (Da)***) уніфікована одиниця атомної маси для приведення величини до кілограмів (кг) можна скористатися наступною рівністю: $1\,\text{u} = 1.66053906660(50) \cdot 10^{-27}\,\text{кг}$. [14]

Основна ідея процесу отримання безрозмірних величин полягає в тому, щоб перетворити наявні змінні в рівняннях так, щоб прибрати їх фізичні величини.

Можливим шляхом реалізації цієї думки може бути використання константних значень. Більш наочний приклад наведено на рисунку 2.2. [9]

$$\text{безрозмірна величина} = \frac{\text{зміна з деякою розмірністю}}{\text{константа з тією ж розмірністю}}$$

*Рисунок2.2 – Схематичне зображення думки про отримання безрозмірних модельних величин*

Тому для того, щоб привести величини рівнянь до безрозмірного вигляду пропонується ввести наступні співвідношення:

1. Модельна величина маси (2.26):

$$\hat{m}_i = m_i \,/\, m_O, \tag{2.26}$$

де $\hat{m}_i$ – безрозмірна модельна величина маси;

$i$ – номер частинки (атома) в рамках моделі;

$m_i$ – маса атому з номером $i$, кг або u або Da;

$m_O$ – маса атому кисню, кг, або, u або Da.

2. Модельна величина рівноважної відстані (потенціал Леннард-Джонса) (2.27):

$$\hat{\sigma}_{i,j} = \sigma_{i,j} \,/\, \sigma_{OO}, \tag{2.27}$$

де $\hat{\sigma}_{i,j}$ – безрозмірна модельна величина рівноважної відстані;

$i$, $j$ – номери частинки (атома) у рамках моделі $i \neq j$;

$\sigma_{i,j}$ – рівноважна відстань для потенціалу Леннард-Джонса між частинками з номерами $i$ та $j$, м або Å;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню для потенціалу Леннард-Джонса, м або Å.

3. Модельна величина енергії зв’язку між парою атомів (потенціал Леннард-Джонса) (2.28):

$$\hat{\varepsilon}_{i,j} = \varepsilon_{i,j} / \varepsilon_{OO}, \tag{2.28}$$

де $\hat{\varepsilon}_{i,j}$ – безрозмірна величина енергії зв'язку;

*i, j* – номери частинки (атома) у рамках моделі $i \neq j$;

$\varepsilon_{i,j}$ – енергія зв'язку між парою атомів, Дж або кал/моль;

$\varepsilon_{OO}$ – енергія зв'язку між парою атомів кисню, Дж або кал/моль.

4. Модельна величина довжини проекції радіус-вектору кожного атому на вісі *OX*, *OY*, *OZ* (2.29), (2.30), (2.31):

$$\hat{x}_i = x_i / \sigma_{OO}, \tag{2.29}$$

$$\hat{y}_i = y_i / \sigma_{OO}, \tag{2.30}$$

$$\hat{z}_i = z_i / \sigma_{OO}, \tag{2.31}$$

де $i$ – номер частинки (атому);

$\hat{x}_i$ – безрозмірна модельна величина проекції радіус-вектору на вісь *OX* частинки з номером *i*;

$\hat{y}_i$ – безрозмірна модельна величина проекції радіус-вектору на вісь *OY* частинки з номером *i*;

$\hat{z}_i$ – безрозмірна модельна величина проекції радіус-вектору на вісь *OZ* частинки з номером *i*;

$x_i$ – довжина проекції радіус-вектору на вісь *OX* частинки з номером *i,* м або Å;

$y_i$ – довжина проекції радіус-вектору на вісь *OY* частинки з номером *i*, м або Å;

$z_i$ – довжина проекції радіус-вектору на вісь *OZ* частинки з номером *i*, м або Å;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню для потенціалу Леннард-Джонса, м або Å.

5. Модельна величина відстані між парою частинок (2.32):

$$\hat{r}_{i,j} = r_{i,j} / \sigma_{OO}, \tag{2.32}$$

де $\hat{r}_{i,j}$ – безрозмірна модельна величина відстані між парою частинок з номерами $i, j$;

$r_{i,j}$ – відстань між частинками з номерами *i* та *j*, м або Å;

$\sigma_{OO}$ – рівноважна відстань між парою частинок з номерами *i* та *j*, м або Å.

6. Модельна величина часу (2.33):

$$\hat{t} = t \Big/ \sqrt{\frac{m_O \cdot \sigma_{OO}^2}{\varepsilon_{OO}}}, \tag{2.33}$$

де $\hat{t}$ – безрозмірна величина часу;

$t$ – час, секунди;

$\sqrt{\dfrac{m_O \cdot \sigma_{OO}^2}{\varepsilon_{OO}}}$ – спеціальний масштабний множник, с.

В одиницях СІ можна переписати його наступним чином:

$$\sqrt{\frac{\text{кг} \cdot \text{м}^2}{\text{Дж}}} = \sqrt{\frac{\text{кг} \cdot \text{м}^2}{\text{Н} \cdot \text{м}}} = \sqrt{\text{кг} \cdot \text{м} \Big/ \frac{\text{кг} \cdot \text{м}}{\text{с}^2}} = \sqrt{\text{кг} \cdot \text{м} \cdot \frac{\text{с}^2}{\text{кг} \cdot \text{м}}} = \sqrt{\text{с}^2} = \text{с}.$$

7. Модельна величина множника $k_{кулон}$ (2.34):

$$\hat{k}_{кулон} = k_{кулон} \frac{q_H^2}{\sigma_{OO} \cdot \varepsilon_{OO}}, \tag{2.34}$$

де $\hat{k}_{кулон}$ – безрозмірна модельна величина;

$k_{кулон}$ – коефіцієнт пропорційності сили Кулона, значення якого рівне: $9 \cdot 10^9$, $\text{Н} \cdot \text{м}^2 / \text{Кл}^2$;

$q_H$ – значення заряду атому водню, що рівне $0.417 \cdot q_e$, Кл;

$q_e$ – елементарний електричний заряд рівний $1.602176487(40) \cdot 10^{-19}$ Кл; [10]

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню з номерами *i* та *j*, м або Å;

$\varepsilon_{OO}$ – енергія зв'язку між парою атомів кисню, Дж або кал/моль.

8. Модельна величина заряду атому (2.35):

$$\hat{q}_i = q_i / q_H, \tag{2.35}$$

де $\hat{q}_i$ – безрозмірна величина заряду;

$q_i$ – величина заряду атому з номером $i$, Кл;

$q_H$ – значення заряду атому водню.

9. Модельна величина множника $k_{зв}$ гармонічної сили (2.36):

$$\hat{k}_{зв} = k_{зв} \cdot \frac{\sigma_{OO}^2}{\varepsilon_{OO}}, \tag{2.36}$$

де $\hat{k}_{зв}$ – безрозмірна величина;

$k_{зв}$ – коефіцієнт жорсткості зв'язку, кг/с$^2$;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню з номерами $i$ та $j$, м або Å;

$\varepsilon_{OO}$ – енергія зв'язку між парою атомів кисню, Дж або кал/моль.

10. Модельна величина множника $k_{кут}$ кутового потенціалу (формула (1.11)) (2.37):

$$\hat{k}_{кут} = \frac{k_{кут}}{\varepsilon_{OO}}, \tag{2.37}$$

де $\hat{k}_{кут}$ – безрозмірна величина;

$k_{кут}$ – енергія зв'язку, Дж або кал/моль;

$\varepsilon_{OO}$ – енергія зв'язку між парою атомів кисню, Дж або кал/моль.

11. Модельне представлення радіус вектору (2.38):

$$\hat{\vec{r}}_i = \frac{\vec{r}_i}{\sigma_{OO}}, \tag{2.38}$$

де $\hat{\vec{r}}_i$ – безрозмірний варіант радіус-вектору для частинки з номером $i$;

$\vec{r}_i$ – розмірний варіант радіус-вектору для частинки з номером $i$, довжина вектору обчислюється в метрах (м) або ангстремах (Å);

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню з номерами $i$ та $j$, м або Å.

12. Модельне представлення вектору між частинками (2.39):

$$\hat{\vec{v}}_{i,j} = \frac{\vec{v}_{i,j}}{\sigma_{OO}}, \tag{2.39}$$

де $\hat{\vec{v}}_{i,j}$ – безрозмірний варіант вектору між двома частинками у просторі;

$\vec{v}_{i,j}$ – розмірний варіант вектору між двома частинками у просторі, довжина вектору обчислюється в метрах (м) або ангстремах (Å);

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню з номерами *i* та *j*, м або Å.

Перевірити коректність підібраних безрозмірних величин можна підстановкою їх у рівняння, що описують динаміку моделі.

Почнемо з рівняння руху Ньютона, якому використаємо знайдені вирази сили з потенціалів Леннард-Джонса (формула (1.3)), Кулона (формула (1.5)), зв'язку (формула (1.7)) та кутового потенціалів (формула (2.25)) (2.40):

$$
\begin{aligned}
m_i \cdot \frac{\partial^2 x_i}{\partial t^2} = \sum_{j=1, i\neq j}^{N} \Bigg( 24 \cdot \frac{\varepsilon_{i,j}}{\sigma_{i,j}} \left[ 2\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{13} - \left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{7} \right] + k_{кулон} \cdot \frac{q_i \cdot q_j}{r_{i,j}^2} \Bigg) + \\
+ \sum_{l=1}^{M} \left( -k_{зв} \cdot \left[ r_{i,l} - r_{i,l}^0 \right] \right) + \\
+ B \cdot \Big( -k_{кут} \cdot \sin\left(\theta - \theta_0\right) \times (-1) / \\
/ \Big[ 1 - \Big( \big( (\vec{r}_{i+1}; \vec{r}_{i+1}) - (\vec{r}_{i+1}; \vec{r}_i) - (\vec{r}_{i+1}; \vec{r}_{i+2}) + (\vec{r}_i; \vec{r}_{i+2}) \big)^2 \times \\
\times \left( \left|\vec{v}_{i+1,i}\right|^2 \cdot \left|\vec{v}_{i+1,i+2}\right|^2 \right)^{-1} \Big) \Big]^{1/2} \times \\
\times \Big[ \left(-x_{i+1} + x_{i+2}\right) \cdot \left|\vec{v}_{i+1,i}\right|^2 + \left(x_{i+1} - x_i\right) \times \\
\times \big( (\vec{r}_{i+1}; \vec{r}_{i+1}) - (\vec{r}_{i+1}; \vec{r}_i) - (\vec{r}_{i+1}; \vec{r}_{i+2}) + (\vec{r}_i; \vec{r}_{i+2}) \big) \Big] \times \\
\times \Big[ \left( \left|\vec{v}_{i+1,i}\right|^2 \cdot \left|\vec{v}_{i+1,i+2}\right|^2 \right)^{1/2} \cdot \left|\vec{v}_{i+1,i}\right|^2 \Big]^{-1} \Big) + \\
+ D \cdot \Big( -k_{кут} \cdot \sin\left(\theta - \theta_0\right) \times (-1) / \\
/ \Big[ 1 - \Big( \big( (\vec{r}_{i-1}; \vec{r}_{i-1}) - (\vec{r}_{i-1}; \vec{r}_{i-2}) - (\vec{r}_{i-1}; \vec{r}_i) + (\vec{r}_{i-2}; \vec{r}_i) \big)^2 \times \\
\times \left( \left|\vec{v}_{i-1,i-2}\right|^2 \cdot \left|\vec{v}_{i-1,i}\right|^2 \right)^{-1} \Big) \Big]^{1/2} \times \\
\times \Big[ \left(-x_{i-1} + x_i\right) \cdot \left|\vec{v}_{i-1,i-2}\right|^2 + \left(x_{i-1} - x_{i-2}\right) \times \\
\times \big( (\vec{r}_{i-1}; \vec{r}_{i-1}) - (\vec{r}_{i-1}; \vec{r}_{i-2}) - (\vec{r}_{i-1}; \vec{r}_i) + (\vec{r}_{i-2}; \vec{r}_i) \big) \Big] \times
\end{aligned}
$$

$$\times\left[\left(\left|\vec{v}_{i-1,i-2}\right|^{2}\cdot\left|\vec{v}_{i-1,i}\right|^{2}\right)^{1/2}\cdot\left|\vec{v}_{i-1,i-2}\right|^{2}\right]^{-1}\Bigg), \qquad (2.40)$$

де $N$ – загальна кількість атомів у системі;

$M$ – кількість зв'язків, що має частинка (атом) у межах моделі молекули;

$B$ – коефіцієнт рівний 1, якщо існує ланцюжок атомів $i$, $i+1$ та $i+2$ або $i-2$, $i-1$ та $i$ у межах молекули, а в іншому випадку 0.

Перетворимо ліву частину рівняння. Вираз для маси частинки:

$$m_i = \hat{m}_i \cdot m_O.$$

Вираз для другої похідної:

$$\frac{\partial x_i}{\partial t} = \frac{\partial\left(\hat{x}_i \cdot \sigma_{OO}\right)}{\partial\left(\hat{t}\cdot\sqrt{\dfrac{m_O\cdot\sigma_{OO}^2}{\varepsilon_{OO}}}\right)} \Rightarrow \frac{\partial x_i}{\partial t} = \frac{\sigma_{OO}}{\sqrt{\dfrac{m_O\cdot\sigma_{OO}^2}{\varepsilon_{OO}}}}\cdot\frac{\partial\hat{x}_i}{\partial\hat{t}};$$

$$\frac{\partial}{\partial t}\cdot\frac{\partial x_i}{\partial t} = \frac{\partial}{\partial\left(\hat{t}\cdot\sqrt{\dfrac{m_O\cdot\sigma_{OO}^2}{\varepsilon_{OO}}}\right)}\cdot\frac{\sigma_{OO}}{\sqrt{\dfrac{m_O\cdot\sigma_{OO}^2}{\varepsilon_{OO}}}}\cdot\frac{\partial\hat{x}_i}{\partial\hat{t}} \Rightarrow$$

$$\frac{\partial^2 x_i}{\partial t^2} = \frac{\sigma_{OO}}{\dfrac{m_O\cdot\sigma_{OO}^2}{\varepsilon_{OO}}}\cdot\frac{\partial}{\partial\hat{t}}\cdot\frac{\partial\hat{x}_i}{\partial\hat{t}} \Rightarrow$$

$$\frac{\partial^2 x_i}{\partial t^2} = \frac{\varepsilon_{OO}}{m_O\cdot\sigma_{OO}}\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}.$$

Тоді ліва частина рівняння руху Ньютона прийме вигляд:

$$m_i\cdot\frac{\partial^2 x_i}{\partial t^2} = \hat{m}_i\cdot m_O\cdot\frac{\varepsilon_{OO}}{m_O\cdot\sigma_{OO}}\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2} \Rightarrow$$

$$m_i\cdot\frac{\partial^2 x_i}{\partial t^2} = \left[\hat{m}_i\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}\right]\cdot\frac{\varepsilon_{OO}}{\sigma_{OO}}.$$

Тепер трансформуємо вираз сили Леннард-Джонса:

$$24\cdot\frac{\varepsilon_{i,j}}{\sigma_{i,j}}\left[2\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{13}-\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{7}\right]=24\cdot\frac{\hat{\varepsilon}_{i,j}\cdot\varepsilon_{OO}}{\hat{\sigma}_{i,j}\cdot\sigma_{OO}}\left[2\left(\frac{\hat{\sigma}_{i,j}\cdot\sigma_{OO}}{\hat{r}_{i,j}\cdot\sigma_{OO}}\right)^{13}-\left(\frac{\hat{\sigma}_{i,j}\cdot\sigma_{OO}}{\hat{r}_{i,j}\cdot\sigma_{OO}}\right)^{7}\right]\Rightarrow$$

$$24\cdot\frac{\varepsilon_{i,j}}{\sigma_{i,j}}\left[2\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{13}-\left(\frac{\sigma_{i,j}}{r_{i,j}}\right)^{7}\right]=24\cdot\frac{\hat{\varepsilon}_{i,j}\cdot\varepsilon_{OO}}{\hat{\sigma}_{i,j}\cdot\sigma_{OO}}\left[2\left(\frac{\hat{\sigma}_{i,j}}{\hat{r}_{i,j}}\right)^{13}-\left(\frac{\hat{\sigma}_{i,j}}{\hat{r}_{i,j}}\right)^{7}\right].$$

Наступна трансформація для виразу сили Кулона:

$$k_{кулон}\cdot\frac{q_i\cdot q_j}{r_{i,j}^2}=\left(\hat{k}_{кулон}\cdot\frac{\sigma_{OO}\cdot\varepsilon_{OO}}{q_e^2}\right)\cdot\frac{\hat{q}_i\cdot q_e\cdot q_j\cdot q_e}{\hat{r}_{i,j}^2\cdot\sigma_{OO}^2}\Rightarrow$$

$$k_{кулон}\cdot\frac{q_i\cdot q_j}{r_{i,j}^2}=\hat{k}_{кулон}\cdot\frac{\hat{q}_i\cdot\hat{q}_j}{\hat{r}_{i,j}^2}\cdot\left[\frac{\varepsilon_{OO}}{\sigma_{OO}}\right].$$

Тепер трансформуємо вираз гармонічної сили:

$$-k_{зв}\cdot\left[r_{i,l}-r_{i,l}^0\right]=-\hat{k}_{зв}\cdot\frac{\varepsilon_{OO}}{\sigma_{OO}^2}\left[\hat{r}_{i,l}\cdot\sigma_{OO}-\hat{r}_{i,l}^0\cdot\sigma_{OO}\right]\Rightarrow$$

$$-k_{зв}\cdot\left[r_{i,l}-r_{i,l}^0\right]=-\hat{k}_{зв}\cdot\left[\hat{r}_{i,l}-\hat{r}_{i,l}^0\right]\left(\frac{\varepsilon_{OO}}{\sigma_{OO}}\right).$$

Наступним кроком перетворимо вираз, що отримано з кутового потенціалу. Зауважимо, що можна перетворити лише одну з двох наявних частин. Перетворимо частину, що починається з коефіцієнта $B$:

$$\hat{F}_{кут}=\left(-\hat{k}_{кут}\cdot\varepsilon_{OO}\cdot\sin\left(\theta-\theta_0\right)\times(-1)\right)/$$

$$/\left[1-\left(\left(\left(\hat{\vec{r}}_{i+1}\cdot\sigma_{OO};\hat{\vec{r}}_{i+1}\cdot\sigma_{OO}\right)-\left(\hat{\vec{r}}_{i+1}\cdot\sigma_{OO};\hat{\vec{r}}_i\cdot\sigma_{OO}\right)+\right.\right.\right.$$

$$\left.+\left(\hat{\vec{r}}_i\cdot\sigma_{OO};\hat{\vec{r}}_{i+2}\cdot\sigma_{OO}\right)-\left(\hat{\vec{r}}_{i+1}\cdot\sigma_{OO};\hat{\vec{r}}_{i+2}\cdot\sigma_{OO}\right)\right)^2\times$$

$$\left.\left.\times\left(\left(\hat{\vec{v}}_{i+1,i}\cdot\sigma_{OO};\hat{\vec{v}}_{i+1,i}\cdot\sigma_{OO}\right)\cdot\left(\hat{\vec{v}}_{i+1,i+2}\cdot\sigma_{OO};\hat{\vec{v}}_{i+1,i+2}\cdot\sigma_{OO}\right)\right)^{-1}\right)\right]^{1/2}\times$$

$$\times\left[\left(-\hat{x}_{i+1}\cdot\sigma_{OO}+\hat{x}_{i+2}\cdot\sigma_{OO}\right)\cdot\left(\hat{\vec{v}}_{i+1,i}\cdot\sigma_{OO};\hat{\vec{v}}_{i+1,i}\cdot\sigma_{OO}\right)+\left(\hat{x}_{i+1}\cdot\sigma_{OO}-\hat{x}_i\cdot\sigma_{OO}\right)\times\right.$$

$$\times\left(\left(\hat{\vec{r}}_{i+1}\cdot\sigma_{OO};\hat{\vec{r}}_{i+1}\cdot\sigma_{OO}\right)-\left(\hat{\vec{r}}_{i+1}\cdot\sigma_{OO};\hat{\vec{r}}_i\cdot\sigma_{OO}\right)+\right.$$

$$\left.\left.+\left(\hat{\vec{r}}_i\cdot\sigma_{OO};\hat{\vec{r}}_{i+2}\cdot\sigma_{OO}\right)-\left(\hat{\vec{r}}_{i+1}\cdot\sigma_{OO};\hat{\vec{r}}_{i+2}\cdot\sigma_{OO}\right)\right)\right]\times$$

$$\left[\left(\left(\hat{\vec{v}}_{i+1,i}\cdot\sigma_{OO};\hat{\vec{v}}_{i+1,i}\cdot\sigma_{OO}\right)\cdot\left(\hat{\vec{v}}_{i+1,i+2}\cdot\sigma_{OO};\hat{\vec{v}}_{i+1,i+2}\cdot\sigma_{OO}\right)\right)^{1/2}\times\right.$$

$$\times\left(\vec{\hat{v}}_{i+1,i}\cdot\sigma_{OO};\vec{\hat{v}}_{i+1,i}\cdot\sigma_{OO}\right)\Big]^{-1}\Big).$$

Спростимо рівняння вище та позначимо окремо (2.41):

$$\hat{F}_{кут}=\Big(-\hat{k}_{кут}\cdot\sin(\theta-\theta_0)\times(-1)\,/$$
$$/\Big[1-\Big(\Big(\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i+1}\big)-\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i}\big)+\big(\vec{\hat{r}}_{i};\vec{\hat{r}}_{i+2}\big)-\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i+2}\big)\Big)^2\times$$
$$\times\Big(\big(\vec{\hat{v}}_{i+1,i};\vec{\hat{v}}_{i+1,i}\big)\cdot\big(\vec{\hat{v}}_{i+1,i+2};\vec{\hat{v}}_{i+1,i+2}\big)\Big)^{-1}\Big)\Big]^{1/2}\times$$
$$\times\Big[\big(-\hat{x}_{i+1}+\hat{x}_{i+2}\big)\cdot\big(\vec{\hat{v}}_{i+1,i};\vec{\hat{v}}_{i+1,i}\big)+\big(\hat{x}_{i+1}-\hat{x}_{i}\big)\times$$
$$\times\Big(\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i+1}\big)-\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i}\big)+\big(\vec{\hat{r}}_{i};\vec{\hat{r}}_{i+2}\big)-\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i+2}\big)\Big)\Big]\times$$
$$\times\Big[\Big(\big(\vec{\hat{v}}_{i+1,i};\vec{\hat{v}}_{i+1,i}\big)\cdot\big(\vec{\hat{v}}_{i+1,i+2};\vec{\hat{v}}_{i+1,i+2}\big)\Big)^{1/2}\times$$
$$\times\big(\vec{\hat{v}}_{i+1,i};\vec{\hat{v}}_{i+1,i}\big)\Big]^{-1}\Big)\cdot\left[\frac{\varepsilon_{OO}}{\sigma_{OO}}\right]. \qquad (2.41)$$

Тепер можна зібрати весь вираз.

$$m_i\cdot\frac{\partial^2 x_i}{\partial t^2}=\left[\hat{m}_i\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}\right]\cdot\frac{\varepsilon_{OO}}{\sigma_{OO}}=$$
$$=\sum_{j=1;i\neq j}^{N}\left[\frac{\varepsilon_{OO}}{\sigma_{OO}}\right]\cdot\left(24\cdot\frac{\hat{\varepsilon}_{i,j}}{\hat{\sigma}_{i,j}}\left[2\left(\frac{\hat{\sigma}_{i,j}}{\hat{r}_{i,j}}\right)^{13}-\left(\frac{\hat{\sigma}_{i,j}}{\hat{r}_{i,j}}\right)^{7}\right]+\hat{k}_{кулон}\cdot\frac{\hat{q}_i\cdot\hat{q}_j}{\hat{r}_{i,j}^2}\right)+$$
$$+\sum_{l=1}^{M}\left[\frac{\varepsilon_{OO}}{\sigma_{OO}}\right]\cdot\left(-\hat{k}_{зв}\cdot\left[\hat{r}_{i,l}-\hat{r}_{i,l}^{0}\right]\right)+$$
$$+B\cdot\left[\frac{\varepsilon_{OO}}{\sigma_{OO}}\right]\cdot\Big(-\hat{k}_{кут}\cdot\sin(\theta-\theta_0)\times(-1)\,/$$
$$/\Big[1-\Big(\Big(\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i+1}\big)-\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i}\big)-\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i+2}\big)+\big(\vec{\hat{r}}_{i};\vec{\hat{r}}_{i+2}\big)\Big)^2\times$$
$$\times\Big(\big|\vec{\hat{v}}_{i+1,i}\big|^2\cdot\big|\vec{\hat{v}}_{i+1,i+2}\big|^2\Big)^{-1}\Big)\Big]^{1/2}\times$$
$$\times\Big[\big(-\hat{x}_{i+1}+\hat{x}_{i+2}\big)\cdot\big|\vec{\hat{v}}_{i+1,i}\big|^2+\big(\hat{x}_{i+1}-\hat{x}_{i}\big)\times$$
$$\times\Big(\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i+1}\big)-\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i}\big)-\big(\vec{\hat{r}}_{i+1};\vec{\hat{r}}_{i+2}\big)+\big(\vec{\hat{r}}_{i};\vec{\hat{r}}_{i+2}\big)\Big)\Big]\times$$

$$\times\left[\left(\left|\hat{\vec{v}}_{i+1,i}\right|^2\cdot\left|\hat{\vec{v}}_{i+1,i+2}\right|^2\right)^{1/2}\cdot\left|\hat{\vec{v}}_{i+1,i}\right|^2\right]^{-1}\Bigg)+$$

$$+D\cdot\left[\frac{\varepsilon_{OO}}{\sigma_{OO}}\right]\cdot\left(-\hat{k}_{кут}\cdot\sin\left(\theta-\theta_0\right)\times\left(-1\right)/\right.$$

$$/\left[1-\left(\left(\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i-1}\right)-\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i-2}\right)-\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i}\right)+\left(\hat{\vec{r}}_{i-2};\hat{\vec{r}}_{i}\right)\right)^2\times\right.\right.$$

$$\left.\left.\times\left(\left|\hat{\vec{v}}_{i-1,i-2}\right|^2\cdot\left|\hat{\vec{v}}_{i-1,i}\right|^2\right)^{-1}\right)\right]^{1/2}\times$$

$$\times\left[\left(-\hat{x}_{i-1}+\hat{x}_{i}\right)\cdot\left|\hat{\vec{v}}_{i-1,i-2}\right|^2+\left(\hat{x}_{i-1}-\hat{x}_{i-2}\right)\times\right.$$

$$\left.\times\left(\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i-1}\right)-\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i-2}\right)-\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i}\right)+\left(\hat{\vec{r}}_{i-2};\hat{\vec{r}}_{i}\right)\right)\right]\times$$

$$\times\left[\left(\left|\hat{\vec{v}}_{i-1,i-2}\right|^2\cdot\left|\hat{\vec{v}}_{i-1,i}\right|^2\right)^{1/2}\cdot\left|\hat{\vec{v}}_{i-1,i-2}\right|^2\right]^{-1}\Bigg).$$

Слід також зауважити, що в лівій частині перетвореного рівняння присутня константа $\frac{\varepsilon_{OO}}{\sigma_{OO}}$, скоротимо її, й отримаємо остаточний вигляд рівняння руху з безрозмірними величинами (2.42):

$$\hat{m}_i\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}=$$

$$=\sum_{j=1;i\neq j}^{N}\left(24\cdot\frac{\hat{\varepsilon}_{i,j}}{\hat{\sigma}_{i,j}}\left[2\left(\frac{\hat{\sigma}_{i,j}}{\hat{r}_{i,j}}\right)^{13}-\left(\frac{\hat{\sigma}_{i,j}}{\hat{r}_{i,j}}\right)^{7}\right]+\hat{k}_{кулон}\cdot\frac{\hat{q}_i\cdot\hat{q}_j}{\hat{r}_{i,j}^2}\right)+$$

$$+\sum_{l=1}^{M}\left(-\hat{k}_{зв}\cdot\left[\hat{r}_{i,l}-\hat{r}_{i,l}^0\right]\right)+B\cdot\left(-\hat{k}_{кут}\cdot\sin\left(\theta-\theta_0\right)\times\left(-1\right)/\right.$$

$$/\left[1-\left(\left(\left(\hat{\vec{r}}_{i+1};\hat{\vec{r}}_{i+1}\right)-\left(\hat{\vec{r}}_{i+1};\hat{\vec{r}}_{i}\right)-\left(\hat{\vec{r}}_{i+1};\hat{\vec{r}}_{i+2}\right)+\left(\hat{\vec{r}}_{i};\hat{\vec{r}}_{i+2}\right)\right)^2\times\right.\right.$$

$$\left.\left.\times\left(\left|\hat{\vec{v}}_{i+1,i}\right|^2\cdot\left|\hat{\vec{v}}_{i+1,i+2}\right|^2\right)^{-1}\right)\right]^{1/2}\times$$

$$\times\left[\left(-\hat{x}_{i+1}+\hat{x}_{i+2}\right)\cdot\left|\hat{\vec{v}}_{i+1,i}\right|^2+\left(\hat{x}_{i+1}-\hat{x}_{i}\right)\times\right.$$

$$\left.\times\left(\left(\hat{\vec{r}}_{i+1};\hat{\vec{r}}_{i+1}\right)-\left(\hat{\vec{r}}_{i+1};\hat{\vec{r}}_{i}\right)-\left(\hat{\vec{r}}_{i+1};\hat{\vec{r}}_{i+2}\right)+\left(\hat{\vec{r}}_{i};\hat{\vec{r}}_{i+2}\right)\right)\right]\times$$

$$\times\left[\left(\left|\hat{\vec{v}}_{i+1,i}\right|^2\cdot\left|\hat{\vec{v}}_{i+1,i+2}\right|^2\right)^{1/2}\cdot\left|\hat{\vec{v}}_{i+1,i}\right|^2\right]^{-1}\Bigg)+$$

$$+D\cdot\left(-\hat{k}_{кут}\cdot\sin\left(\theta-\theta_0\right)\times\left(-1\right)/\right.$$

$$/\left[1-\left(\left(\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i-1}\right)-\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i-2}\right)-\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i}\right)+\left(\hat{\vec{r}}_{i-2};\hat{\vec{r}}_{i}\right)\right)^2\times\right.\right.$$

$$\left.\left.\times\left(\left|\hat{\vec{v}}_{i-1,i-2}\right|^2\cdot\left|\hat{\vec{v}}_{i-1,i}\right|^2\right)^{-1}\right)\right]^{1/2}\times$$

$$\times\left[\left(-\hat{x}_{i-1}+\hat{x}_i\right)\cdot\left|\hat{\vec{v}}_{i-1,i-2}\right|^2+\left(\hat{x}_{i-1}-\hat{x}_{i-2}\right)\times\right.$$

$$\left.\times\left(\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i-1}\right)-\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i-2}\right)-\left(\hat{\vec{r}}_{i-1};\hat{\vec{r}}_{i}\right)+\left(\hat{\vec{r}}_{i-2};\hat{\vec{r}}_{i}\right)\right)\right]\times$$

$$\times\left[\left(\left|\hat{\vec{v}}_{i-1,i-2}\right|^2\cdot\left|\hat{\vec{v}}_{i-1,i}\right|^2\right)^{1/2}\cdot\left|\hat{\vec{v}}_{i-1,i-2}\right|^2\right]^{-1}\Bigg). \qquad (2.42)$$

Рівняння для компонент $y$ та $z$ можна отримати простою заміною усіх $x$ на $y$ та $z$ відповідно.

Тепер можна перетворити рівняння чисельної схеми для розрахунку координат та швидкостей частинок. Для більш компактного запису приймемо, що масштабний множник часу $\sqrt{\dfrac{m_O\cdot\sigma_{OO}^2}{\varepsilon_{OO}}}=\tilde{t}$ .

$$x\left(t_{n+1}\right)=x\left(t_n\right)+v_x\left(t_n\right)\cdot\Delta t+\frac{1}{2}a_x\left(t_n\right)\cdot\left(\Delta t\right)^2\Rightarrow$$

$$\hat{x}\left(\hat{t}_{n+1}\cdot\tilde{t}\right)\cdot\sigma_{OO}=\hat{x}\left(\hat{t}_n\cdot\tilde{t}\right)\cdot\sigma_{OO}+\left[\frac{\sigma_{OO}}{\tilde{t}}\cdot\frac{\partial\hat{x}_i}{\partial\hat{t}}\right]\left(\hat{t}_n\cdot\tilde{t}\right)\cdot\Delta\left[\hat{t}\cdot\tilde{t}\right]+$$

$$+\frac{1}{2}\left[\frac{\sigma_{OO}}{\tilde{t}^2}\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}\right]\left(\hat{t}\cdot\tilde{t}\right)\cdot\Delta\left[\hat{t}\cdot\tilde{t}\right]^2\Rightarrow$$

$$\hat{x}\left(\hat{t}_{n+1}\cdot\tilde{t}\right)\cdot\sigma_{OO}=\hat{x}\left(\hat{t}_n\cdot\tilde{t}\right)\cdot\sigma_{OO}+\left[\frac{\tilde{t}}{\tilde{t}}\sigma_{OO}\right]\left[\frac{\partial\hat{x}_i}{\partial\hat{t}}\right]\left(\hat{t}_n\cdot\tilde{t}\right)\cdot\Delta\hat{t}+$$

$$\left[\frac{\tilde{t}^2}{\tilde{t}^2}\sigma_{OO}\right]\cdot\left[\frac{1}{2}\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}\right]\left(\hat{t}\cdot\tilde{t}\right)\Delta\hat{t}^2\Rightarrow$$

$$\hat{x}\left(\hat{t}_{n+1}\cdot\tilde{t}\right)=\hat{x}\left(\hat{t}_n\cdot\tilde{t}\right)+\left[\frac{\partial\hat{x}_i}{\partial\hat{t}}\right]\left(\hat{t}_n\cdot\tilde{t}\right)\cdot\Delta\hat{t}+\left[\frac{1}{2}\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}\right]\left(\hat{t}\cdot\tilde{t}\right)\Delta\hat{t}^2.$$

Позначимо окремо останнє рівняння (2.43):

$$\hat{x}\left(\hat{t}_{n+1}\cdot\tilde{t}\right)=\hat{x}\left(\hat{t}_{n}\cdot\tilde{t}\right)+\left[\frac{\partial\hat{x}_i}{\partial\hat{t}}\right]\left(\hat{t}_{n}\cdot\tilde{t}\right)\cdot\Delta\hat{t}+\left[\frac{1}{2}\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}\right]\left(\hat{t}\cdot\tilde{t}\right)\Delta\hat{t}^2. \tag{2.43}$$

Тоді скориставшись отриманими виразами вище можна перетворити рівняння для розрахунку швидкостей. Для скорочення об’єму викладок введемо ще декілька узагальнень: $\frac{\sigma_{OO}}{\tilde{t}}\cdot\frac{\partial\hat{x}_i}{\partial\hat{t}}=v_x$, $\frac{\partial\hat{x}_i}{\partial\hat{t}}=\hat{v}_x$ та $\frac{\sigma_{OO}}{\tilde{t}^2}\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}=a_x$, $\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}=\hat{a}_x$.

$$v_x\left(t_{n+1}\right)=v_x\left(t_n\right)+\frac{1}{2}\left(a_x\left(t_{n+1}\right)+a_x\left(t_n\right)\right)\Delta t\Rightarrow$$

$$\left[\frac{\sigma_{OO}}{\tilde{t}}\cdot\frac{\partial\hat{x}_i}{\partial\hat{t}}\right]\left(\hat{t}_{n+1}\cdot\tilde{t}\right)=\left[\frac{\sigma_{OO}}{\tilde{t}}\cdot\frac{\partial\hat{x}_i}{\partial\hat{t}}\right]\left(\hat{t}_{n}\cdot\tilde{t}\right)+$$

$$+\frac{1}{2}\left(\left[\frac{\sigma_{OO}}{\tilde{t}^2}\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}\right]\left(\hat{t}_{n+1}\cdot\tilde{t}\right)+\left[\frac{\sigma_{OO}}{\tilde{t}^2}\cdot\frac{\partial^2\hat{x}_i}{\partial\hat{t}^2}\right]\left(\hat{t}_{n}\cdot\tilde{t}\right)\right)\Delta\left[\hat{t}\cdot\tilde{t}\right]\Rightarrow$$

$$\left[\frac{\sigma_{OO}}{\tilde{t}}\right]\cdot\hat{v}_x\left(\hat{t}_{n+1}\cdot\tilde{t}\right)=\left[\frac{\sigma_{OO}}{\tilde{t}}\right]\cdot\hat{v}_x\left(\hat{t}_{n}\cdot\tilde{t}\right)+$$

$$\frac{1}{2}+\left(\left[\frac{\sigma_{OO}}{\tilde{t}^2}\right]\cdot\hat{a}_x\left(\hat{t}_{n+1}\cdot\tilde{t}\right)+\left[\frac{\sigma_{OO}}{\tilde{t}^2}\right]\cdot\hat{a}_x\left(\hat{t}_{n}\cdot\tilde{t}\right)\right)\cdot\left[\tilde{t}\right]\cdot\Delta\hat{t}\Rightarrow$$

$$\hat{v}_x\left(\hat{t}_{n+1}\cdot\tilde{t}\right)=\hat{v}_x\left(\hat{t}_{n}\cdot\tilde{t}\right)+\frac{1}{2}\left[\hat{a}_x\left(\hat{t}_{n+1}\cdot\tilde{t}\right)+\hat{a}_x\left(\hat{t}_{n}\cdot\tilde{t}\right)\right]\cdot\Delta\hat{t}.$$

Знову відмітимо окремо результат (2.44):

$$\hat{v}_x\left(\hat{t}_{n+1}\cdot\tilde{t}\right)=\hat{v}_x\left(\hat{t}_{n}\cdot\tilde{t}\right)+\frac{1}{2}\left[\hat{a}_x\left(\hat{t}_{n+1}\cdot\tilde{t}\right)+\hat{a}_x\left(\hat{t}_{n}\cdot\tilde{t}\right)\right]\cdot\Delta\hat{t}. \tag{2.44}$$

Базуючись на рівняннях (2.26) – (2.44) можна будувати комп’ютерну програму для проведення віртуальних експериментів.

## 2.3 Обчислення модельних константних значень у безрозмірних величинах

Обчислення безрозмірних констант виконуємо за допомогою формул, що було визначено в розділі «Перехід до умовних величин».

Безрозмірне значення маси атому водню (2.45):

$$\hat{m}_H = m_H / m_O \Rightarrow \hat{m}_H = \frac{1.008}{15.9994} \cdot \frac{\text{u}}{\text{u}} \cdot \Rightarrow$$

$$\hat{m}_H \approx 0.06300236258859. \tag{2.45}$$

де $\hat{m}_H$ – безрозмірна величина маси атому водню;

$m_H$ – маса атому водню, u;

$m_O$ – маса атому кисню, u.

Розрахунок безрозмірного значення маси атому кисню (2.46):

$$\hat{m}_O = m_O / m_O \Rightarrow \hat{m}_O = \frac{15.9994}{15.9994} \cdot \frac{\text{u}}{\text{u}} \cdot \Rightarrow$$

$$\hat{m}_O = 1.0. \tag{2.46}$$

де $\hat{m}_O$ – безрозмірна величина маси атому кисню;

$m_O$ – значення маси атому кисню, u.

Розрахунок безрозмірної величини рівноважної відстані між парою моделей атомів кисню для виразу потенціалу Леннард-Джонса (2.47):

$$\hat{\sigma}_{OO} = \sigma_{OO} / \sigma_{OO} \Rightarrow \hat{\sigma}_{OO} = \frac{3.1506}{3.1506} \cdot \frac{\text{Å}}{\text{Å}} \Rightarrow$$

$$\hat{\sigma}_{OO} = 1.0. \tag{2.47}$$

де $\hat{\sigma}_{OO}$ – безрозмірне значення відстані між парою частинок;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню для потенціалу Леннард-Джонса, Å.

Розрахунок безрозмірного значення рівноважної відстані між парою атомів водню (2.48):

$$\hat{\sigma}_{HH} = \sigma_{HH} / \sigma_{OO} \Rightarrow \hat{\sigma}_{HH} = \frac{0.4}{3.1506} \cdot \frac{\text{Å}}{\text{Å}} \Rightarrow$$

$$\hat{\sigma}_{HH} = 0.12695994413762. \tag{2.48}$$

де $\hat{\sigma}_{HH}$ – безрозмірна величина рівноважної відстані між парою атомів водню для виразів Леннард-Джонса;

$\sigma_{HH}$ – рівноважна відстань між парою атомів водню для виразів Леннард-Джонса, Å;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню для виразів Леннард-Джонса, Å.

Розрахунок безрозмірної величини рівноважної відстані між атомом кисню та атомом водню (2.49):

$$\hat{\sigma}_{OH} = \sigma_{OH} / \sigma_{OO} \Rightarrow \hat{\sigma}_{OH} = \frac{1.7753}{3.1506} \cdot \frac{\text{Å}}{\text{Å}} \Rightarrow$$

$$\hat{\sigma}_{OH} = 0.56347997206881. \qquad (2.49)$$

де $\hat{\sigma}_{OH}$ – безрозмірне значення відстані між атомами кисню та водню;

$\sigma_{OH}$ – рівноважне значення відстані між атомами кисню та водню для виразів Леннард-Джонса, Å;

$\sigma_{OO}$ – рівноважне значення відстані між парою атомів кисню для виразів Леннард-Джонса, Å.

Обчислення безрозмірного значення енергії зв'язку між парою атомів кисню в рівняннях Леннард-Джонса (2.50):

$$\hat{\varepsilon}_{OO} = \varepsilon_{OO} / \varepsilon_{OO} \Rightarrow \hat{\varepsilon}_{OO} = \frac{0.1521}{0.1521} \cdot \frac{\text{ккал/моль}}{\text{ккал/моль}} \Rightarrow$$

$$\hat{\varepsilon}_{OO} = 1.0. \qquad (2.50)$$

де $\hat{\varepsilon}_{OO}$ – безрозмірне значення енергії зв'язку між парою атомів кисню;

$\varepsilon_{OO}$ – енергія зв'язку між парою атомів кисню, ккал/моль.

Розрахунок безрозмірного значення енергії зв'язку між парою атомів водню в рівняннях Леннард-Джонса (2.51):

$$\hat{\varepsilon}_{HH} = \varepsilon_{HH} / \varepsilon_{OO} \Rightarrow \hat{\varepsilon}_{HH} = \frac{0.046}{0.1521} \cdot \frac{\text{ккал/моль}}{\text{ккал/моль}} \Rightarrow$$

$$\hat{\varepsilon}_{HH} = 0.3024326101249178. \qquad (2.51)$$

де $\hat{\varepsilon}_{HH}$ – безрозмірне значення енергії зв'язку між парою атомів водню;

$\varepsilon_{HH}$ – значення енергії зв'язку між парою атомів водню, ккал/моль;

$\varepsilon_{OO}$ – значення енергії зв'язку між парою атомів кисню, ккал/моль.

Обчислення безрозмірного значення енергії зв'язку між атомом кисню та водню (2.52):

$$\hat{\varepsilon}_{OH} = \varepsilon_{OH} / \varepsilon_{OO} \Rightarrow \hat{\varepsilon}_{OH} = \frac{0.0836}{0.1521} \cdot \frac{\text{ккал/моль}}{\text{ккал/моль}} \Rightarrow$$

$$\hat{\varepsilon}_{OH} = 0.54963839579224. \qquad (2.52)$$

де $\hat{\varepsilon}_{OH}$ – безрозмірна величина енергії зв'язку між атомом кисню та атомом водню;

$\varepsilon_{OH}$ – значення енергії зв'язку між атомом кисню та атомом водню, ккал/моль;

$\varepsilon_{OO}$ – значення енергії зв'язку між парою атомів кисню, ккал/моль.

Обчислення масштабного множнику для значення часу :

$$\tilde{t} = \sqrt{\left(m_O \cdot \sigma_{OO}^2\right) / \varepsilon_{OO}} \Rightarrow$$

$$\tilde{t} = \left[ \frac{\left[15.994 \cdot 1.6605390666 \cdot 10^{-27}\right] \cdot \left[\left(3.1506\right)^2 \cdot 10^{-20}\right]}{0.1521 \cdot 4.2 \cdot 1000 / 6.022 \cdot 10^{-23}} \times \right.$$

$$\left. \times \frac{\text{кг} \cdot \text{м}^2}{\left(\text{Дж/моль}\right) \cdot \text{моль}} \right]^{1/2} \Rightarrow$$

$$\tilde{t} = \left[ \left[263.62872332292613 \cdot 10^{-47}\right] \cdot \left[0.10608103620059 \cdot 10^{-20}\right]^{-1} \times \right.$$

$$\left. \times \left[\text{кг} \cdot \text{м}^2 / \left(\text{кг} \cdot \text{м}^2 \cdot \text{с}^{-2}\right)\right] \right]^{1/2} \Rightarrow$$

$$\tilde{t} = \left[2485.1635387913 \cdot 10^{-27}\right]^{1/2} [c] \Rightarrow$$

$$\tilde{t} = \left[248.51635387913 \cdot 10^{-26}\right]^{1/2} [c] \Rightarrow$$

$$\tilde{t} = 15.764401475448 \cdot 10^{-13} [c], \qquad (2.53)$$

де $\tilde{t}$ – час, с;

$m_O$ – маса атому кисню, u;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню для рівнянь Леннард-Джонса, Å;

$\varepsilon_{OO}$ – значення енергії зв'язку між парою атомів кисню для рівняння Леннард-Джонса, ккал/моль.

Також останній результат можна записати в еквівалентній формі (2.54):

$$\tilde{t} = 1.5764401475448[nc], \quad (2.54)$$

де $n$ – піко дорівнює $10^{-12}$.

Розрахунок безрозмірного значення коефіцієнту рівнянь Кулона (2.55):

$$\hat{k}_{Кул} = k_{Кул} \cdot q_H^2 \cdot [\sigma_{OO} \cdot \varepsilon_{OO}]^{-1} \Rightarrow$$

$$\hat{k}_{Кул} = \Big[9 \cdot 10^9 \cdot \left(0.417 \cdot 1.602176487 \cdot 10^{-19}\right)^2 \times$$

$$\times \left(3.1506 \cdot 10^{-10} \cdot 0.1521 \cdot 4.2 \cdot 10^3 / 6.022 \cdot 10^{-23}\right)^{-1}\Big]$$

$$\left(\frac{\text{Н} \cdot \text{м}^2}{\text{Кл}^2} \cdot \text{Кл}^2 \cdot [\text{м} \cdot \text{Н} \cdot \text{м}]^{-1}\right) \Rightarrow$$

$$\hat{k}_{Кул} = \left[4.0173098274202 \cdot 10^{-29}\right] \cdot \left[0.3342189126536 \cdot 10^{-30}\right]^{-1} \Rightarrow$$

$$\hat{k}_{Кул} = 12.019995503916 \cdot 10^1 \Rightarrow$$

$$\hat{k}_{Кул} = 120.19995503916, \quad (2.55)$$

де $\hat{k}_{Кул}$ – безрозмірне значення коефіцієнту пропорційності рівнянь Кулона;

$k_{Кул}$ – коефіцієнт пропорційності рівнянь Кулона, $\text{Н} \cdot \text{м}^2 / \text{Кл}^2$;

$q_e$ – елементарний заряд, Кл;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню для рівнянь Леннард-Джонса, Å;

$\varepsilon_{OO}$ – енергія зв’язку між парою атомів кисню для рівнянь Леннард-Джонса, ккал/моль.

Розрахунок безрозмірної величини заряду атому водню (2.56):

$$\hat{q}_H = q_H / q_H = 1, \quad (2.56)$$

де $\hat{q}_H$ – безрозмірна величина заряду атому водню;

$q_H$ – значення заряду атому водню, Кл.

Розрахунок безрозмірної величини заряду атому кисню (2.57):

$$\hat{q}_O = q_O / q_H \Rightarrow \left(-0.834 \cdot q_e\right) / \left(0.417 \cdot q_e\right) \Rightarrow \hat{q}_O = -2, \quad (2.57)$$

де $\hat{q}_O$ – безрозмірна величина заряду атому кисню;

$q_O$ – величина заряду атому кисню Кл;

$q_e$ – величина елементарного заряду, Кл.

Обчислення значення безрозмірного коефіцієнту потенціалу зв'язку (2.58):

$$\hat{k}_{Зв} = k_{Зв} \cdot \sigma_{OO}^2 / \varepsilon_{OO} \Rightarrow$$

$$\hat{k}_{Зв} = \left[450 \cdot \left(3.1506 \cdot 10^{-10}\right)^2\right] \times$$

$$\times \left[0.1521 \cdot 4.2 \cdot 10^3 / 6.022 \cdot 10^{-23}\right]^{-1} \Rightarrow$$

$$\hat{k}_{Зв} = \left[4466.826162 \cdot 10^{-20}\right] \cdot \left[0.1060810362006 \cdot 10^{-20}\right]^{-1} \Rightarrow$$

$$\hat{k}_{Зв} = 42107.678450211. \qquad (2.58)$$

де $\hat{k}_{Зв}$ – безрозмірне значення коефіцієнту пружності зв'язку;

$k_{Зв}$ – розмірне значення коефіцієнту пружності зв'язку, кг/с$^2$;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню для рівнянь Леннард-Джонса, Å;

$\varepsilon_{OO}$ – енергія зв'язку між парою атомів кисню для рівнянь Леннард-Джонса, ккал/моль.

Розрахунок безрозмірної величини коефіцієнту кутового потенціалу (2.59):

$$\hat{k}_{Кут} = k_{Кут} / \varepsilon_{OO} \Rightarrow$$

$$\hat{k}_{Кут} = [55 / 0.1521]\left(\frac{\text{ккал}}{\text{моль}} \cdot \frac{\text{моль}}{\text{ккал}}\right) \Rightarrow$$

$$\hat{k}_{Кут} = 361.60420775805. \qquad (2.59)$$

де $\hat{k}_{Кут}$ – безрозмірна величина коефіцієнту рівняння кутового потенціалу;

$k_{Кут}$ – розмірна величина коефіцієнту кутового потенціалу, ккал/моль;

$\varepsilon_{OO}$ – енергія зв'язку між парою атомів кисню для рівнянь Леннард-Джонса, ккал/моль.

Розрахунок значення безрозмірної величини рівноважної відстані потенціалу зв'язку (2.60):

$$\hat{r}_{i,j}^{0} = r_{i,j}^{0} / \sigma_{OO} \Rightarrow \hat{r}_{i,j}^{0} = 0.957 \cdot 10^{-10} / \left(3.1506 \cdot 10^{-10}\right) \Rightarrow$$

$$\hat{r}_{i,j}^{0} = 0.30375166634926. \quad (2.60)$$

де $\hat{r}_{i,j}^{0}$ – безрозмірне значення рівноважної відстані потенціалу зв’язку;

$r_{i,j}^{0}$ – значення рівноважної відстані потенціалу зв’язку, Å;

$\sigma_{OO}$ – рівноважна відстань між парою атомів кисню для рівнянь Леннард-Джонса, Å.

# 3 РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОГО ЕКСПЕРИМЕНТУ

## 3.1 Методика чисельного експерименту

### 3.1.1 Загальний алгоритм проведення комп'ютерного моделювання динаміки системи моделей молекул

Загальний алгоритм проведення комп'ютерного експерименту, який проводиться в межах даної роботи, являє собою наступну послідовність дій:

1. Визначення розмірів модельованої області у тривимірному віртуальному просторі, тобто встановлення *ширини*, *довжини* та *висоти* абстрактного обмежуючого контейнеру.
2. Введення значення кількості моделей молекул у програму.
3. Обчислення початкових значень координат у віртуальному просторі та швидкостей для кожної моделі молекули.
4. Розрахунок динаміки системи у часі. У якості критерію зупинки можна використати штучне значення максимуму часу. Даний максимум часу моделювання визначається експериментатором до початку самого процесу моделювання. [5,11]

### 3.1.2 Етап визначення розмірів віртуального обмежуючого контейнеру модельованої області

Визначення розмірів обмежуючого контейнеру модельованої області відбувається шляхом запиту до користувача. Введені значення з клавіатури проходять етап перевірки: якщо хоча б одне зі значень, що були введені є меншим за розмір мінімальної області моделювання, то такі значення не зберігаються, а користувачу надається повторна можливість ввести нові значення. У випадку проходження етапу перевірки – введені значення

зберігаються у спеціально відведені змінні програми. Відповідний код програми можна побачити в розділі ДОДАТОК Ф.

### 3.1.3 Етап визначення кількості моделей молекул

Кількість моделей молекул визначається експериментатором через запит програми про кількість молекул. На екрані з'явиться запрошення для введення кількості моделей молекул, а також значення максимальної кількості молекул, що можливо помістити в модельовану область, параметри якої вже було збережено в програмі. У разі перевищення доступного ліміту буде відображено попередження та надано ще одну спробу введення значення кількості молекул у систему. Відповідний програмний код можна побачити в розділі ДОДАТОК Ф.

### 3.1.4 Етап обчислення початкових значень координат моделей молекул

Даний етап складається з декількох послідовних кроків.

**Крок №1.** Визначення *базових* координат атомів молекули в тривимірному просторі.

У даній роботі вибрано наступний шлях розрахунку *базових* координат атомів молекули води. Перший атом отримує наступні значення координат:

- компонента *OX*: значення безрозмірної рівноважної відстані потенціалу зв'язку $\left(\hat{r}^{0}_{i,j}\right)$ формула (2.60);
- компонента *OY*: 0.0;
- компонента *OZ*: 0.0.

Другий атом отримує такі значення:

- компонента *OX*: 0.0;
- компонента *OY*: 0.0;
- компонента *OZ*: 0.0.

Координати третього атому розраховуються за допомогою матриці повороту навколо осі $OZ$ на необхідний кут $\theta_0 = 104.52$. У даному випадку використано матрицю повороту проти годинникової стрілки (3.1):

$$RotXY = \begin{bmatrix} \cos(\theta_0) & -\sin(\theta_0) \\ \sin(\theta_0) & \cos(\theta_0) \end{bmatrix}, \tag{3.1}$$

де $RotXY$ – умовне позначення для матриці повороту навколо осі $OZ$ проти годинникової стрілки на кут $\theta_0$.

Відповідно до формули (3.1) координати для третього атому можна розрахувати таким чином:

- компонента $OX$ (3.2):

$$a_3^x = \cos(\theta_0) \cdot a_1^x + \left(-\sin(\theta_0)\right) \cdot a_1^y, \tag{3.2}$$

де $a_3^x$ – компонента $OX$ координати третього атому;

$\theta_0$ – значення рівноважного кута відповідно кутового потенціалу;

$a_1^x$ – компонента $OX$ координати першого атому;

$a_1^y$ – компонента $OY$ координати першого атому.

- компонента $OY$:

$$a_3^y = \sin(\theta_0) \cdot a_1^x + \cos(\theta_0) \cdot a_1^y, \tag{3.3}$$

де $a_3^y$ – компонента $OY$ координати третього атому;

$\theta_0$ – значення рівноважного кута відповідно кутового потенціалу;

$a_1^x$ – компонента $OX$ координати першого атому;

$a_1^y$ – компонента $OY$ координати першого атому.

- компонента $OZ$: 0.0.

Таким чином визначено «базові» координати моделі молекули води, які можна використовувати для того, щоб у подальшому розташовувати моделі молекули у віртуальному просторі з вірною конфігурацією атомів.

**Крок 2.** Визначення підпростору для розміщення в його центрі молекули перед початком моделювання динаміки системи та розрахунку оцінки

максимальної кількості моделей молекул, які можна розмістити в заданому просторі.

Габаритні розміри підпростору вираховуються наступним чином:

1. Обчислюються координати центральної точки моделі молекули з базовими координатами, що було обчислено на **кроці 1** даного алгоритму.
2. Обчислюються відстані від центральної точки моделі до кожного атома моделі молекули. Визначається абсолютне максимальне значення.
3. Розраховуються розміри підпростору шляхом подвоєння максимальної відстані, що була вирахувана в пункті 2 і додається деяка додатна константа, що буде грати роль відступу (щоб гарантувати, що моделі молекул не будуть накладатися одна на одну).

Оцінка ж кількості моделей молекул, яка може поміститися в задану модельовану область розраховується шляхом обчислення добутку відношень відповідних сторін заданої модельованої області та розрахованої під області.

**Крок 3.** Виконуємо обхід усього простору моделювання за допомогою підпростору, розміри якого було розраховано раніше. І розміщуємо усі необхідні моделі молекул у центрах підпросторів.

**Крок 3.1.** Визначаємо положення підпростору на основі кількості підпросторів вздовж кожного додатного напрямку осі віртуального простору. Для обчислення координат у просторі поточного підпростору використовуються наступні формули. Розрахунок індексів під областей вздовж осі *OX* (3.4):

$$subarea_i^x = mol_num \cdot \mathrm{mod}\left(subareas_along_OX\right), \qquad (3.4)$$

де $subarea_i^x$ – індекс під області вздовж осі *OX* віртуальної області моделювання;

$mol_num$ – поточний номер моделі молекули (у діапазоні $mol_num \cdot \in \left\{0,1,...,\left(N-1\right)\right\}$);

$N$ – кількість молекул у системі, що була задана користувачем;

$subareas_along_OX$ – кількість підпросторів, що можна послідовно розмістити вздовж осі *OX*.

Розрахунок індексів під областей вздовж осі *OY* (3.5):

$$subarea_i^y = mol_num \cdot \mathrm{mod}\left(subareas_along_OY\right), \qquad (3.5)$$

де $subarea_i^Y$ – індекс під області вздовж осі *OY* віртуальної області моделювання;

$mol_num$ – поточний номер моделі молекули (у діапазоні $mol_num \cdot \in \left\{0,1,...,\left(N-1\right)\right\}$);

$N$ – кількість молекул у системі, що була задана користувачем;

$subareas_along_OY$ – кількість підпросторів, що можна послідовно розмістити вздовж осі *OY*.

Розрахунок індексів під областей вздовж осі *OZ* (3.6):

$$subarea_i^z = mol_num \cdot \mathrm{mod}\left(subareas_along_OZ\right), \qquad (3.6)$$

де $subarea_i^z$ – індекс під області вздовж осі *OZ* віртуальної області моделювання;

$mol_num$ – поточний номер моделі молекули (у діапазоні $mol_num \cdot \in \left\{0,1,...,\left(N-1\right)\right\}$);

$N$ – кількість молекул у системі, що була задана користувачем;

$subareas_along_OZ$ – кількість підпросторів, що можна послідовно розмістити вздовж осі *OZ*.

Координати під області можна знайти за допомогою множення знайдених індексів та наступних індексів (на одиницю більше від обрахованих) на відповідні розмірності під області вздовж осей *OX*, *OY* та *OZ*.

**Крок 3.2.** Обчислюємо центральну точку поточної під області шляхом простого усереднення відомих координат під області вздовж кожної з осей віртуального простору.

**Крок 3.3.** Розраховуємо вектор від центральної точки моделі молекули з «базовими» координатами до центральної точки поточного підпростору. Це просто різниця між відповідними компонентами центральної точки поточного підпростору та центральної точки моделі з базовими координатами.

**Крок 3.4.** Координати поточної моделі молекули формуються шляхом сумування відповідних компонент моделі молекули з базовими координатами та компонентами вектору, що розрахований на попередньому **кроці 3.3**.

Відповідний код програми для розрахунку початкових координат можна знайти в розділі ДОДАТОК С.

### 3.1.5 Етап обчислення початкових значень швидкостей моделей молекул

У даній роботі вирішено заповнити початкові значення швидкостей моделей молекул нулями.

### 3.1.6 Етап безпосереднього моделювання динаміки системи моделей молекул води

Алгоритм моделювання динаміки системи полягає в наступних кроках:

1. Після визначення початкових координат та швидкостей усіх частинок системи визначаються сумарна сила, що діє на кожну окрему частинку системи, за допомогою рівнянь, що були визначені в попередніх розділах, та відповідні значення прискорень.
2. На основі поточних значень координат, швидкостей та прискорень обчислюється нові значення координат кожної моделі атому використовуючи рівняння (1.15). А також частково визначаються нові значення швидкостей кожної частинки з використанням поточних значень прискорень (відповідно до формули (1.16)).
3. На основі нових значень координат визначаються сили та прискорення частинок.
4. Використовуючи нові значення прискорень знаходимо остаточні значення швидкостей кожної моделі відповідно до формули (1.16).

Код програми, що відповідає даному етапу знаходиться в розділі ДОДАТОК Т.

## 3.2 Алгоритм розрахунку значення радіусу відсікання

У даній роботі пропонується наступна ідея для розрахунку значення радіусу відсікання, а саме розрахувати середнє значення за кількістю кроків за часом суми середніх значень енергії системи за кількістю частинок.

Слід зазначити, що енергією системи буде сума потенціальної енергії системи та кінетичної енергії системи відповідно до формули (3.7):

$$E_{сист} = E_{p,сист} + E_{k,сист}, \tag{3.7}$$

де $E_{сист}$ – повна енергія системи, Дж;

$E_{p,сист}$ – кінетична енергія системи, Дж;

$E_{k,сист}$ – потенціальна енергія системи, Дж. [10]

У свою чергу $E_{p,сист}$ має вигляд (3.8):

$$E_{p,сист} = U_{Л-Дж} + U_{Кул} + U_{Кут} + U_{Зв}, \tag{3.8}$$

де $U_{Л-Дж}$ – це сума всіх значень потенціалів Леннард-Джонса в системі, Дж;

$U_{Кул}$ – сума всіх значень потенціалів Кулона в системі, Дж;

$U_{Кут}$ – сума всіх значень кутових потенціалів у системі, Дж;

$U_{Зв}$ – сума всіх значень потенціалів зв'язку в системі, Дж.

А кінетична енергія системи частинок визначається за формулою (3.9):

$$E_{k,сист} = \frac{m_i \cdot v_i^2}{2}, \tag{3.9}$$

де $m_i$ – це маса частинки з номером $i$, кг;

$v_i$ – швидкість частинки з номером $i$, м/с.

Алгоритм реалізації ідеї, що викладено вище полягає в наступних кроках:

1. Визначається кількість кроків за часу, яку буде проводитися моделювання динаміки системи; кількість моделей молекул та розміри модельованої області. А також підготовлюється набір значень радіусів відсікання, які будуть використані для визначення оптимального.
2. Виконується моделювання динаміки системи частинок без використання радіусу відсікання, тобто під час обчислення сил, що діють на кожну окрему частинку використовується всі наявні частинки. Контрольне значення середньої енергії за кількістю частинок, що усереднено за кількістю кроків моделювання можна зберегти в окремому файлі для подальшої обробки.
3. Виконуються моделювання динаміки системи частинок з використанням радіусу відсікання, тобто під час обчислення значень сили, що діють на кожну окрему частинку враховуються лише ті частинки, що знаходяться на відстані яка є меншою або рівною значенню радіусу відсікання. Такі моделювання проводяться для всього набору значень радіусу відсікання, що було підготовано на кроці 1. Контрольні значення можна зберігати в окремому фалі для подальшої обробки.
4. Після виконання моделювань динаміки системи частинок для всіх випадків з пунктів 2 та 3, оптимальне значення радіусу відсікання можна визначити за допомогою збережених контрольних значень наступним чином: можна розрахувати різниці між першим контрольним значенням, яке відповідає комп’ютерному експерименту без урахування радіусу відсікання, тобто еталонне значення параметру, усіма іншими наявними значеннями. Найбільш підходящому значенню радіуса відсікання буде відповідати найближче до нуля значення різниці.

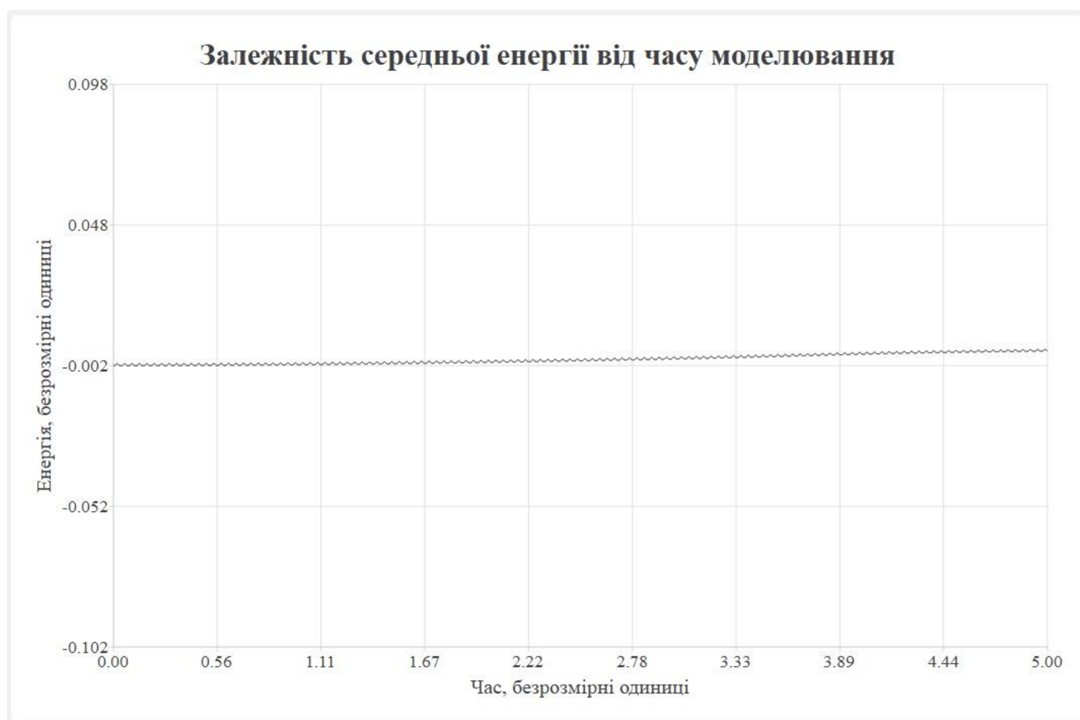
Відповідний код знаходиться в розділі ДОДАТОК Д.

## 3.3 Результати розрахунків

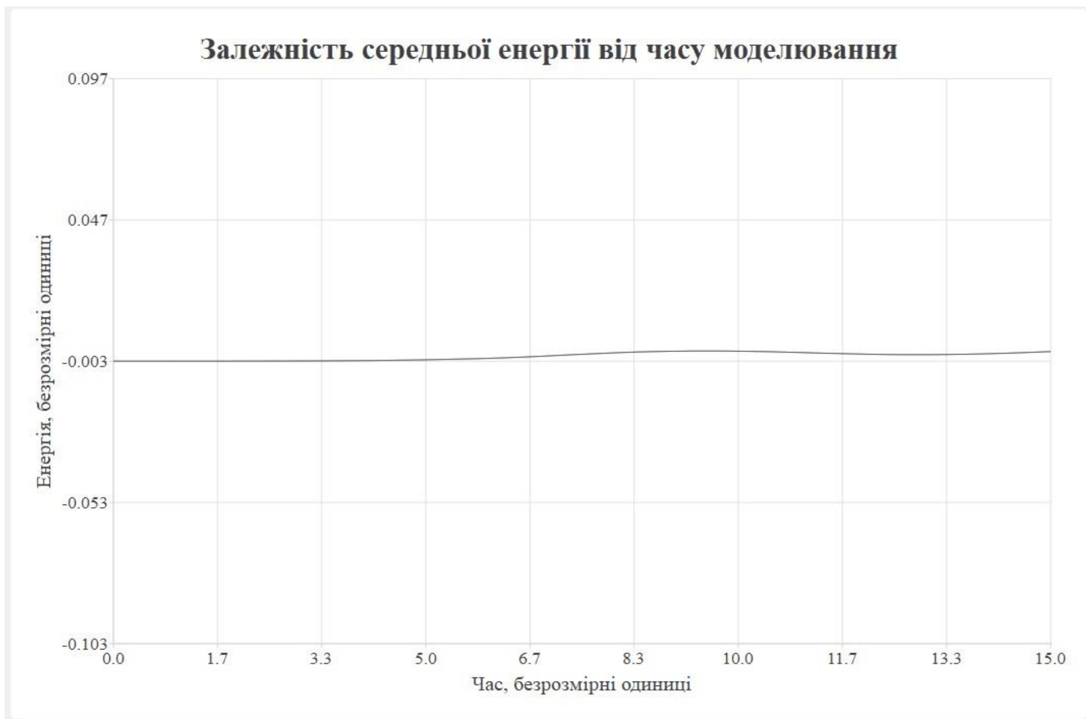
Під час виконання даної роботи було проведено серію комп'ютерних експериментів та отримано наступні результати:

1. Графіки залежності середньої енергії системи частинок від значення часу моделювання.
2. Таблиці зі контрольними значеннями середньої за кількістю атомів у системі енергії системи частинок, що в свою чергу усереднено за кількістю кроків моделювання.

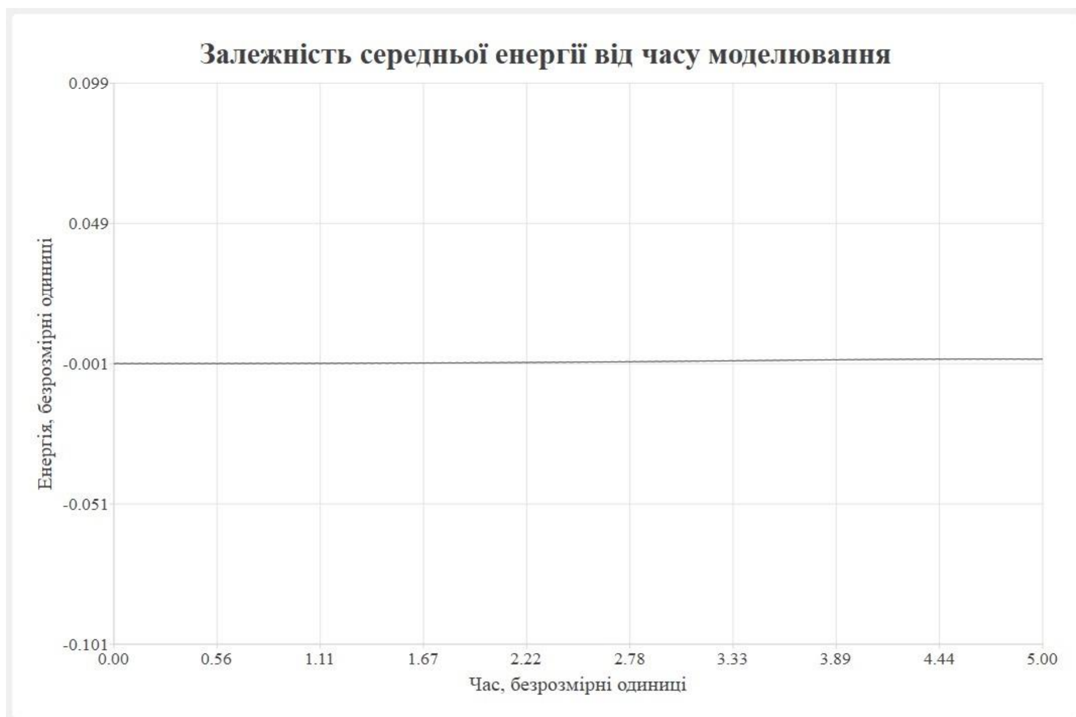
Щодо залежності середньої енергії системи на одну частинку було отримано наступні графіки (рисунок 3.1 (а-г)):



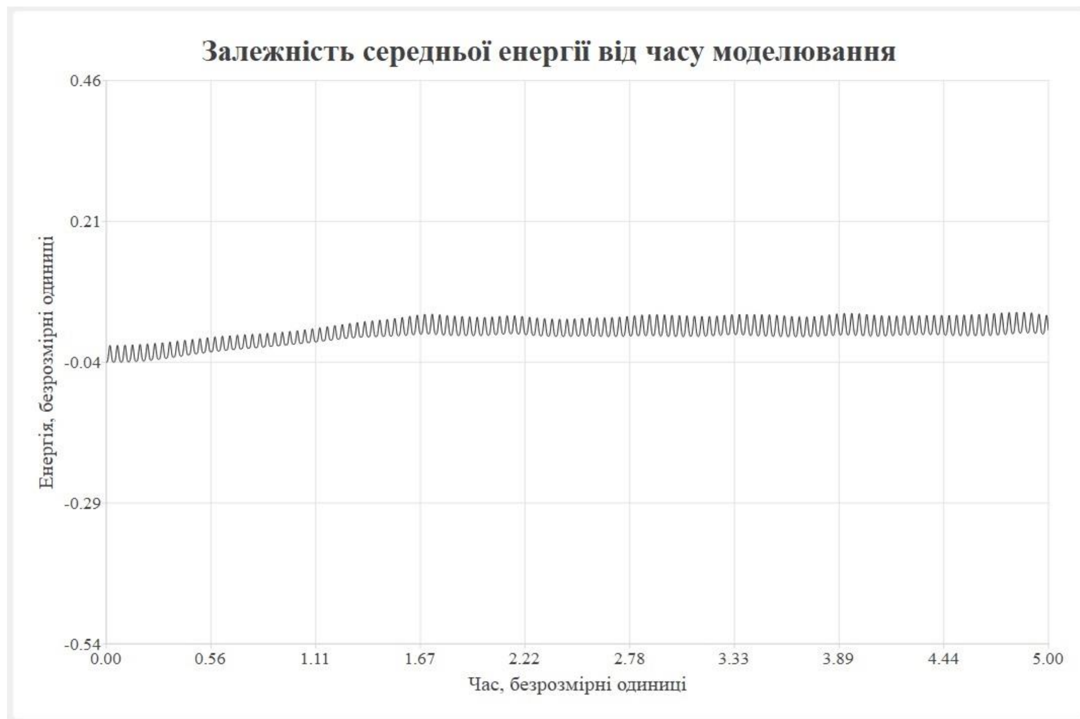

а)




б)




в)




г)

Графік, що відповідає рисунку **а)** отримано на основі таких параметрів: *розмір контейнеру* – 200 м.о. × 200 м.о. × 200 м.о., де м.о. – модельні одиниці; *кількість моделей молекул* – 150; *крок за часом* – 0.001; *діапазон часу моделювання* – [0.0; 5.0].

Графік, що відповідає рисунку **б)** побудовано за допомогою таких параметрів: *розмір контейнеру* – 610 м.о. × 610 м.о. × 610 м.о.; *кількість моделей молекул* – 100; *крок за часом* – 0.001; *діапазон часу моделювання* – [0.0; 15.0].

Графік, що відповідає рисунку **в)** отримано з використанням наступного набору параметрів: *розмір контейнеру* – 999 м.о. × 999 м.о. × 999 м.о.; *кількість моделей молекул* – 100; *крок за часом* – 0.001; *діапазон часу моделювання*: [0.0; 5.0].

Графік, що відповідає рисунку **г)** побудовано на основі таких параметрів: *розмір контейнеру* – 100.0-100.0-100.0; *кількість моделей молекул* – 200; *крок за часом* – 0.0005; *діапазон часу моделювання* – [0.0; 5.0].

*Рисунок3.1 – вид залежності середньої енергії системи від часу моделювання для різних конфігурацій а) – г)*

Опираючись на вигляд отриманих кривих можна сказати, що похибка, яка є в даній моделі вносить невеликий вклад в енергію системи частинок.

Таблиці, що отримано в результаті проведення комп’ютерного експерименту для розрахунку оптимального значення радіусу відсікання можна побачити в розділах ДОДАТОК Х та ДОДАТОК Ц.

На основі отриманих значень можна сказати, що значення радіусу відсікання, який зазвичай пропонується в літературних джерелах не дає бажаного результату.

# ВИСНОВКИ

У даній кваліфікаційній роботі було побудовано математичну модель трьохатомної молекули, визначено її структуру та вихідні параметри. А також отримано набір рівнянь, за допомогою яких описується динаміка руху моделі молекули: рівняння руху Ньютона для визначення прискорення частинок системи, набір потенціалів, для визначення сил, що діють у модельованій області та чисельна схема Верле для розрахунку координат і швидкостей моделей молекул.

Було визначено, що для системи взаємодіючих гнучких куль ефективне значення лежить у діапазоні $[4\sigma;6\sigma]$, на противагу значенню $2\sigma$, що пропонується в літературних джерелах.

Було проведено серію комп'ютерних експериментів для визначення коректних початкових параметрів: кількості моделей молекул та розмірів віртуальної області, у межах якої проводиться моделювання. Під «коректними параметрами» у даному випадку слід розуміти такі, за яких енергія системи не зростає занадто швидкими темпами, через особливості комп'ютерних обчислень та наявності чисельного методу в описі руху моделей молекул повна відсутність похибки не є можливою.

Також було проведено комп'ютерні експерименти, метою яких було визначення оптимального значення радіусу для системи моделей молекул. Результати цих експериментів свідчать про те, що для такої моделі системи, яка була побудована в даній кваліфікаційній роботі, неможливо використовувати оцінку радіусу відсікання, яка пропонується в тематичній літературі ($2\sigma$). Однією з основних причин такого висновку є той факт, що в запропонованій моделі взаємодіючих систем частинок присутня сила Кулона, що відноситься до сил, що діють відносно великій відстані у порівнянні з, наприклад, силами, що описуються за допомогою потенціалу Леннард-Джонса, що змушує враховувати взаємозв'язки на більшій відстані.

# ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. ДСТУ 3008:2015. Звіти у сфері науки і техніки. Структура та правила оформлювання. [На заміну ДСТУ 3008-95; чинний від 2017-07-01]. Вид. офіц. Київ: Державне підприємство «Український науково-дослідний і навчальний центр проблем стандартизації, сертифікації та якості», 2016. 31 с.
2. Зубко И.Ю. Математическое моделирование: дискретные подходы и численные методы : : учеб. Пособие / И.Ю. Зубко, Н.Д. Няшина. – Пермь : Изд-во Перм. нац. исслед. политехн. ун-та, 2012. – 365 с.
3. ДСТУ ГОСТ 7.1:2006. Бібліографічний запис, бібліографічний опис. Загальні вимоги та правила складання : метод. Рекомендації з впровадження / уклали: Галевич О. К., Штогрин І. М. – Львів, 2008. – 20 с.
4. Кривцов А. М., Кривцова Н. В. Метод частиц и его использование в механике деформируемого твердого тела. *Дальневосточный математический журнал ДВО РАН*. 2002, Т. 3, № 2, с. 254-276.
5. Rapaport D. C. The Art of Molecular Dynamics Simulation; Cambridge University Press: Cambridge 2004. – 564 с.
6. Allen M. P. Introduction to Molecular Dynamics Simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins.* 2004, Vol. 23. p. 1-28.
7. Eom K. Simulations in Nanobiotechnology; CRC Press: Boca Raton, FL, 2012. – p 552.
8. Dias R. A., Rapini M., Costa B. V., Coura P. Z. Temperature Dependent Molecular Dynamic Simulation of Friction. *Brazilian Journal of Physics*. 2006, vol. 36, no. 3A, p. 741-745.
9. Griebel M., Knapek S. Zumbusch G. Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications; Springer Berlin, 2007. – p. 472.
10. Фізика: Конспект лекцій / Укладач О.В. Лисенко. – Суми: Вид-во СумДУ,

2010. – Ч.1. – 199 с.

11. Неелов И. М. Введение в молекулярное моделирование биополимеров СПб: НИУ ИТМО, 2014. – 101 с.
12. Калорія // Wikipedia : веб-сайт. URL: https://uk.wikipedia.org/wiki/Калорія (дата звернення: 24.11.20).
13. Ангстем // Wikipedia : веб-сайт. URL: https://uk.wikipedia.org/wiki/Ангстрем (дата звернення: 15.11.20).
14. Non-SI units mentioned on the SI // Wikipedia : веб-сайт. URL: https://en.wikipedia.org/wiki/Non-SI_units_mentioned_in_the_SI (дата звернення: 24.11.20).

# ДОДАТОК А

Лістинг програми, що дозволяє будувати двовимірні графіки та зберігати їх у вигляді файлів зображень .png, мовою C++. Текст файлу main.cpp , що містить функцію main

```
#include <iostream>

#include <qapplication.h>
#include <QtWidgets/qmainwindow.h>

#include "user_chart_wrapper.h"

// Function prototypes
// ---------------------------------------------------------------------------
void lennard_jones_potential
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
);

void lennard_jones_force
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
);

void coulomb_potential
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
);

void coulomb_force
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
);

void harmonic_potential
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
);
```

```
void harmonic_force
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
);

void mie_potential
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
);

void mie_force
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
);
// -----------------------------------------------------------------------------


int main(int argc, char *argv[])
{
	// Creating application manager
	QApplication app(argc, argv);


	// Building and saving picture of Lennard-Jones Potential
	{
		QMainWindow window;
		User_chart_wrapper plot_obj
		{
			"00_Poten_Lennard-Jones",
			"Потенціал Леннард-Джонса",
			"Відстань, умов. од.",
			"Енергія, умов. од.",
			lennard_jones_potential,
			3.0, 6.0, 0.01,
			10, 10,
			"png"
		};
		window.setCentralWidget(plot_obj.chart_view);
		window.resize(640, 380);

		std::cout << "Lennard-Jones Potential graph is built!" << '\n';
	}

	// Building and saving picture of Lennard-Jones Force
	{
		QMainWindow window;
		User_chart_wrapper plot_obj
		{
			"01_Force_Lennard-Jones",
			"Сила Леннард-Джонса",
			"Відстань, умов. од.",
```

```
                "Сила, умов. од",
                lennard_jones_force,
                3.5, 6.0, 0.01,
                10, 10,
                "png"
        };
        window.setCentralWidget(plot_obj.chart_view);
        window.resize(640, 380);

        std::cout << "Lennard-Jones Force graph is built!" << '\n';
}

// Building and saving picture of Coulomb Potential
{
        QMainWindow window;
        User_chart_wrapper plot_obj
        {
                "02_Poten_Coulomb",
                "Потенціал Кулона",
                "Відстань, умов. од.",
                "Енергія, умов. од.",
                coulomb_potential,
                3.0, 100.0, 0.01,
                10, 10,
                "png"
        };
        window.setCentralWidget(plot_obj.chart_view);
        window.resize(640, 380);

        std::cout << "Coulomb Potential graph is built!" << '\n';
}

// Building and saving picture of Coulomb Force
{
        QMainWindow window;
        User_chart_wrapper plot_obj
        {
                "03_Force_Coulomb",
                "Сила Кулона",
                "Відстань, умов. од.",
                "Сила, умов. од.",
                coulomb_force,
                3.0, 100.0, 0.01,
                10, 10,
                "png"
        };
        window.setCentralWidget(plot_obj.chart_view);
        window.resize(640, 380);

        std::cout << "Coulomb Force graph is built!" << '\n';
}

// Building and saving picture of Hook's Law (Harmonic Potential)
{
        QMainWindow window;
        User_chart_wrapper plot_obj
        {
                "04_Poten_Harmonic",
                "Гармонічний потенціал",
                "Відстань, умов. од.",
                "Енергія, умов. од.",
                harmonic_potential,
                0.0, 6.0, 0.01,
                10, 10,
```

```
			"png"
		};
		window.setCentralWidget(plot_obj.chart_view);
		window.resize(640, 380);

		std::cout << "Harmonic Potential graph is built!" << '\n';
	}

	// Building and saving picture of Harmonic Force
	{
		QMainWindow window;
		User_chart_wrapper plot_obj
		{
			"05_Force_Harmonic",
			"Гармонічна сила",
			"Відстань, умов. од.",
			"Сила, умов. од.",
			harmonic_force,
			0.0, 6.0, 0.01,
			10, 10,
			"png"
		};
		window.setCentralWidget(plot_obj.chart_view);
		window.resize(640, 380);

		std::cout << "Harmonic Force graph is built!" << '\n';
	}

	// Building and saving picture of Mie Potential
	{
		QMainWindow window;
		User_chart_wrapper plot_obj
		{
			"06_Poten_Mie",
			"Потенціал Мі",
			"Відстань, умов. од.",
			"Енергія, умов. од.",
			mie_potential,
			2.5, 9.0, 0.01,
			10, 10,
			"png"
		};
		window.setCentralWidget(plot_obj.chart_view);
		window.resize(640, 380);

		std::cout << "Mie Potential graph is built!" << '\n';
	}

	// Building and saving picture of Mie Force
	{
		QMainWindow window;
		User_chart_wrapper plot_obj
		{
			"07_Force_Mie",
			"Сила Мі",
			"Відстань, умов. од.",
			"Сила, умов. од.",
			mie_force,
			3.0, 9.0, 0.01,
			10, 10,
			"png"
		};
		window.setCentralWidget(plot_obj.chart_view);
		window.resize(640, 380);
```

```
        std::cout << "Mie Force graph is built!" << '\n';
    }

    return 0;
}


// Lennard-Jones: Potential
void lennard_jones_potential
(
    std::vector<double> * x_axis_data,
    std::vector<double> * y_axis_data,
    double left_x_limit,
    double right_x_limit,
    double x_step
)
{
    // Local variables defining
    // -------------------------------------------------------------------------
    // Dimensionless parameters for gas that was named Argon

    // the real value of equilibrium_distance = equilibrium_distance * 10^{-10}
    double equilibrium_distance = 3.4;

    // the real value of link energy = link_energy * 10^{-21}
    double link_energy = 1.65;

    // the real malue of atom mass = atom_mass * 10^{-26}
    double atom_mass = 6.69;
    // -------------------------------------------------------------------------

    // left_x_limit == current distance between couple of atoms
    while (left_x_limit < right_x_limit)
    {
        // Inserting Y value
        // ---------------------------------------------------------------------
        y_axis_data->push_back
        (
            4.0 * link_energy * (
                pow((equilibrium_distance / left_x_limit), 12) -
                pow((equilibrium_distance / left_x_limit), 6))
        );
        // ---------------------------------------------------------------------

        // Inserting X Value
        x_axis_data->push_back(left_x_limit);

        // Move right left X limit value
        left_x_limit += x_step;
    }

    return;
}


// Lennard-Jones: Force
void lennard_jones_force
(
    std::vector<double> * x_axis_data,
    std::vector<double> * y_axis_data,
    double left_x_limit,
    double right_x_limit,
    double x_step
```

```
)
{
	// Local variables defining
	// ------------------------------------------------------------------------
	// Dimensionless parameters for gas that was named Argon

	// the real value of equilibrium_distance = equilibrium_distance * 10^{-10}
	double equilibrium_distance = 3.4;

	// the real value of link energy = link_energy * 10^{-21}
	double link_energy = 1.65;

	// the real malue of atom mass = atom_mass * 10^{-26}
	double atom_mass = 6.69;
	// ------------------------------------------------------------------------

	// left_x_limit == current distance between couple of atoms
	while (left_x_limit < right_x_limit)
	{
		// Inserting Y value
		// --------------------------------------------------------------------
		y_axis_data->push_back
		(
			24.0 * link_energy / equilibrium_distance * (
				pow((equilibrium_distance / left_x_limit), 13) * 2.0 -
				pow((equilibrium_distance / left_x_limit), 7))
		);
		// --------------------------------------------------------------------

		// Inserting X Value
		x_axis_data->push_back(left_x_limit);

		// Move right left X limit value
		left_x_limit += x_step;
	}

	return;
}


// Coulomb: Potential
void coulomb_potential
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
)
{
	// Local variables defining
	// ------------------------------------------------------------------------
	// Dimensionless parameters for gas that was named Argon

	// atom's charge values
	double charge_1 = 1.0 * pow(10.0, -4);
	double charge_2 = -1.0 * pow(10.0, -4);

	// coefficient k = 9 * 10^{9}
	double coefficient_k = 9.0 * pow(10, 9);
	// ------------------------------------------------------------------------

	// left_x_limit == current distance between couple of atoms
	while (left_x_limit < right_x_limit)
```

```
	{
		// Inserting Y value
		// --------------------------------------------------------------------
		y_axis_data->push_back
		(
			-coefficient_k * (charge_1 * charge_2 / left_x_limit)
		);
		// --------------------------------------------------------------------

		// Inserting X Value
		x_axis_data->push_back(left_x_limit);

		// Move right left X limit value
		left_x_limit += x_step;
	}

	return;
}


// Coulomb: Force
void coulomb_force
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
)
{
	// Local variables defining
	// --------------------------------------------------------------------
	// Dimensionless parameters for gas that was named Argon

	// atom's charge values
	double charge_1 = 1.0 * pow(10, -4);
	double charge_2 = -1.0 * pow(10, -4);

	// coefficient k = 9 * 10^{9}
	double coefficient_k = 9.0 * pow(10, 9);
	// --------------------------------------------------------------------

	// left_x_limit == current distance between couple of atoms
	while (left_x_limit < right_x_limit)
	{
		// Inserting Y value
		// --------------------------------------------------------------------
		y_axis_data->push_back
		(
			coefficient_k * (charge_1 * charge_2 / (pow(left_x_limit, 2)))
		);
		// --------------------------------------------------------------------

		// Inserting X Value
		x_axis_data->push_back(left_x_limit);

		// Move right left X limit value
		left_x_limit += x_step;
	}

	return;
}
```

```
// Hook's Law: (Harmonic Potential)
void harmonic_potential
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
)
{
	// Local variables defining
	// ------------------------------------------------------------------------
	// Dimensionless parameters for gas that was named Argon

	double equilibrium_distance = 3.0;
	double coefficient_k = 1.1;
	// ------------------------------------------------------------------------

	// left_x_limit == current distance between couple of atoms
	while (left_x_limit < right_x_limit)
	{
		// Inserting Y value
		// --------------------------------------------------------------------
		y_axis_data->push_back
		(
			0.5*coefficient_k*pow(left_x_limit - equilibrium_distance, 2)
		);
		// --------------------------------------------------------------------

		// Inserting X Value
		x_axis_data->push_back(left_x_limit);

		// Move right left X limit value
		left_x_limit += x_step;
	}

	return;
}


// Hook's Law: (Harmonic Force)
void harmonic_force
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
)
{
	// Local variables defining
	// ------------------------------------------------------------------------
	double equilibrium_distance = 3.0;
	double coefficient_k = 1.1;
	// ------------------------------------------------------------------------

	// left_x_limit == current distance between couple of atoms
	while (left_x_limit < right_x_limit)
	{
		// Inserting Y value
		// --------------------------------------------------------------------
		y_axis_data->push_back
		(
			-coefficient_k*(left_x_limit - equilibrium_distance)
```

```
        );
        // ----------------------------------------------------------------------

        // Inserting X Value
        x_axis_data->push_back(left_x_limit);

        // Move right left X limit value
        left_x_limit += x_step;
    }

    return;
}


// Mie: Potential
void mie_potential
(
    std::vector<double> * x_axis_data,
    std::vector<double> * y_axis_data,
    double left_x_limit,
    double right_x_limit,
    double x_step
)
{
    // Local variables defining
    // ----------------------------------------------------------------------
    // Dimensionless parameters for gas that was named Argon

    // the real value of equilibrium_distance = equilibrium_distance * 10^{-10}
    double equilibrium_distance = 3.4;

    // the real value of link energy = link_energy * 10^{-21}
    double link_energy = 1.65;

    // the real malue of atom mass = atom_mass * 10^{-26}
    double atom_mass = 6.69;

    // addition parameters
    double n = 9.0, m = 4.0;
    // ----------------------------------------------------------------------

    // left_x_limit == current distance between couple of atoms
    while (left_x_limit < right_x_limit)
    {
        // Inserting Y value
        // ----------------------------------------------------------------------
        y_axis_data->push_back
        (
            (link_energy/(n - m))*
            (m*(pow(equilibrium_distance / left_x_limit, n)) -
             n*(pow(equilibrium_distance / left_x_limit, m)))
        );
        // ----------------------------------------------------------------------

        // Inserting X Value
        x_axis_data->push_back(left_x_limit);

        // Move right left X limit value
        left_x_limit += x_step;
    }

    return;
}
```

```
// Mie: Force
void mie_force
(
	std::vector<double> * x_axis_data,
	std::vector<double> * y_axis_data,
	double left_x_limit,
	double right_x_limit,
	double x_step
)
{
	// Local variables defining
	// ------------------------------------------------------------------------
	// Dimensionless parameters for gas that was named Argon

	// the real value of equilibrium_distance = equilibrium_distance * 10^{-10}
	double equilibrium_distance = 3.4;

	// the real value of link energy = link_energy * 10^{-21}
	double link_energy = 1.65;

	// the real malue of atom mass = atom_mass * 10^{-26}
	double atom_mass = 6.69;

	// addition parameters
	double n = 9.0, m = 4.0;
	// ------------------------------------------------------------------------

	// left_x_limit == current distance between couple of atoms
	while (left_x_limit < right_x_limit)
	{
		// Inserting Y value
		// --------------------------------------------------------------------
		y_axis_data->push_back
		(
			(n * m * link_energy / ((n - m)*equilibrium_distance))*
			(pow(equilibrium_distance / left_x_limit, n + 1) -
			 pow(equilibrium_distance / left_x_limit, m + 1))
		);
		// --------------------------------------------------------------------

		// Inserting X Value
		x_axis_data->push_back(left_x_limit);

		// Move right left X limit value
		left_x_limit += x_step;
	}

	return;
}
```

# ДОДАТОК Б

Лістинг програми, що дозволяє будувати двовимірні графіки та зберігати їх у вигляді файлів зображень *.png*, мовою C++. Текст файлу *user_chart_wrapper.h*, що містить опис класу *User_chart_wrapper*, за допомогою якого будуються графіки

```
#ifndef USER_CHART_WRAPPER
#define USER_CHART_WRAPPER

#include <QtCharts/qchart.h>
#include <QtCharts/qchartview.h>
#include <QtCharts/qlineseries.h>
#include <QtCharts/qvalueaxis.h>

#include <string>
#include <vector>
#include <cassert>
#include <iostream>


class User_chart_wrapper
{
public:
	// Chart Elements Section
	// -------------------------------------------------------------------------
	// Qt5 Chart View Object
	QtCharts::QChartView * chart_view = Q_NULLPTR;

	// X-axis
	// Ticks per axis
	int ticks_per_x_axis = 10;
	// Limits
	double left_x_limit = 0.0;
	double right_x_limit = 1.0;
	double x_step = 0.1;

	// Y-axis
	// Ticks per axis
	int ticks_per_y_axis = 10;
	// Limits
	double left_y_limit = 0.0;
	double right_y_limit = 1.0;
	// -------------------------------------------------------------------------

	User_chart_wrapper
	(
		char * file_name,
		char * chart_name,
		char * x_axis_label,
		char * y_axis_label,
		void (*get_values_func)
			(
				std::vector<double> * x_axis_data,
				std::vector<double> * y_axis_data,
				double left_x_limit,
				double right_x_limit,
				double x_step
```

```
                    ),
            double left_x_limit=0.0,
            double right_x_limit=1.0,
            double x_step=0.1,
            int ticks_per_x_axis=10,
            int ticks_per_y_axis=10,
            char * file_extension="png"
        );

        ~User_chart_wrapper();

private:
        // Chart Elements Section
        // -------------------------------------------------------------------------
        // Qt5 Chart Object
        QtCharts::QChart * chart = Q_NULLPTR;
        // Qt5 Series Object (for data storing)
        QtCharts::QLineSeries * series = Q_NULLPTR;
        // Qt5 Pen Object (for line drawing)
        QPen * pen = Q_NULLPTR;
        // Qt5 QValue Axis
        QtCharts::QValueAxis * axis_x = Q_NULLPTR;
        QtCharts::QValueAxis * axis_y = Q_NULLPTR;
        // Qt5 Fonts
        QFont * title_font = Q_NULLPTR;
        QFont * axes_font = Q_NULLPTR;

        // Function pointer for data calculating
        void(*get_values_func)
                (
                        std::vector<double> * x_axis_data,
                        std::vector<double> * y_axis_data,
                        double left_x_limit,
                        double right_x_limit,
                        double x_step
                )  = nullptr;

        // Chart labels
        char * chart_name = "Simple line chart";
        char * x_axis_label = "X-axis Label";
        char * y_axis_label = "Y-axis Label";

        // File name for saving
        char * file_name = "pic_name";
        char * file_extension = "png";
        // -------------------------------------------------------------------------

        // Chart Processing Methods
        // -------------------------------------------------------------------------
        // Function tweaks Series
        void tweak_data();

        // Function tweaks Chart
        void tweak_chart();

        // Function tweaks ChartView
        void tweak_chartview();

        // Function tweaks Pen
        void tweak_pen();

        // Function tweaks Axes
        void tweak_axes();
```

```
        // Function tweaks Fonts
        void tweak_fonts();

        // Function saves Chart as a file
        void save_chart();
        // -------------------------------------------------------------------------
}; // class User_chart_wrapper

#endif USER_CHART_WRAPPER
```

# ДОДАТОК В

Лістинг програми, що дозволяє будувати двовимірні графіки та зберігати їх у вигляді файлів зображень *.png*, мовою С++. Текст файлу *user_chart_wrapper.cpp*, що містить реалізацію класу *User_chart_wrapper*, за допомогою якого будуються графіки

```
#include "user_chart_wrapper.h"


User_chart_wrapper::User_chart_wrapper
(
	char * file_name,
	char * chart_name,
	char * x_axis_label,
	char * y_axis_label,
	void(*get_values_func)
	(
		std::vector<double> * x_axis_data,
		std::vector<double> * y_axis_data,
		double left_x_limit,
		double right_x_limit,
		double x_step
	),
	double left_x_limit,
	double right_x_limit,
	double x_step,
	int ticks_per_x_axis,
	int ticks_per_y_axis,
	char * file_extension
) : left_x_limit{ left_x_limit },
right_x_limit{ right_x_limit },
x_step{ x_step },
get_values_func{ get_values_func },
file_name{ file_name },
chart_name{ chart_name },
x_axis_label{ x_axis_label },
y_axis_label{ y_axis_label },
ticks_per_x_axis{ ticks_per_x_axis },
ticks_per_y_axis{ ticks_per_y_axis },
file_extension{ file_extension }
{
	tweak_pen();
	tweak_data();
	tweak_fonts();
	tweak_axes();
	tweak_chart();
	tweak_chartview();

	save_chart();
}


User_chart_wrapper::~User_chart_wrapper()
{
	// Freeing all allocated memory
	// -------------------------------------------------------------------------
	delete this->axis_y;
```

```
        delete this->axis_x;
        delete this->pen;
        delete this->series;
        delete this->chart;
        delete this->chart_view;
        // -----------------------------------------------------------------------
}


void User_chart_wrapper::tweak_chart()
{
        // Memory allocating for Chart object
        this->chart = new QtCharts::QChart();
        this->chart->legend()->hide();
        // Title
        this->chart->setTitle(QString::fromLocal8Bit(this->chart_name));
        this->chart->setTitleFont(*(this->title_font));
        // Axes
        this->chart->addAxis(this->axis_x, Qt::AlignBottom);
        this->chart->addAxis(this->axis_y, Qt::AlignLeft);
        // Series
        this->chart->addSeries(this->series);

        return;
}


void User_chart_wrapper::tweak_chartview()
{
        // Memory allocating for ChartView object
        this->chart_view = new QtCharts::QChartView(this->chart);

        this->chart_view->setRenderHint(QPainter::Antialiasing);

        return;
}


void User_chart_wrapper::tweak_data()
{
        // Memory allocating for Series object
        this->series = new QtCharts::QLineSeries();

        // Allocating Local variables
        // -----------------------------------------------------------------------
        std::vector<double> * x_axis_data = new std::vector<double>{};
        std::vector<double> * y_axis_data = new std::vector<double>{};
        // -----------------------------------------------------------------------

        // If assert is happened -> get_values_func is not defined
        assert(this->get_values_func != nullptr);

        // Getting values from external function
        // -----------------------------------------------------------------------
        this->get_values_func
        (
                x_axis_data, y_axis_data,
                this->left_x_limit, this->right_x_limit, this->x_step
        );
        // -----------------------------------------------------------------------

        assert(x_axis_data->size() == y_axis_data->size());

        // Appending data into Series
```

```
        // ------------------------------------------------------------------------
        for (std::vector<double>::iterator x_iter = x_axis_data->begin(),
                x_end_iter = x_axis_data->end(), y_iter =  y_axis_data->begin();
                x_iter < x_end_iter; ++x_iter, ++y_iter)
        {
                this->series->append((*x_iter), (*y_iter));
        }
        // ------------------------------------------------------------------------

        // Searching minimum and maximum value for Y axis values
        // ------------------------------------------------------------------------
        std::vector<double>::iterator y_iter = y_axis_data->begin();
        this->left_y_limit = (*y_iter);
        this->right_y_limit = this->left_y_limit;
        for (std::vector<double>::iterator y_iter_end = y_axis_data->end();
                y_iter < y_iter_end; ++y_iter)
        {
                this->left_y_limit = (this->left_y_limit > (*y_iter)) ?
                        (*y_iter) : (this->left_y_limit);
                this->right_y_limit = (this->right_y_limit < (*y_iter)) ?
                        (*y_iter) : (this->right_y_limit);
        }
        // ------------------------------------------------------------------------

        // Appending pen object
        this->series->setPen(*(this->pen));

        // Deallocating memory
        // ------------------------------------------------------------------------
        delete x_axis_data;
        delete y_axis_data;
        // ------------------------------------------------------------------------

        return;
}


void User_chart_wrapper::tweak_pen()
{
        // Memory allocating for Pen object
        this->pen = new QPen(QRgb(0x000000));

        this->pen->setWidth(3);

        return;
}


void User_chart_wrapper::tweak_axes()
{
        // Tweaking Axis X
        // ------------------------------------------------------------------------
        this->axis_x = new QtCharts::QValueAxis();

        this->axis_x->setRange(this->left_x_limit, this->right_x_limit);
        this->axis_x->setTickCount(this->ticks_per_x_axis);

        this->axis_x->setTitleText(QString::fromLocal8Bit(this->x_axis_label));
        this->axis_x->setTitleFont(*(this->axes_font));
        // ------------------------------------------------------------------------

        // Tweaking Axis Y
        // ------------------------------------------------------------------------
        this->axis_y = new QtCharts::QValueAxis();
```

```
        this->axis_y->setRange(this->left_y_limit, this->right_y_limit);
        this->axis_y->setTickCount(this->ticks_per_y_axis);

        this->axis_y->setTitleText(QString::fromLocal8Bit(this->y_axis_label));
        this->axis_y->setTitleFont(*(this->axes_font));
        // ------------------------------------------------------------------

        return;
}


void User_chart_wrapper::tweak_fonts()
{
        // Chart title font tweaking
        // ------------------------------------------------------------------
        this->title_font = new QFont("Times New Roman", 18, QFont::Bold);
        // ------------------------------------------------------------------

        // Chart axes font tweaking
        // ------------------------------------------------------------------
        this->axes_font = new QFont("Times New Roman", 14, QFont::Light);
        // ------------------------------------------------------------------

        return;
}


void User_chart_wrapper::save_chart()
{
        this->chart_view->grab().save
        (
                QString(QString(this->file_name) + '.' +
                        QString(this->file_extension)),
                this->file_extension
        );

        return;
}
```

## ДОДАТОК Г

Лістинг програми, що розраховує частинні похідні для виразу сили кутового потенціалу мовою програмування Python

```
# Importing library for symbolic calculations
# ----------------------------------------------------------------------------
import sympy
# ----------------------------------------------------------------------------

# Variables defining
# ----------------------------------------------------------------------------
x1, x2, x3 = sympy.symbols("x_1 x_2 x_3")
y1, y2, y3 = sympy.symbols("y_1 y_2 y_3")
z1, z2, z3 = sympy.symbols("z_1 z_2 z_3")
x = sympy.symbols('x')
sympy.init_printing(use_unicode=True)
# ----------------------------------------------------------------------------

numerator = \
    (x2 - x1)*(x3 - x1) + \
    (y2 - y1)*(y3 - y1) + \
    (z2 - z1)*(z3 - z1)

numerator = numerator.expand()
numerator = numerator.together()

# Showing result
numerator

denominator = \
(
    (((x2 - x1)**2) + ((y2 - y1)**2) + ((z2 - z1)**2)) *
    (((x3 - x1)**2) + ((y3 - y1)**2) + ((z3 - z1)**2))
)**(1/2)

# Showing result
denominator

acos_diff = sympy.diff(sympy.acos(x), x)
# Showing result
acos_diff

acos_diff = acos_diff.subs(x, numerator/denominator)
acos_diff

# Calculating derivatives for body #2
# ----------------------------------------------------------------------------
# dU/d(x2)
d_d_x2 = sympy.diff(numerator/denominator, x2)
d_d_x2.simplify()
```

```
# dU/d(y2)
d_d_y2 = sympy.diff(numerator/denominator, y2)
d_d_y2.simplify()

# dU/d(z2)
d_d_z2 = sympy.diff(numerator/denominator, z2)
d_d_z2.simplify()
# ------------------------------------------------------------------------------

# Calculating derivatives for body #3
# ------------------------------------------------------------------------------
# dU/d(x3)
d_d_x3 = sympy.diff(numerator/denominator, x3)
d_d_x3.simplify()

# dU/d(y3)
d_d_y3 = sympy.diff(numerator/denominator, y3)
d_d_y3.simplify()

# dU/d(z3)
d_d_z3 = sympy.diff(numerator/denominator, z3)
d_d_z3.simplify()
# ------------------------------------------------------------------------------
```

# ДОДАТОК Д

Лістинг програми для проведення експерименту розрахунку радіусу відсікання мовою програмування С++. Файл *main.cu*

```
#include "Dynamic_chart.h"
#include <QtCharts/qchartview.h>
#include <QtWidgets/qapplication.h>
#include <QtWidgets/qmainwindow.h>
#include <iostream>
#include <vector>
#include <chrono>
#include <fstream>


int main(int argc, char ** argv)
{
	std::fstream result_file;
	MD_experiment * CPU_experiment = new MD_experiment{};
	long double system_energy_accumulator = 0.0, system_energy = 0.0;
	long double total_result = 0.0, time_step = 0.0005;
	std::vector<long double> radiuses{};
	int t = 1, t_max = 99, t_total = t_max + 1, iter_count = 10;

	result_file.open("Compare_results.txt", std::ios::out | std::ios::in);

	// Setting radiuses
	for (int i = 1; i <= 100; i++)
	{
		radiuses.push_back(
			i*0.5*(*Water_molecule::equilibrium_distance_LJ_OO));
	}

	// Experiment without R-cut : CPU
	total_result = 0.0;
	for (int i = 1; i <= iter_count; ++i)
	{
		initialise_md_experiment_1(CPU_experiment, system_energy);
		// std::cout << "\n\nSystem # " << i << " is initialized!";
		(*CPU_experiment->time_step) = time_step;
		system_energy_accumulator = 0.0;
		system_energy_accumulator += system_energy / t_total;
		for (t = 1; t < t_max; ++t)
		{
			CPU_experiment->run_experiment_brute_force(system_energy);
			system_energy_accumulator += system_energy / t_total;
		}
		total_result += system_energy_accumulator / iter_count;
		system_energy_accumulator = 0.0;
	}
	result_file << "\nExperiment Without R-cut. Result: " << total_result;
	total_result = 0.0;


	// Experiment with R-cut : CPU
	for (long double radius : radiuses)
	{
		total_result = 0.0;
		for (int i = 1; i <= iter_count; ++i)
		{
```

```
                system_energy = 0.0;
                initialise_md_experiment_1(CPU_experiment, system_energy);
                // std::cout << "\n\nSystem # " << i << " is initialized!";
                (*CPU_experiment->time_step) = time_step;
                system_energy_accumulator = 0.0;
                system_energy_accumulator += system_energy / t_total;
                for (t = 1; t < t_max; ++t)
                {
                        CPU_experiment->run_experiment_brute_force_with_r
                        (
                                system_energy, radius
                        );
                        system_energy_accumulator += system_energy / t_total;
                }
                total_result += system_energy_accumulator / iter_count;
                system_energy_accumulator = 0.0;
        }
        result_file << "\nExperiment With R-cut.";
        result_file << "\n\tRadius: " << radius;
        result_file << "\n\tResult: " << total_result;
    }

    std::cout << "\nCompleted!";
    return 0;
}
```

# ДОДАТОК Е

Лістинг програми для проведення моделювання динаміки системи взаємодіючих частинок з побудовою графіку залежності енергії від часу моделювання мовою програмування C++. Файл *main.cu*

```
#include "Dynamic_chart.h"
#include <QtCharts/qchartview.h>
#include <QtWidgets/qapplication.h>
#include <QtWidgets/qmainwindow.h>
#include <iostream>
#include <chrono>


int main(int argc, char ** argv)
{
	QApplication * app = new QApplication{ argc, argv };
	QMainWindow * window = new QMainWindow{};
	Dynamic_chart * chart = new Dynamic_chart
	(
		0.0005, 10.0, 0.0, 5.0, 200.0, 0,
		QString::fromLocal8Bit
		(
			"Залежність середньої енергії від часу моделювання"
		),
		QString::fromLocal8Bit("Енергія"),
		QString::fromLocal8Bit
		(
			"Час, безрозмірні одиниці"
		),
		QString::fromLocal8Bit
		(
			"Енергія, безрозмірні одиниці"
		)
	);
	QtCharts::QChartView * chartView =
		new QtCharts::QChartView{ chart };

	chart->legend()->hide();

	chartView->setRenderHint(QPainter::Antialiasing);

	window->setCentralWidget(chartView);
	window->resize(1080, 720);
	window->show();

	chart->run_MD_experiment_CPU();

	return app->exec();
```

# ДОДАТОК Ж

Лістинг оголошення класу «Atom» для побудови об’єктів моделей атомів мовою програмування С++. Файл *Atom.h*

```
#ifndef MOLDY_ATOM
#define MOLDY_ATOM


class Atom
{
private:
	char * a_type = nullptr;
	int * size = nullptr;
	long double * data = nullptr;

	// Static variables: Offsets
	// --------------------------------------------------------------------------
	// Coordinates   [00] [01] [02]
	// Velocities    [03] [04] [05]
	// Accelerations [06] [07] [08]
	// Forces        [09] [10] [11]
	// static int * coordinate_offset;
	static int const * velocity_index_offset;
	static int const * acceleration_index_offset;
	static int const * force_index_offset;
	static int const * mass_index;
	// --------------------------------------------------------------------------

public:
	Atom();
	~Atom();

	// Set... methods
	// --------------------------------------------------------------------------
	inline void set_coordinate(int const index, long double value)
	{
		this->data[index] = value;
	}

	inline void set_velocity(int const index, long double value)
	{
		this->data[index + (*(this->velocity_index_offset))] = value;
	}

	inline void set_acceleration(int const index, long double value)
	{
		this->data[index + (*(this->acceleration_index_offset))] = value;
	}

	inline void set_force(int const index, long double value)
	{
		this->data[index + (*(this->force_index_offset))] = value;
	}

	inline void set_type(char const atom_type)
	{
		(*this->a_type) = atom_type;
	}
	// --------------------------------------------------------------------------
```

```
        // Get... methods
        // -----------------------------------------------------------------------
        inline long double get_coordinate(int const index)
        {
                return this->data[index];
        }

        inline long double get_velocity(int const index)
        {
                return this->data[index + (*(this->velocity_index_offset))];
        }

        inline long double get_acceleration(int const index)
        {
                return this->data[index + (*(this->acceleration_index_offset))];
        }

        inline long double get_force(int const index)
        {
                return this->data[index + (*(this->force_index_offset))];
        }

        inline long double get_mass()
        {
                return this->data[*(this->mass_index)];
        }

        inline char get_type()
        {
                return (*this->a_type);
        }
        // -----------------------------------------------------------------------

        // Update... methods
        // upd == update
        // -----------------------------------------------------------------------
        inline void upd_coordinate(int const index, long double value)
        {
                this->data[index] += value;
        }

        inline void upd_velocity(int const index, long double value)
        {
                this->data[index + (*(this->velocity_index_offset))] += value;
        }

        inline void upd_acceleration(int const index, long double value)
        {
                this->data[index + (*(this->acceleration_index_offset))] += value;
        }

        inline void upd_force(int const index, long double value)
        {
                this->data[index + (*(this->force_index_offset))] += value;
        }
        // -----------------------------------------------------------------------
}; // class Atom END

#endif // MOLDY_ATOM
```

# ДОДАТОК И

Лістинг оголошення класу «Dynamic_chart» для побудови об'єктів, що дозволяють будувати графіки мовою програмування C++. Файл *Dynamic_chart.h*

```
#ifndef DYNAMIC_CHART
#define DYNAMIC_CHART


#include "GPU_functions.cuh"

#include <cstdlib>

#include <qdebug.h>
#include <QtCore/qtimer.h>
#include <QtCore/qrandom.h>
#include <QtCore/qdebug.h>
#include <QtCharts/qchart.h>
#include <QtCharts/qabstractaxis.h>
#include <QtCharts/qsplineseries.h>
#include <QtCharts/qvalueaxis.h>

#include "MD_experiment.cuh"
#include "MD_experiment_init_1.cuh"


class Dynamic_chart: public QtCharts::QChart
{
    Q_OBJECT
public:
    Dynamic_chart
       (
              long double time_step,
              long double axis_OX_tick_count = 10,
              long double scale_OX_left_lim = 0.0,
              long double scale_OX_right_lim = 10.0,
              long double scroll_limit = 12.0,
              int time_interval = 0,
              QString chart_title = QString::fromUtf8("Chart Title"),
              QString chart_legend = QString::fromUtf8("Legend Title"),
              QString axis_OX_label = QString::fromUtf8("Axis OX Label"),
              QString axis_OY_label = QString::fromUtf8("Axis OY Label"),
              QGraphicsItem *parent = nullptr,
              Qt::WindowFlags wFlags = {}
       );
    virtual ~Dynamic_chart();

       // For CPU computations
       void init_MD_experiment_CPU();
       void run_MD_experiment_CPU();

       // For GPU computations (CUDA)
       void init_MD_experiment_GPU();
       void run_MD_experiment_GPU();

private slots:
       // Function executes chart drawing and molecular dynamic experiment
    void user_timeout_CPU();
```

```
        void user_timeout_GPU();

private:
        QString * title = Q_NULLPTR;
        QString * legend_label = Q_NULLPTR;
        QString * axis_OX_label = Q_NULLPTR;
        QString * axis_OY_label = Q_NULLPTR;
    QTimer * timer_CPU = Q_NULLPTR;
        QTimer * timer_GPU = Q_NULLPTR;
        QtCharts::QSplineSeries * data_series = Q_NULLPTR;
    QtCharts::QValueAxis * axis_OX = Q_NULLPTR;
        QtCharts::QValueAxis * axis_OY = Q_NULLPTR;
        QPen * chart_pen = Q_NULLPTR;
        QFont * title_font = Q_NULLPTR;
        QFont * legend_font = Q_NULLPTR;
        QFont * axes_label_font = Q_NULLPTR;
        QFont * axes_ticks_font = Q_NULLPTR;
        long double (*md_do_iteration)() = nullptr;
    long double * x_scrolling_step = nullptr;
        long double * point_x_component = nullptr;
        long double * point_y_component = nullptr;
        long double * buffer = nullptr;
        long double * point_x_component_step = nullptr;
        long double * axis_OX_tick_count = nullptr;
        long double * scale_OX_left_lim = nullptr;
        long double * scale_OX_right_lim = nullptr;
        long double * scale_OY_min = nullptr;
        long double * scale_OY_max = nullptr;
        long double * scroll_limit = nullptr;
        int * time_interval = nullptr;

        // Experiment
        MD_experiment * moldy_experiment;
        long double * system_energy;
};

#endif // DYNAMIC_CHART
```

# ДОДАТОК К

Лістинг оголошення класу «MD_experiment» для побудови об’єктів, що містять функції та дані для проведення комп’ютерного експерименту мовою програмування С++. Файл *MD_experiment.cuh*

```
#ifndef MOLDY_EXPERIMENT
#define MOLDY_EXPERIMENT


#include <math_constants.h>
#include <cassert>
#include <iostream>
#include "Water_molecule.h"


class MD_experiment
{
public:
	// Time step value
	long double * time_step = nullptr;

	// Number of particles in system
	size_t const * particle_count = nullptr;

	// Maximum particle count
	static size_t * max_particle_count;
	void calculate_max_particle_count();

	// Function show the current parameters of MD experiment
	void show_experiment_parameters();

	// Constructor
	MD_experiment();

	// Destructor
	~MD_experiment();

	// Step #0: Preparing constants
	void setup_constants();

	// Step #1: Setting the modelling area size
	void set_modelling_area_sizes
	(
		long double width, long double length, long double height
	);

	// Step #2: Setting the particle count in the system
	void set_particle_count(size_t particle_count);

	// Step #3: Defining particle coordinates
	void set_particle_coordinates();

	// Step #4: Defining particle velocities
	void set_particle_velocities();

	// Step #5: Defining particle accelerations. Brute Force
	// with system energy calculation
	long double upd_particle_accelerations_brute_force();
```

```
        long double upd_particle_accelerations_brute_force_with_r(long double r_cut);

        // Step #6: Running experiment. Brute force. With System Energy
        // File: 'MD_experiment_Brute_Force_iteration.cu'
        // Function does 1 Simulation iteration
        void run_experiment_brute_force(long double & system_energy);
        void run_experiment_brute_force_with_r
        (
                long double & system_energy, long double r_cut
        );

private:
        // Force equations -----------------------------------------------------
        // Function calculates Lennard-Jones Force value
        long double lennard_jones_force
        (
                long double & distance,
                long double & energy,
                long double & equilibrium_distance
        );

        // Function calculates Coulomb Force value
        long double coulomb_force
        (
                long double & distance,
                long double & coulomb_const,
                long double & charge_1,
                long double & charge_2
        );

        // Function calculates Bond Force value
        long double bond_force
        (
                long double & distance,
                long double & equilibrium_distance,
                long double & bond_coefficient
        );

        // Function calculates Angle Force value -- OLD Version
        long double angle_force_left_atom
        (
                Water_molecule & molecule,
                long double & equilibrium_angle,
                long double & angle_coefficient,
                long double atom_0_coordinate_component,
                long double atom_1_coordinate_component,
                long double atom_2_coordinate_component,
                long double angle
        );

        // Function calculates Angle Force value -- OLD Version
        long double angle_force_right_atom
        (
                Water_molecule & molecule,
                long double & equilibrium_angle,
                long double & angle_coefficient,
                long double atom_0_coordinate_component,
                long double atom_1_coordinate_component,
                long double atom_2_coordinate_component,
                long double & angle
        );

        // Function calculates Angle Force value
        long double angle_force
```

```
(
        long double angle,
        long double & equilibrium_angle,
        long double & angle_coefficient
);
// Force equations: END --------------------------------------------------


// Potential equations
// -------------------------------------------------------------------------
// Function calculates Lennard-Jones Potential value
long double lennard_jones_potential
(
        long double & distance,
        long double & energy,
        long double & equilibrium_distance
);

// Function calculates Coulomb Potential value
long double coulomb_potential
(
        long double & distance,
        long double & coulomb_const,
        long double & charge_1,
        long double & charge_2
);

// Function calculates Bond Potential value
long double bond_potential
(
        long double & distance,
        long double & equilibrium_distance,
        long double & bond_coefficient
);

// Function calculates Angle Potential value
long double angle_potential
(
        long double & angle_value,
        long double & equilibrium_angle,
        long double & angle_coefficient
);
// -------------------------------------------------------------------------


// Operations with Atom models
// -------------------------------------------------------------------------
// Function calculates vector dot product using Atom's radius vectors
long double get_vector_dot_product
(
        long double vector_1_x, long double vector_1_y, long double vector_1_z,
        long double vector_2_x, long double vector_2_y, long double vector_2_z
);

// Function calculates squared distance between Atom couple
long double get_squared_distance
(
        long double atom_1_x, long double atom_1_y, long double atom_1_z,
        long double atom_2_x, long double atom_2_y, long double atom_2_z
);

// Function calculates distance between Atom couple
long double get_distance
(
```

```
		long double atom_1_x, long double atom_1_y, long double atom_1_z,
		long double atom_2_x, long double atom_2_y, long double atom_2_z
	);

	// Function corrects coordinates
	void coordinate_correction
	(
		Water_molecule * mol_ptr, int atom_i, int dim_i
	);

	// Function corrects distance component
	long double distance_correction(long double value, int dim_i);
	long double distance_correction0(long double value, int dim_i);

	// Function calculates the angle between bonds
	long double get_angle(Water_molecule & mol);
	// -----------------------------------------------------------------------
// Energy calculations
	// -----------------------------------------------------------------------
	// Function calculates kinetic energy value
	long double get_kinetic_energy
	(
		long double squared_velocity,
		long double & mass
	);
	// -----------------------------------------------------------------------

// private:
public:
	size_t * array_size = nullptr;
	Water_molecule * particle_array = nullptr;

	// Dimension count
	static int * dimension_count;

	// Distance epsilon
	static long double * distance_eps;

	// Modelling area shape parameters
	long double * modelling_area_width = nullptr;
	long double * modelling_area_length = nullptr;
	long double * modelling_area_height = nullptr;

	// Start atom coordinates for molecule model
	static long double * start_coordinate_atom_1;
	static long double * start_coordinate_atom_2;
	static long double * start_coordinate_atom_3;
	static void calculate_basic_atom_coordinates();



public:
	// Modelling subarea shape parameters
	static long double * modelling_subarea_width;  // OX
	static long double * modelling_subarea_length; // OY
	static long double * modelling_subarea_height; // OZ
	static void calculate_modelling_subarea_params();

// Class members for GPU using
// ---------------------------------------------------------------------------
public:
	// Variables for Water_molecule constants
	// -----------------------------------------------------------------------
	// Atomic types: 'O' - Oxygen; 'H' - Hydrogen
```

```
        char * atom_type_O_GPU = nullptr;
        char * atom_type_H_GPU = nullptr;

        // Atomic masses: O - Oxygen; H - Hydrogen
        long double * atom_mass_O_GPU = nullptr;
        long double * atom_mass_H_GPU = nullptr;

        // Bond Potential Params
        long double * equilibrium_distance_bond_poten_GPU = nullptr;
        long double * k_coef_bond_poten_GPU = nullptr;

        // Angle potential
        long double * equilibrium_angle_GPU = nullptr;
        long double * k_coef_angle_poten_GPU = nullptr;

        // Coulomb potential
        long double * q_charge_O_GPU = nullptr;
        long double * q_charge_H_GPU = nullptr;
        long double * k_coef_coulomb_potential_GPU = nullptr;

        // Lennard-Jones potential
        long double * energy_LJ_HH_GPU = nullptr;
        long double * equilibrium_distance_LJ_HH_GPU = nullptr;
        long double * energy_LJ_OO_GPU = nullptr;
        long double * equilibrium_distance_LJ_OO_GPU = nullptr;
        long double * energy_LJ_OH_GPU = nullptr;
        long double * equilibrium_distance_LJ_OH_GPU = nullptr;

        // Number of atoms in molecule
        int * atom_count_GPU = nullptr;
        // -----------------------------------------------------------------------


// General variables
        // -----------------------------------------------------------------------
        // The total energy of the system of Water molecule models
        long double * system_energy_GPU = nullptr;

        // The kinetic energy of the system of water molecule models
        long double * kinetic_energy_GPU = nullptr;

        // The potential energy of the system of water molecule models
        long double * potential_energy_GPU = nullptr;

        // The number of molecules in system
        int * molecule_count_GPU = nullptr;

        // The number of atoms in the system
        int * total_atom_count_GPU = nullptr;

        // The parameters for submodelling area
        long double * subarea_width_GPU = nullptr;
        long double * subarea_length_GPU = nullptr;
        long double * subarea_height_GPU = nullptr;

        // The parameters for modelling area
        long double * area_width_GPU = nullptr;
        long double * area_length_GPU = nullptr;
        long double * area_height_GPU = nullptr;

        // Buffer
        long double * buffer_GPU = nullptr;

        // Dimension count
```

```
        int * dims_GPU = nullptr;

        // GPU computation configuration parameters
        int * block_size_GPU = nullptr;
        int * block_count_GPU = nullptr;

        // Time step
        long double * time_step_GPU = nullptr;
        // -----------------------------------------------------------------------


// Atoms arrays
        // -----------------------------------------------------------------------
        // Atoms coordinates
        long double * atoms_OX_coordinates_GPU = nullptr;
        long double * atoms_OY_coordinates_GPU = nullptr;
        long double * atoms_OZ_coordinates_GPU = nullptr;

        // Atoms velocities
        long double * atoms_OX_velocities_GPU = nullptr;
        long double * atoms_OY_velocities_GPU = nullptr;
        long double * atoms_OZ_velocities_GPU = nullptr;

        // Atoms accelerations
        long double * atoms_OX_accelerations_GPU = nullptr;
        long double * atoms_OY_accelerations_GPU = nullptr;
        long double * atoms_OZ_accelerations_GPU = nullptr;

        // Atoms forces
        long double * atoms_OX_forces_GPU = nullptr;
        long double * atoms_OY_forces_GPU = nullptr;
        long double * atoms_OZ_forces_GPU = nullptr;

        // Atoms masses
        long double * atoms_masses_GPU = nullptr;

        // Atoms types
        char * atoms_types_GPU = nullptr;
        // -----------------------------------------------------------------------
// -----------------------------------------------------------------------------
}; // MD_experiment

#endif // MOLDY_EXPERIMENT#
```

# ДОДАТОК Л

Лістинг інтерфейсу функції ініціалізації експерименту мовою програмування C++. Файл *MD_experiment_init_1.cuh*

```
#ifndef MD_EXPERIMENT_INIT_1
#define MD_EXPERIMENT_INIT_1

#include <iostream>
#include "MD_experiment.cuh"


void initialise_md_experiment_1
(
	MD_experiment * moldy_experiment,
	long double & system_energy
);

#endif // MD_EXPERIMENT_INIT_1
```

# ДОДАТОК М

Лістинг оголошення класу «Water_molecule» для побудови об’єктів моделей молекули води мовою програмування С++. Файл *Water_molecule.h*

```
#ifndef MOLDY_WATER_MOLECULE
#define MOLDY_WATER_MOLECULE


#include "Atom.h"


class Water_molecule
{
private:
	// Molecule structure
	// -------------------------------------------------------------------------
	Atom * atom_array = nullptr;
	char * molecule_type = nullptr;
	// -------------------------------------------------------------------------

public:
	// Static variables
	// -------------------------------------------------------------------------
	// Atomic types: 'O' - Oxygen; 'H' - Hydrogen
	static char * type_O;
	static char * type_H;

	// Atomic masses: O - Oxygen; H - Hydrogen
	static long double * mass_H;
	static long double * mass_O;

	// Bond Potential Params
	static long double * equilibrium_distance_bond_poten;
	static long double * k_coef_bond_poten;

	// Angle potential
	static long double * equilibrium_angle;
	static long double * k_coef_angle_poten;

	// Coulomb potential
	static long double * q_charge_H;
	static long double * q_charge_O;
	static long double * k_coef_coulomb_potential;

	// Lennard-Jones potential
	static long double * energy_LJ_HH;
	static long double * equilibrium_distance_LJ_HH;
	static long double * energy_LJ_OO;
	static long double * equilibrium_distance_LJ_OO;
	static long double * energy_LJ_OH;
	static long double * equilibrium_distance_LJ_OH;

	// Number of atoms in molecue
	static int const * atom_count;
	// -------------------------------------------------------------------------

	Water_molecule();
	Water_molecule(char * mol_type);
	~Water_molecule();
```

```
// Set... methods
// -----------------------------------------------------------------
inline void set_coordinate(int atom_i, int coordinate_i, long double value)
{
	this->atom_array[atom_i].set_coordinate(coordinate_i, value);
}

inline void set_velocity(int atom_i, int velocity_i, long double value)
{
	this->atom_array[atom_i].set_velocity(velocity_i, value);
}

inline void set_acceleration
(
	int atom_i, int acceleration_i, long double value
)
{
	this->atom_array[atom_i].set_acceleration(acceleration_i, value);
}

inline void set_force(int atom_i, int force_i, long double value)
{
	this->atom_array[atom_i].set_force(force_i, value);
}

inline void set_type(int atom_i, char type)
{
	this->atom_array[atom_i].set_type(type);
}
// -----------------------------------------------------------------

// Get... methods
// -----------------------------------------------------------------
inline long double get_coordinate(int atom_i, int coordinate_i)
{
	return this->atom_array[atom_i].get_coordinate(coordinate_i);
}

inline long double get_velocity(int atom_i, int velocity_i)
{
	return this->atom_array[atom_i].get_velocity(velocity_i);
}

inline long double get_acceleration(int atom_i, int acceleration_i)
{
	return this->atom_array[atom_i].get_acceleration(acceleration_i);
}

inline long double get_force(int atom_i, int force_i)
{
	return this->atom_array[atom_i].get_force(force_i);
}

inline char get_type(int atom_i)
{
	return this->atom_array[atom_i].get_type();
}
// -----------------------------------------------------------------

// Update... methods
// -----------------------------------------------------------------
// upd -- Update
// Updating molecue fields
```

```
	inline void upd_coordinate
	(
		int atom_i, int coordinate_i, long double value
	)
	{
		this->atom_array[atom_i].upd_coordinate(coordinate_i, value);
	}

	inline void upd_velocity(int atom_i, int velocity_i, long double value)
	{
		this->atom_array[atom_i].upd_velocity(velocity_i, value);
	}

	inline void upd_acceleration
	(
		int atom_i, int acceleration_i, long double value
	)
	{
		this->atom_array[atom_i].upd_acceleration(acceleration_i, value);
	}

	inline void upd_force(int atom_i, int force_i, long double value)
	{
		this->atom_array[atom_i].upd_force(force_i, value);
	}
	// ------------------------------------------------------------------------

}; // class Water_molecue

#endif // MOLDY_WATER_MOLECULE
```

# ДОДАТОК Н

Лістинг реалізації класу «Atom» мовою програмування C++. Файл *Atom.cpp*

```
#include "Atom.h"

// Static variables initialisation
// ----------------------------------------------------------------------------
// /*
int const * Atom::velocity_index_offset = new int(3);
int const * Atom::acceleration_index_offset = new int(6);
int const * Atom::force_index_offset = new int(9);
int const * Atom::mass_index = new int(12);
// */
// ----------------------------------------------------------------------------

Atom::Atom():
	a_type{ new char{'\0'} },
	size{ new int {13} },
	data{ new long double[*size] }
{
	// data[0]  -> Coordinate OX
	// data[1]  -> Coordinate OY
	// data[2]  -> Coordinate OZ
	// data[3]  -> Velocity OX
	// data[4]  -> Velocity OY
	// data[5]  -> Velocity OZ
	// data[6]  -> Acceleration OX
	// data[7]  -> Acceleration OY
	// data[8]  -> Acceleration OZ
	// data[9]  -> Force OX
	// data[10] -> Force OY
	// data[11] -> Force OZ
	// data[12] -> Mass

	// Array initialization
	// ----------------------------------------------------------------------------
	for (long double * beg = this->data, *end = this->data + (*size);
		end > beg; ++beg)
	{
		(*beg) = 0.0;
	}
	// ----------------------------------------------------------------------------
}

Atom::~Atom()
{
	delete[] this->data;
	delete this->size;
	delete this->a_type;
}
```

# ДОДАТОК П

Лістинг реалізації класу «Dynamic_chart» мовою програмування C++. Файл *Dynamic_chart.cpp*

```
#include "Dynamic_chart.h"


Dynamic_chart::Dynamic_chart
(
	long double time_step,
	long double axis_OX_tick_count,
	long double scale_OX_left_lim,
	long double scale_OX_right_lim,
	// long double scale_OY_left_lim,
	// long double scale_OY_right_lim,
	long double scroll_limit,
	int time_interval,
	QString chart_title,
	QString chart_legend,
	QString axis_OX_label,
	QString axis_OY_label,
	QGraphicsItem * parent,
	Qt::WindowFlags wFlags
):
	QChart(QChart::ChartTypeCartesian, parent, wFlags),
	timer_CPU{ new QTimer() },
	timer_GPU{ new QTimer() },
	data_series{ new QtCharts::QSplineSeries() },
	axis_OX{ new QtCharts::QValueAxis() },
	axis_OY{ new QtCharts::QValueAxis() },
	title{ new QString(chart_title) },
	legend_label{ new QString(chart_legend) },
	axis_OX_label{ new QString(axis_OX_label) },
	axis_OY_label{ new QString(axis_OY_label) },
	chart_pen{ new QPen() },
	title_font{ new QFont() },
	legend_font{ new QFont() },
	axes_label_font{ new QFont() },
	axes_ticks_font{ new QFont() },
	md_do_iteration{ md_do_iteration },
	x_scrolling_step{ new long double{0.0} },
	point_x_component{ new long double{0.0} },
	point_y_component{ new long double{0.0} },
	buffer{ new long double{0.0} },
	point_x_component_step{ new long double{time_step} },
	axis_OX_tick_count{ new long double{axis_OX_tick_count} },
	scale_OX_left_lim{ new long double{scale_OX_left_lim} },
	scale_OX_right_lim{ new long double{scale_OX_right_lim} },
	scale_OY_min{ new long double{ 0.0 } },
	scale_OY_max{ new long double{ 0.0 } },
	scroll_limit{ new long double{scroll_limit} },
	time_interval{ new int{time_interval} },
	moldy_experiment{ new MD_experiment{} },
	system_energy{ new long double{ 0.0 } }
{
	// Tweaking the timers
	// ---------------------------------------------------------------------------
	QObject::connect
	(
```

```
        this->timer_CPU, &QTimer::timeout,
        this, &Dynamic_chart::user_timeout_CPU
);
this->timer_CPU->setInterval(*this->time_interval);

QObject::connect
(
        this->timer_GPU, &QTimer::timeout,
        this, &Dynamic_chart::user_timeout_GPU
);
this->timer_GPU->setInterval(*this->time_interval);
// -----------------------------------------------------------------------

// Tweaking the chart
// -----------------------------------------------------------------------
this->addSeries(this->data_series);
this->addAxis(this->axis_OX, Qt::AlignBottom);
this->addAxis(this->axis_OY, Qt::AlignLeft);
// -----------------------------------------------------------------------

// Tweaking the pen for the chart line
// -----------------------------------------------------------------------
this->chart_pen->setColor(Qt::black);
this->chart_pen->setWidth(1);
// -----------------------------------------------------------------------

// Tweaking the title font
// -----------------------------------------------------------------------
this->title_font->setFamily("Times New Roman");
this->title_font->setPointSize(18);
this->title_font->setWeight(QFont::Bold);
this->setTitleFont(*this->title_font);
// -----------------------------------------------------------------------

// Tweaking the legend font
// -----------------------------------------------------------------------
this->legend_font->setFamily("Times New Roman");
this->legend_font->setPointSize(10);
this->legend_font->setWeight(QFont::Light);
this->legend()->setFont(*this->legend_font);
// -----------------------------------------------------------------------

// Tweaking the axes scale labels font
// -----------------------------------------------------------------------
this->axes_ticks_font->setFamily("Times New Roman");
this->axes_ticks_font->setPointSize(10);
this->axes_ticks_font->setWeight(QFont::Light);

this->axis_OX->setLabelsFont(*this->axes_ticks_font);
this->axis_OY->setLabelsFont(*this->axes_ticks_font);
// -----------------------------------------------------------------------

// Tweaking the axes labels font
// -----------------------------------------------------------------------
this->axes_label_font->setFamily("Times New Roman");
this->axes_label_font->setPointSize(12);
this->axes_label_font->setWeight(QFont::Light);

this->axis_OX->setTitleFont(*this->axes_label_font);
this->axis_OY->setTitleFont(*this->axes_label_font);
// -----------------------------------------------------------------------

// Tweaking the chart title
// -----------------------------------------------------------------------
```

```
        this->setTitle(*this->title);
        // -----------------------------------------------------------------------

        // Tweaking the series
        // -----------------------------------------------------------------------
        this->data_series->setPen(*this->chart_pen);
        this->data_series->attachAxis(this->axis_OX);
        this->data_series->attachAxis(this->axis_OY);
        this->data_series->setName(*this->legend_label);
        // -----------------------------------------------------------------------



        // Tweaking the Axis OX
        // -----------------------------------------------------------------------
        this->axis_OX->setTickCount(*this->axis_OX_tick_count);
        this->axis_OX->setRange
        (
                *this->scale_OX_left_lim, *this->scale_OX_right_lim
        );
        this->axis_OX->setTitleText(*this->axis_OX_label);
        // -----------------------------------------------------------------------

        // Tweaking the Axis OY
        // -----------------------------------------------------------------------
        this->axis_OY->setTickCount(5); // 3
        (*this->scale_OY_min) = (*this->point_y_component) - 0.1;
        (*this->scale_OY_max) = (*this->point_y_component) + 0.1;
        this->axis_OY->setRange
        (
                *this->scale_OY_min, *this->scale_OY_max
        );
        this->axis_OY->setTitleText(*this->axis_OY_label);
        // -----------------------------------------------------------------------
}


// Function initialises the molecular dynamics experiment on CPU only
void Dynamic_chart::init_MD_experiment_CPU()
{
        // Initialising molecular dynamic simulation
        // -----------------------------------------------------------------------
        initialise_md_experiment_1(this->moldy_experiment, *this->system_energy);
        (*this->moldy_experiment->time_step) = (*this->point_x_component_step);
        (*this->point_y_component) = (*this->system_energy);
        // -----------------------------------------------------------------------

        // Inserting firs point on the plot
        // -----------------------------------------------------------------------
        this->data_series->append(0.0, (*this->point_y_component));
        // -----------------------------------------------------------------------

        this->axis_OY->setTickCount(5); // 3
        (*this->scale_OY_min) = (*this->point_y_component) - 0.5;
        (*this->scale_OY_max) = (*this->point_y_component) + 0.5;
        this->axis_OY->setRange
        (
                *this->scale_OY_min, *this->scale_OY_max
        );
}


// Function executes the molecular dynamics experiment on CPU only
```

```
void Dynamic_chart::run_MD_experiment_CPU()
{
	this->init_MD_experiment_CPU();
	this->timer_CPU->start();
}


// Function executes the molecular dynamics experiment use CPU & GPU
void Dynamic_chart::run_MD_experiment_GPU()
{
	init_values_on_GPU(this->moldy_experiment);
	this->timer_GPU->start();
}


Dynamic_chart::~Dynamic_chart()
{ }


void Dynamic_chart::user_timeout_CPU()
{
	// Calculating data
	// OX
	(*this->point_x_component) += (*this->point_x_component_step);
	// OY
	this->moldy_experiment->run_experiment_brute_force
	(
		*this->point_y_component
	);
	// Changing OY scale
	if ((*this->scale_OY_min) > (*this->point_y_component) ||
		(*this->scale_OY_max) < (*this->point_y_component))
	{
		(*this->scale_OY_min) =
			((*this->scale_OY_min) > (*this->point_y_component)) ?
			(*this->point_y_component) : (*this->scale_OY_min);
		(*this->scale_OY_max) =
			((*this->scale_OY_max) < (*this->point_y_component)) ?
			(*this->point_y_component) : (*this->scale_OY_max);

		// Updating chart OY scale
		this->axis_OY->setRange(*this->scale_OY_min, *this->scale_OY_max);
		this->axis_OY->setTickCount(3);
		this->update();
	}

	// Inserting data
	this->data_series->append
	(
		*this->point_x_component, *this->point_y_component
	);

	// Scrolling chart canvas
	if ((*this->point_x_component) -
		(*this->buffer) >= (*this->scroll_limit))
	{
		this->scroll(this->plotArea().width(), 0);
		(*this->buffer) = (*this->point_x_component);
	}

	// Exit condition
	if ((*this->point_x_component) >= (*this->scale_OX_right_lim))
	{
		this->timer_CPU->stop();
```

```
            qDebug() << "\n\n*** "
                   << QString::fromLocal8Bit("Finished!")
                   << " ***\n\n";
        }
}



void  Dynamic_chart::user_timeout_GPU()
{
        // Calculating data
        // OX
        (*this->point_x_component) += (*this->point_x_component_step);
        // OY
        do_MD_interation_on_GPU(this->moldy_experiment);
        (*this->point_y_component) = (*this->moldy_experiment->system_energy_GPU);
        // Changing OY scale
        if ((*this->scale_OY_min) > (*this->point_y_component) ||
                (*this->scale_OY_max) < (*this->point_y_component))
        {
                (*this->scale_OY_min) =
                        ((*this->scale_OY_min) > (*this->point_y_component)) ?
                        (*this->point_y_component) : (*this->scale_OY_min);
                (*this->scale_OY_max) =
                        ((*this->scale_OY_max) < (*this->point_y_component)) ?
                        (*this->point_y_component) : (*this->scale_OY_max);

                // Updating chart OY scale
                this->axis_OY->setRange(*this->scale_OY_min, *this->scale_OY_max);
                this->axis_OY->setTickCount(3);
                this->update();
        }

        // Inserting data
        this->data_series->append
        (
                *this->point_x_component, *this->point_y_component
        );

        // Scrolling chart canvas
        if ((*this->point_x_component) -
                (*this->buffer) >= (*this->scroll_limit))
        {
                this->scroll(this->plotArea().width(), 0);
                (*this->buffer) = (*this->point_x_component);
        }

        // Exit condition
        if ((*this->point_x_component) >= (*this->scale_OX_right_lim))
        {
                this->timer_CPU->stop();
                qDebug() << "\n\n*** "
                        << QString::fromLocal8Bit("Finished!")
                        << " ***\n\n";
        }
}
```

# ДОДАТОК Р

Лістинг реалізації частини функцій для класу «MD_experiment» мовою програмування C++. Файл *MD_equations.cpp*

```
#include "MD_experiment.cuh"


// Force equations definition
// ---------------------------------------------------------------------------
// Function calculates Lennard-Jones Force value
long double MD_experiment::lennard_jones_force
(
	long double & distance,
	long double & energy,
	long double & equilibrium_distance
)
{
	return (24.0 * energy / equilibrium_distance) *
		(2.0*powl(equilibrium_distance / distance, 13) -
			powl(equilibrium_distance / distance, 7));
}

// Function calculates Coulomb Force value
long double MD_experiment::coulomb_force
(
	long double & distance,
	long double & coulomb_const,
	long double & charge_1,
	long double & charge_2
)
{
	return coulomb_const * ((charge_1 * charge_2) / (powl(distance, 2.0)));
}

// Function calculates Bond Force value
long double MD_experiment::bond_force
(
	long double & distance,
	long double & equilibrium_distance,
	long double & bond_coefficient
)
{
	return -bond_coefficient * (distance - equilibrium_distance);
}

long double MD_experiment::angle_force_left_atom
(
	Water_molecule & molecule,
	long double & equilibrium_angle,
	long double & angle_coefficient,
	long double atom_0_coordinate_component,
	long double atom_1_coordinate_component,
	long double atom_2_coordinate_component,
	long double angle
)
{
	// Local variables defining
	// ---------------------------------------------------------------------
	long double atom_0_x = molecule.get_coordinate(0, 0);
```

```
long double atom_0_y = molecule.get_coordinate(0, 1);
long double atom_0_z = molecule.get_coordinate(0, 2);

long double atom_1_x = molecule.get_coordinate(1, 0);
long double atom_1_y = molecule.get_coordinate(1, 1);
long double atom_1_z = molecule.get_coordinate(1, 2);

long double atom_2_x = molecule.get_coordinate(2, 0);
long double atom_2_y = molecule.get_coordinate(2, 1);
long double atom_2_z = molecule.get_coordinate(2, 2);

// i == 0 (for force formula)
long double dot_product_1_1 =
        get_vector_dot_product
        (
                atom_1_x, atom_1_y, atom_1_z, atom_1_x, atom_1_y, atom_1_z
        );
long double dot_product_1_0 =
        get_vector_dot_product
        (
                atom_1_x, atom_1_y, atom_1_z, atom_0_x, atom_0_y, atom_0_z
        );
long double dot_product_1_2 =
        get_vector_dot_product
        (
                atom_1_x, atom_1_y, atom_1_z, atom_2_x, atom_2_y, atom_2_z
        );
long double dot_product_0_2 =
        get_vector_dot_product
        (
                atom_0_x, atom_0_y, atom_0_z, atom_2_x, atom_2_y, atom_2_z
        );

long double squared_distance_1_0 =
        get_squared_distance
        (
                atom_1_x, atom_1_y, atom_1_z, atom_0_x, atom_0_y, atom_0_z
        );
long double squared_distance_1_2 =
        get_squared_distance
        (
                atom_1_x, atom_1_y, atom_1_z, atom_2_x, atom_2_y, atom_2_z
        );

long double difference_1 = -atom_1_coordinate_component +
        atom_2_coordinate_component;
long double difference_2 = atom_1_coordinate_component -
        atom_0_coordinate_component;

long double dot_products_sum =
        dot_product_1_1 - dot_product_1_0 -
        dot_product_1_2 + dot_product_0_2;
// ---------------------------------------------------------------------------

return (angle_coefficient * std::sinl(
        (angle - equilibrium_angle) * CUDART_PI / 180.0)) /
        powl
        (
                1 -
                ((dot_products_sum * dot_products_sum) /
                 (squared_distance_1_0 * squared_distance_1_2)), 0.5
        ) *
        (
                (difference_1 * squared_distance_1_0) +
```

```
                        (difference_2 * dot_products_sum)
                ) /
                (
                        powl(squared_distance_1_0 * squared_distance_1_2, 0.5) *
                        squared_distance_1_0
                );
}

// Function calculates Angle Force value
long double MD_experiment::angle_force_right_atom
(
        Water_molecule & molecule,
        long double & equilibrium_angle,
        long double & angle_coefficient,
        long double atom_0_coordinate_component,
        long double atom_1_coordinate_component,
        long double atom_2_coordinate_component,
        long double & angle
)
{
        // Local variables defining
        // ------------------------------------------------------------------------
        long double atom_0_x = molecule.get_coordinate(0, 0);
        long double atom_0_y = molecule.get_coordinate(0, 1);
        long double atom_0_z = molecule.get_coordinate(0, 2);

        long double atom_1_x = molecule.get_coordinate(1, 0);
        long double atom_1_y = molecule.get_coordinate(1, 1);
        long double atom_1_z = molecule.get_coordinate(1, 2);

        long double atom_2_x = molecule.get_coordinate(2, 0);
        long double atom_2_y = molecule.get_coordinate(2, 1);
        long double atom_2_z = molecule.get_coordinate(2, 2);

        // i == 2 (for force formula)
        long double dot_product_1_1 =
                this->get_vector_dot_product
                (
                        atom_1_x, atom_1_y, atom_1_z, atom_1_x, atom_1_y, atom_1_z
                );
        long double dot_product_1_0 =
                this->get_vector_dot_product
                (
                        atom_1_x, atom_1_y, atom_1_z, atom_0_x, atom_0_y, atom_0_z
                );
        long double dot_product_1_2 =
                this->get_vector_dot_product
                (
                        atom_1_x, atom_1_y, atom_1_z, atom_2_x, atom_2_y, atom_2_z
                );
        long double dot_product_0_2 =
                this->get_vector_dot_product
                (
                        atom_0_x, atom_0_y, atom_0_z, atom_2_x, atom_2_y, atom_2_z
                );
        long double squared_distance_1_0 =
                this->get_squared_distance
                (
                        atom_1_x, atom_1_y, atom_1_z, atom_0_x, atom_0_y, atom_0_z
                );
        long double squared_distance_1_2 =
                this->get_squared_distance
                (
                        atom_1_x, atom_1_y, atom_1_z, atom_2_x, atom_2_y, atom_2_z
```

```
            );

        long double difference_1 = -atom_1_coordinate_component +
            atom_2_coordinate_component;
        long double difference_2 = atom_1_coordinate_component -
            atom_0_coordinate_component;

        angle = std::acosl
        (
            (this->get_vector_dot_product
            (
                atom_0_x - atom_1_x, atom_0_y - atom_1_y, atom_0_z - atom_1_z,
                atom_2_x - atom_1_x, atom_2_y - atom_1_y, atom_2_z - atom_1_z
            )
            ) /
                (this->get_distance
                (
                    atom_0_x, atom_0_y, atom_0_z, atom_1_x, atom_1_y, atom_1_z
                ) *
                    this->get_distance
                    (
                        atom_2_x, atom_2_y, atom_2_z, atom_1_x, atom_1_y, atom_1_z
                    )
                )
        ) * 180.0 / CUDART_PI;

        long double dot_products_sum =
            dot_product_1_1 - dot_product_1_0 -
            dot_product_1_2 + dot_product_0_2;
        // ------------------------------------------------------------------------

        return (angle_coefficient * std::sinl(
            (angle - equilibrium_angle) * CUDART_PI / 180.0)) /
            powl
            (
                1 -
                ((dot_products_sum * dot_products_sum) /
                (squared_distance_1_0 * squared_distance_1_2)), 0.5
            ) *
            (
            (difference_1 * squared_distance_1_0) +
                (difference_2 * dot_products_sum)
            ) /
            (
                powl(squared_distance_1_0 * squared_distance_1_2, 0.5) *
                squared_distance_1_0
            );
}

// Function calculates Angle Force value
long double MD_experiment::angle_force
(
        long double angle,
        long double & equilibrium_angle,
        long double & angle_coefficient
)
{
        return -angle_coefficient * std::sinl((angle - equilibrium_angle) * CUDART_PI /
180.0);
}
// ----------------------------------------------------------------------------

// Potential equations definition
// ----------------------------------------------------------------------------
```

```
// Function calculates Lennard-Jones Potential value
long double MD_experiment::lennard_jones_potential
(
	long double & distance,
	long double & energy,
	long double & equilibrium_distance
)
{
	return (4.0 * energy) *
		(powl(equilibrium_distance / distance, 12) -
		 powl(equilibrium_distance / distance, 6));
}

// Function calculates Coulomb Potential value
long double MD_experiment::coulomb_potential
(
	long double & distance,
	long double & coulomb_const,
	long double & charge_1,
	long double & charge_2
)
{
	return coulomb_const * ((charge_1 * charge_2) / distance);
}

// Function calculates Bond Potential value
long double MD_experiment::bond_potential
(
	long double & distance,
	long double & equilibrium_distance,
	long double & bond_coefficient
)
{
	return (0.5 * bond_coefficient) * powl(distance - equilibrium_distance, 2);
}

// Function calculates Angle Potential value
long double MD_experiment::angle_potential
(
	long double & angle_value,
	long double & equilibrium_angle,
	long double & angle_coefficient
)
{
	// sin (param*PI/180)
	return -angle_coefficient * (std::cosl
	(
		(angle_value - equilibrium_angle) * (CUDART_PI / 180.0)
	) - 1.0);
}
// ---------------------------------------------------------------------------

// Operations with Atom models
// ---------------------------------------------------------------------------
// Function calculates vector dot product using Atom's radius vectors
long double MD_experiment::get_vector_dot_product
(
	long double vector_1_x, long double vector_1_y, long double vector_1_z,
	long double vector_2_x, long double vector_2_y, long double vector_2_z
)
{
	return (vector_1_x * vector_2_x) + (vector_1_y * vector_2_y) +
		   (vector_1_z * vector_2_z);
}
```

```
// Function calculates squared distance between Atom couple
long double  MD_experiment::get_squared_distance
(
	long double atom_1_x, long double atom_1_y, long double atom_1_z,
	long double atom_2_x, long double atom_2_y, long double atom_2_z
)
{
	return powl(atom_2_x - atom_1_x, 2) + powl(atom_2_y - atom_1_y, 2) +
		powl(atom_2_z - atom_1_z, 2);
}

// Function calculates distance between Atom couple
long double MD_experiment::get_distance
(
	long double atom_1_x, long double atom_1_y, long double atom_1_z,
	long double atom_2_x, long double atom_2_y, long double atom_2_z
)
{
	return powl
		(
			powl(atom_2_x - atom_1_x, 2) +    powl(atom_2_y - atom_1_y, 2) +
			powl(atom_2_z - atom_1_z, 2),
			0.5
		);
}

// Function corrects coordinates
void MD_experiment::coordinate_correction
(
	Water_molecule * mol_ptr, int atom_i, int dim_i
)
{
	// Local variable defining
	// -------------------------------------------------------------------
	long double buffer = mol_ptr->get_coordinate(atom_i, dim_i);
	// -------------------------------------------------------------------

	switch (dim_i)
	{
	case 0:
		if (buffer > (*this->modelling_area_width))
		{
			buffer = fmodl(buffer, *this->modelling_area_width);
		}
		else if (buffer < 0.0)
		{
			buffer = (*this->modelling_area_width) +
				fmodl(buffer, *this->modelling_area_width);
		}
		break;
	case 1:
		if (buffer > (*this->modelling_area_length))
		{
			buffer = fmodl(buffer, *this->modelling_area_length);
		}
		else if (buffer < 0.0)
		{
			buffer = (*this->modelling_area_length) +
				fmodl(buffer, *this->modelling_area_length);
		}
		break;
	case 2:
		if (buffer > (*this->modelling_area_height))
```

```
			{
				buffer = fmodl(buffer, *this->modelling_area_height);
			}
			else if (buffer < 0.0)
			{
				buffer = (*this->modelling_area_height) +
					fmodl(buffer, *this->modelling_area_height);
			}
			break;
		}
		mol_ptr->set_coordinate(atom_i, dim_i, buffer);
}
// Function corrects distance
long double MD_experiment::distance_correction0(long double value, int dim_i)
{
		switch (dim_i)
		{
		case 0:
			if (value >= (0.5*(*this->modelling_area_width)))
			{
				return fmodl(value, *this->modelling_area_width);
			}
			else
			if (value < (-0.5*(*this->modelling_area_width)))
			{
				return fmodl(value, *this->modelling_area_width);
			}
			else
			{
				return value;
			}
			break;
		case 1:
			if (value >= (0.5*(*this->modelling_area_length)))
			{
				return fmodl(value, *this->modelling_area_length);
			}
			else
			if (value < (-0.5*(*this->modelling_area_length)))
			{
				return fmodl(value, *this->modelling_area_length);
			}
			else
			{
				return value;
			}
			break;
		case 2:
			if (value >= (0.5*(*this->modelling_area_height)))
			{
				return fmodl(value, *this->modelling_area_height);
			}
			else
			if (value < (-0.5*(*this->modelling_area_height)))
			{
				return fmodl(value, *this->modelling_area_height);
			}
			else
			{
				return value;
			}
			break;
		}
}
```

```
// Function corrects distance
long double MD_experiment::distance_correction(long double value, int dim_i)
{
	switch (dim_i)
	{
	case 0:
		if (value > (0.5*(*this->modelling_area_width)))
		{
			return ((*this->modelling_area_width) - value);
		}
		else if (value < (-0.5*(*this->modelling_area_width)))
		{
			return -((*this->modelling_area_width) + value);
		}
		else
		{
			return value;
		}
		break;
	case 1:
		if (value > (0.5*(*this->modelling_area_length)))
		{
			return ((*this->modelling_area_length) - value);
		}
		else if (value < (-0.5*(*this->modelling_area_length)))
		{
			return -((*this->modelling_area_length) + value);
		}
		else
		{
			return value;
		}
		break;
	case 2:
		if (value > (0.5*(*this->modelling_area_height)))
		{
			return ((*this->modelling_area_height) - value);
		}
		else if (value < (-0.5*(*this->modelling_area_height)))
		{
			return -((*this->modelling_area_height) + value);
		}
		else
		{
			return value;
		}
		break;
	}
}

// Function calculates the angle between bonds
long double MD_experiment::get_angle(Water_molecule & mol)
{
	// Local variables defining
	// ----------------------------------------------------------------------------
	long double v1x{ 0.0 }, v1y{ 0.0 }, v1z{ 0.0 },
			    v2x{ 0.0 }, v2y{ 0.0 }, v2z{ 0.0 };
	// ----------------------------------------------------------------------------

	// Calculating vector components
	// ----------------------------------------------------------------------------
	// Vector-1
	v1x = mol.get_coordinate(0, 0) - mol.get_coordinate(1, 0);
	v1y = mol.get_coordinate(0, 1) - mol.get_coordinate(1, 1);
```

```
        v1z = mol.get_coordinate(0, 2) - mol.get_coordinate(1, 2);

        // Vector-2
        v2x = mol.get_coordinate(2, 0) - mol.get_coordinate(1, 0);
        v2y = mol.get_coordinate(2, 1) - mol.get_coordinate(1, 1);
        v2z = mol.get_coordinate(2, 2) - mol.get_coordinate(1, 2);
        // ------------------------------------------------------------------------

        // Calculating the angle value
        return std::acosl
        (
               (v1x*v2x + v1y*v2y + v1z*v2z) /
               ((std::powl(v1x*v1x + v1y*v1y + v1z*v1z, 0.5)) *
                (std::powl(v2x*v2x + v2y*v2y + v2z*v2z, 0.5)))
        ) * 180.0 / CUDART_PI;
}
// ------------------------------------------------------------------------------

// Energy calculations
// ------------------------------------------------------------------------------
// Function calculates kinetic energy value
long double MD_experiment::get_kinetic_energy
(
        long double squared_velocity,
        long double & mass
)
{
        return mass * (0.5 * squared_velocity);
}
// ------------------------------------------------------------------------------
```

# ДОДАТОК С

Лістинг реалізації класу «MD_experiment» мовою програмування С++. Файл *MD_experiment.cu*

```
#include "MD_experiment.cuh"


// Static variables defining
// ----------------------------------------------------------------------------
// Dimension count
int * MD_experiment::dimension_count = new int{3};

// Start atom coordinates for molecule model
long double * MD_experiment::start_coordinate_atom_1 =
new long double[*MD_experiment::dimension_count];
long double * MD_experiment::start_coordinate_atom_2 =
new long double[*MD_experiment::dimension_count];
long double * MD_experiment::start_coordinate_atom_3 =
new long double[*MD_experiment::dimension_count];

// Modelling subarea shape parameters
long double * MD_experiment::modelling_subarea_width = new long double{ 0.0 };
long double * MD_experiment::modelling_subarea_length = new long double{ 0.0 };
long double * MD_experiment::modelling_subarea_height = new long double{ 0.0 };

// Maximum particle count
size_t * MD_experiment::max_particle_count = new size_t{ 0 };
// ----------------------------------------------------------------------------


// MD_experiment methods defining
// ----------------------------------------------------------------------------
MD_experiment::MD_experiment()
	: array_size{ nullptr }, particle_array{ nullptr }
{
	this->time_step = new long double{ 0.0 };
}


MD_experiment::~MD_experiment()
{
	if (this->particle_array != nullptr)
	{
		delete[] this->particle_array;
		delete this->array_size;
		delete this->modelling_area_width;
		delete this->modelling_area_height;
		delete this->modelling_area_length;
	}
}


void MD_experiment::calculate_basic_atom_coordinates()
{
	// Defining local variables
	// ----------------------------------------------------------------------------
	long double sin_equilibrium_angle =
		std::sin
		(
```

```
                (*Water_molecule::equilibrium_angle) * CUDART_PI / 180.0
                );
        long double cos_equilibrium_angle =
                std::cos
                (
                (*Water_molecule::equilibrium_angle) * CUDART_PI / 180.0
                );
        long double bias = -2.0;
        // -----------------------------------------------------------------------

        // Defining first atom basic coordinates
        MD_experiment::start_coordinate_atom_3[0] =
                (*Water_molecule::equilibrium_distance_bond_poten);
        MD_experiment::start_coordinate_atom_3[1] = 0.0;
        MD_experiment::start_coordinate_atom_3[2] = 0.0;

        // Defining second atom basic coordinates
        MD_experiment::start_coordinate_atom_2[0] = 0.0;
        MD_experiment::start_coordinate_atom_2[1] = 0.0;
        MD_experiment::start_coordinate_atom_2[2] = 0.0;

        // Defining third atom basic coordinates (rotation counterclockwise)
        MD_experiment::start_coordinate_atom_1[0] =
                cos_equilibrium_angle * MD_experiment::start_coordinate_atom_3[0] +
                -sin_equilibrium_angle * MD_experiment::start_coordinate_atom_3[1];
        MD_experiment::start_coordinate_atom_1[1] =
                sin_equilibrium_angle * MD_experiment::start_coordinate_atom_3[0] +
                cos_equilibrium_angle * MD_experiment::start_coordinate_atom_3[1];
        MD_experiment::start_coordinate_atom_1[2] = 0.0;

        MD_experiment::start_coordinate_atom_1[0] += bias;
        MD_experiment::start_coordinate_atom_2[0] += bias;
        MD_experiment::start_coordinate_atom_3[0] += bias;
} // void MD_experiment::calculate_start_atom_coordinates()

// Calculating of modelling subarea parameters
void MD_experiment::calculate_modelling_subarea_params()
{
        // Local variables defining
        // -----------------------------------------------------------------------
        long double buffer = 0.0;
        long double max_distance = 0.0;
        long double offset_OX =
                8.0 * (*Water_molecule::equilibrium_distance_LJ_OO);
        long double offset_OY =
                8.0 * (*Water_molecule::equilibrium_distance_LJ_OO);
        long double offset_OZ =
                8.0 * (*Water_molecule::equilibrium_distance_LJ_OO);
        // -----------------------------------------------------------------------

        // Searching the maximum value of the distance between the atoms of
        // molecule
        // -----------------------------------------------------------------------
        // squared distance between Atom-1 and Atom-2
        max_distance = (*Water_molecule::equilibrium_distance_bond_poten);

        // distance between Atom-1 and Atom-3
        buffer =
                powl(MD_experiment::start_coordinate_atom_1[0] -
                        MD_experiment::start_coordinate_atom_3[0], 2) +
                powl(MD_experiment::start_coordinate_atom_1[1] -
                        MD_experiment::start_coordinate_atom_3[1], 2) +
                powl(MD_experiment::start_coordinate_atom_1[2] -
                        MD_experiment::start_coordinate_atom_3[2], 2);
```

```
        buffer = powl(buffer, 0.5);

        // checking max distance value
        max_distance = (buffer > max_distance) ? (buffer) : (max_distance);
        // ---------------------------------------------------------------------------

        // Calculating subarea shape parameters
        // ---------------------------------------------------------------------------
        (*MD_experiment::modelling_subarea_width) = 2.0 * max_distance + offset_OX;
        (*MD_experiment::modelling_subarea_length) = 2.0 * max_distance + offset_OY;
        (*MD_experiment::modelling_subarea_height) = 2.0 * max_distance + offset_OZ;

        // ---------------------------------------------------------------------------
} // void MD_experiment::calculate_modelling_subarea()


// Function calculates maximum particle count
void MD_experiment::calculate_max_particle_count()
{
        // !!! Dangerous casting !!!
        (*MD_experiment::max_particle_count) = (size_t)
                (
                        std::floorl
                        (
                        (*this->modelling_area_width) /
                                (*MD_experiment::modelling_subarea_width)
                        ) *
                        std::floorl
                        (
                        (*this->modelling_area_length) /
                                (*MD_experiment::modelling_subarea_length)
                        ) *
                        std::floorl
                        (
                        (*this->modelling_area_height) /
                                (*MD_experiment::modelling_subarea_height)
                        )
                );
} // void MD_experiment::calculate_max_particle_count()


// Function prepares different constant values. Step #0
void MD_experiment::setup_constants()
{
        // Preparing basic molecular atom coordinates
        MD_experiment::calculate_basic_atom_coordinates();

        // Preparing modelling subarea sizes
        MD_experiment::calculate_modelling_subarea_params();
} // void MD_experiment::setup_constants()


// Function sets the modelling area sizes. Step #1
void MD_experiment::set_modelling_area_sizes
(
        long double width, long double length, long double height
)
{
        this->modelling_area_width = new long double(width);
        this->modelling_area_length = new long double(length);
        this->modelling_area_height = new long double(height);
}
```

```
// Function sets the particle count. Step #2
void MD_experiment::set_particle_count(size_t particle_count)
{
	this->array_size = new size_t{ particle_count };
	this->particle_count = this->array_size;
	this->particle_array = new Water_molecule[*this->array_size];
}

// Step #3: Defining particle coordinates
void MD_experiment::set_particle_coordinates()
{
	// Local variables defining
	// ------------------------------------------------------------------------
	long double average_molecule_point_OX = 0.0;
	long double average_molecule_point_OY = 0.0;
	long double average_molecule_point_OZ = 0.0;

	long double average_subarea_point_OX = 0.0;
	long double average_subarea_point_OY = 0.0;
	long double average_subarea_point_OZ = 0.0;

	long double moving_vector_OX = 0.0;
	long double moving_vector_OY = 0.0;
	long double moving_vector_OZ = 0.0;

	int subareas_by_OX =
		(int)(std::floorl
		((*this->modelling_area_width) /
			(*MD_experiment::modelling_subarea_width)));
	int subareas_by_OY =
		(int)(std::floorl
		((*this->modelling_area_length) /
			(*MD_experiment::modelling_subarea_length)));
	int subareas_by_OZ =
		(int)(std::floorl
		((*this->modelling_area_height) /
			(*MD_experiment::modelling_subarea_height)));

	// Number of subareas
	int subarea_OX_i = 0, subarea_OY_i = 0, subarea_OZ_i = 0;
	int mol_i = 0;
	// ------------------------------------------------------------------------

	// Calculating average point for Water molecule
	// ------------------------------------------------------------------------
	average_molecule_point_OX =
		(MD_experiment::start_coordinate_atom_1[0] +
		 MD_experiment::start_coordinate_atom_2[0] +
		 MD_experiment::start_coordinate_atom_3[0]) / 3.0;

	average_molecule_point_OY =
		(MD_experiment::start_coordinate_atom_1[1] +
		 MD_experiment::start_coordinate_atom_2[1] +
		 MD_experiment::start_coordinate_atom_3[1]) / 3.0;

	average_molecule_point_OZ =
		(MD_experiment::start_coordinate_atom_1[2] +
		 MD_experiment::start_coordinate_atom_2[2] +
		 MD_experiment::start_coordinate_atom_3[2]) / 3.0;
	// ------------------------------------------------------------------------

	// Calculating coordinates for molecules
	// ------------------------------------------------------------------------
	for (Water_molecule * mol_ptr = this->particle_array,
```

```
            *mol_ptr_end = this->particle_array + (*this->array_size);
            (mol_ptr < mol_ptr_end); ++mol_ptr)
       {
            // Resetting values
            average_subarea_point_OX = 0.0;
            average_subarea_point_OY = 0.0;
            average_subarea_point_OZ = 0.0;

            // Calculating molecule index
            mol_i = (int)(mol_ptr - this->particle_array);

            // Calculating subarea indexes
            subarea_OX_i = mol_i % subareas_by_OX;
            subarea_OY_i = (mol_i / subareas_by_OX) % subareas_by_OY;
            subarea_OZ_i = (mol_i / (subareas_by_OX * subareas_by_OY)) % subareas_by_OZ;

            // Calculating average point of subarea
            // OX
            average_subarea_point_OX =
0.5*(*MD_experiment::modelling_subarea_width)*(2.0*subarea_OX_i + 1.0);
            // OY
            average_subarea_point_OY =
0.5*(*MD_experiment::modelling_subarea_length)*(2.0*subarea_OY_i + 1.0);
            // OZ
            average_subarea_point_OZ =
0.5*(*MD_experiment::modelling_subarea_height)*(2.0*subarea_OZ_i + 1.0);

            // Calculating moving vector components
            moving_vector_OX =
                   average_subarea_point_OX - average_molecule_point_OX;
            moving_vector_OY =
                   average_subarea_point_OY - average_molecule_point_OY;
            moving_vector_OZ =
                   average_subarea_point_OZ - average_molecule_point_OZ;

            // Moving molecue to the center of the area with OX_i, OY_i, OZ_i
            // Atom -> 0
            (*mol_ptr).set_coordinate
            (
                   0, 0,
                   MD_experiment::start_coordinate_atom_1[0] + moving_vector_OX
            );
            (*mol_ptr).set_coordinate
            (
                   0, 1,
                   MD_experiment::start_coordinate_atom_1[1] + moving_vector_OY
            );
            (*mol_ptr).set_coordinate
            (
                   0, 2,
                   MD_experiment::start_coordinate_atom_1[2] + moving_vector_OZ
            );
            // Atom -> 1
            (*mol_ptr).set_coordinate
            (
                   1, 0,
                   MD_experiment::start_coordinate_atom_2[0] + moving_vector_OX
            );
            (*mol_ptr).set_coordinate
            (
                   1, 1,
                   MD_experiment::start_coordinate_atom_2[1] + moving_vector_OY
            );
            (*mol_ptr).set_coordinate
```

```
                (
                        1, 2,
                        MD_experiment::start_coordinate_atom_2[2] + moving_vector_OZ
                );
                // Atom -> 2
                (*mol_ptr).set_coordinate
                (
                        2, 0,
                        MD_experiment::start_coordinate_atom_3[0] + moving_vector_OX
                );
                (*mol_ptr).set_coordinate
                (
                        2, 1,
                        MD_experiment::start_coordinate_atom_3[1] + moving_vector_OY
                );
                (*mol_ptr).set_coordinate
                (
                        2, 2,
                        MD_experiment::start_coordinate_atom_3[2] + moving_vector_OZ
                );
        }
        // ------------------------------------------------------------------------
}


// Step #4: Defining particle velocities
void MD_experiment::set_particle_velocities()
{
        // Local variables defining
        // ------------------------------------------------------------------------
        int atom_i = 0;
        Water_molecule * mol_ptr = this->particle_array,
                * mol_end_ptr = this->particle_array + (*this->array_size);
        // ------------------------------------------------------------------------

        for (; mol_ptr < mol_end_ptr; ++mol_ptr)
        {
                for (atom_i = 0; atom_i < (*Water_molecule::atom_count); ++atom_i)
                {
                        (*mol_ptr).set_velocity(atom_i, 0, 0.0);
                        (*mol_ptr).set_velocity(atom_i, 1, 0.0);
                        (*mol_ptr).set_velocity(atom_i, 2, 0.0);
                }
        }
}


// Step #5: Defining particle accelerations. Brute Force
/*
void MD_experiment::set_particle_accelerations_brute_force()
{
        // File: 'MD_experiment_Brute_Force_iteration.cu'
}
*/

// Function shows Molecular Dynamics Experiment parameters
void MD_experiment::show_experiment_parameters()
{
        std::cout << '\n' << '\n'
                << "*** EXPERIMENT: CURRENT PARAMETER VALUES ***"
                << '\n' << '\n';

        std::cout << "\t\t** Modelling area shape **" << '\n';
        std::cout << "Width:  " << (*this->modelling_area_width) << '\n';
```

```
        std::cout << "Length: " << (*this->modelling_area_length) << '\n';
        std::cout << "Height: " << (*this->modelling_area_height) << '\n';
        // std::cout << '\n';

        std::cout << "\t\t** Particles count in the system **" << '\n';
        std::cout << "Particles count: " << (*this->particle_count) << '\n';
        // std::cout << '\n';
}
// ---------------------------------------------------------------------------
```

# ДОДАТОК Т

Лістинг реалізації алгоритму моделювання динаміки системи без радіусу відсікання мовою програмування С++. Файл *MD_experiment_Brute_Force_iteration.cu*

```
#include "MD_experiment.cuh"


long double MD_experiment::upd_particle_accelerations_brute_force()
{
	// Local variables defining
	// -----------------------------------------------------------------
	int outer_mol_i = 0, inner_mol_i = 0;
	int outer_atom_i = 0, inner_atom_i = 0, dim_i = 0;
	long double reset_value = 0.0, force_buffer = 0.0, distance_buffer = 0.0;
	long double charge_1_buffer = 0.0, charge_2_buffer = 0.0, angle = 0.0;
	long double dx = 0.0, dy = 0.0, dz = 0.0, vx = 0.0, vy = 0.0, vz = 0.0;
	long double kinetic_energy = 0.0, potential_energy = 0.0;
	char atom_1_type = '\0', atom_2_type = '\0';
	// -----------------------------------------------------------------

	// Setting 'Forces' and 'Accelerations' as 0.0
	for (outer_mol_i = 0; outer_mol_i < (*this->array_size); ++outer_mol_i)
	{
		for (outer_atom_i = 0; outer_atom_i < (*Water_molecule::atom_count);
			++outer_atom_i)
		{
			for (dim_i = 0; dim_i < (*MD_experiment::dimension_count); ++dim_i)
			{
				// Force
				this->particle_array[outer_mol_i].set_force(outer_atom_i, dim_i,
reset_value);

				// Acceleration
				this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
dim_i, reset_value);
			}
		}
	}

	// Calculating forces
	for (outer_mol_i = 0; outer_mol_i < (*this->array_size); ++outer_mol_i)
	{
		// Intramolecular forces

		// Angle Force
		// -----------------------------------------------------------------
		outer_atom_i = 0;
		inner_atom_i = 2;

		// Distance calculating
		dx = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 0) -
			this->particle_array[outer_mol_i].get_coordinate(inner_atom_i, 0);
		dx = this->distance_correction(dx, 0);
		dy = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 1) -
			this->particle_array[outer_mol_i].get_coordinate(inner_atom_i, 1);
		dy = this->distance_correction(dy, 1);
		dz = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 2) -
```

```
                this->particle_array[outer_mol_i].get_coordinate(inner_atom_i, 2);
        dz = this->distance_correction(dz, 2);
        distance_buffer = powl(dx*dx + dy * dy + dz * dz, 0.5);

        // Calculating angle value
        angle = this->get_angle(this->particle_array[outer_mol_i]);

        // Atom-0 Force-OX
        force_buffer = this->angle_force_left_atom
        (
                this->particle_array[outer_mol_i],
                (*Water_molecule::equilibrium_angle),
                (*Water_molecule::k_coef_angle_poten),
                this->particle_array[outer_mol_i].get_coordinate(0, 0),
                this->particle_array[outer_mol_i].get_coordinate(1, 0),
                this->particle_array[outer_mol_i].get_coordinate(2, 0),
                angle
        );
        // Updating force-OX for atom-0
        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 0, force_buffer);
        // Updating force-OX for atom-2
        this->particle_array[outer_mol_i].upd_force(inner_atom_i, 0, -force_buffer);

        // Atom-0 Force-OY
        force_buffer = this->angle_force_left_atom
        (
                this->particle_array[outer_mol_i],
                (*Water_molecule::equilibrium_angle),
                (*Water_molecule::k_coef_angle_poten),
                this->particle_array[outer_mol_i].get_coordinate(0, 1),
                this->particle_array[outer_mol_i].get_coordinate(1, 1),
                this->particle_array[outer_mol_i].get_coordinate(2, 1),
                angle
        );
        // Updating force-OY for atom-0          this-
>particle_array[outer_mol_i].upd_force(outer_atom_i, 1, force_buffer);
        // Updating force-OY for atom-2
        this->particle_array[outer_mol_i].upd_force(inner_atom_i, 1, -force_buffer);

        // Atom-0 Force-OZ
        force_buffer = this->angle_force_left_atom
        (
                this->particle_array[outer_mol_i],
                (*Water_molecule::equilibrium_angle),
                (*Water_molecule::k_coef_angle_poten),
                this->particle_array[outer_mol_i].get_coordinate(0, 2),
                this->particle_array[outer_mol_i].get_coordinate(1, 2),
                this->particle_array[outer_mol_i].get_coordinate(2, 2),
                angle
        );
        // Updating force-OZ for atom-0
        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 2, force_buffer);
        // Updating force-OZ for atom-2
        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 2, -force_buffer);

        // Potential
        potential_energy = potential_energy + (1.0*(this->angle_potential(angle,
(*Water_molecule::equilibrium_angle), (*Water_molecule::k_coef_angle_poten))));
        // ---------------------------------------------------------------------

        // Bond Force
        for (outer_atom_i = 0, inner_atom_i = 1; outer_atom_i < 2;
                ++outer_atom_i, ++inner_atom_i)
        {
```

```
                        // Distance calculating
                        dx = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 0)
-
                                this->particle_array[outer_mol_i].get_coordinate(inner_atom_i,
0);
                        dx = this->distance_correction(dx, 0);
                        dy = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 1)
-
                                this->particle_array[outer_mol_i].get_coordinate(inner_atom_i,
1);
                        dy = this->distance_correction(dy, 1);
                        dz = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 2)
-
                                this->particle_array[outer_mol_i].get_coordinate(inner_atom_i,
2);
                        dz = this->distance_correction(dz, 2);
                        distance_buffer = powl(dx*dx + dy * dy + dz * dz, 0.5);

                        // Bond link: FORCE
                        force_buffer = this->bond_force(distance_buffer,
(*Water_molecule::equilibrium_distance_bond_poten), (*Water_molecule::k_coef_bond_poten));

                        // Bond link: Potential
                        potential_energy = potential_energy + 1.0*(this-
>bond_potential(distance_buffer, (*Water_molecule::equilibrium_distance_bond_poten),
(*Water_molecule::k_coef_bond_poten)) /
                                (*this->array_size));

                        // Updating Force values
                        // OX
                        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 0, dx /
distance_buffer * force_buffer);
                        // OX -- pair
                        this->particle_array[outer_mol_i].upd_force(inner_atom_i, 0, -dx /
distance_buffer * force_buffer);
                        // OY
                        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 1, dy /
distance_buffer * force_buffer);
                        // OY -- pair
                        this->particle_array[outer_mol_i].upd_force(inner_atom_i, 1, -dy /
distance_buffer * force_buffer);
                        // OZ
                        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 2, dz /
distance_buffer * force_buffer);
                        // OZ -- pair
                        this->particle_array[outer_mol_i].upd_force(inner_atom_i, 2, -dz /
distance_buffer * force_buffer);
                }

                // Inner molecules cycle
                for (inner_mol_i = outer_mol_i + 1; inner_mol_i < (*this->array_size);
                        ++inner_mol_i)
                {
                        // Outer atom cycle
                        for (outer_atom_i = 0;
                                outer_atom_i < (*Water_molecule::atom_count); ++outer_atom_i)
                        {
                                // Checking first atom type
                                atom_1_type = this-
>particle_array[outer_mol_i].get_type(outer_atom_i);

                                // Checking first atom charge
                                charge_1_buffer = (atom_1_type == (*Water_molecule::type_O)) ?
                                        (*Water_molecule::q_charge_O) :
```

```
                                    (*Water_molecule::q_charge_H);

                              // Inner atom cycle
                              for (inner_atom_i = 0;
                                    inner_atom_i < (*Water_molecule::atom_count);
                                    ++inner_atom_i)
                              {
                                    // Rewritting
                                    force_buffer = 0.0;

                                    // Checking second atom type
                                    atom_2_type = this-
>particle_array[inner_mol_i].get_type(inner_atom_i);

                                    // Checking second atom charge
                                    charge_2_buffer = (atom_2_type ==
(*Water_molecule::type_O)) ?
                                          (*Water_molecule::q_charge_O) :
                                          (*Water_molecule::q_charge_H);

                                    // Distance calculating
                                    dx = this-
>particle_array[outer_mol_i].get_coordinate(outer_atom_i, 0) -
                                          this-
>particle_array[inner_mol_i].get_coordinate(inner_atom_i, 0);
                                    dx = this->distance_correction(dx, 0);
                                    dy = this-
>particle_array[outer_mol_i].get_coordinate(outer_atom_i, 1) -
                                          this-
>particle_array[inner_mol_i].get_coordinate(inner_atom_i, 1);
                                    dy = this->distance_correction(dy, 1);
                                    dz = this-
>particle_array[outer_mol_i].get_coordinate(outer_atom_i, 2) -
                                          this-
>particle_array[inner_mol_i].get_coordinate(inner_atom_i, 2);
                                    dz = this->distance_correction(dz, 2);
                                    distance_buffer = powl(dx*dx + dy * dy + dz * dz, 0.5);

                                    // /*
                                    // Lennard-Jones
                                    if ((atom_1_type == (*Water_molecule::type_O)) &&
                                          (atom_2_type == (*Water_molecule::type_O)))
                                    {
                                          force_buffer = this->lennard_jones_force
                                          (
                                                distance_buffer,
(*Water_molecule::energy_LJ_OO),

      (*Water_molecule::equilibrium_distance_LJ_OO)
                                          );
                                          potential_energy = potential_energy + 1.0*((this-
>lennard_jones_potential

                                          (
                                                distance_buffer,
(*Water_molecule::energy_LJ_OO),

      (*Water_molecule::equilibrium_distance_LJ_OO)
                                          )) / (*this->array_size));
                                    }
                                    else
                                    if ((atom_1_type == (*Water_molecule::type_H)) &&
                                          (atom_2_type == (*Water_molecule::type_H)))
                                    {
                                          force_buffer = this->lennard_jones_force
```

```
                                        (
                                                distance_buffer,
(*Water_molecule::energy_LJ_HH),

        (*Water_molecule::equilibrium_distance_LJ_HH)
                                        );
                                        potential_energy = potential_energy + 1.0*((this-
>lennard_jones_potential
                                        (
                                                distance_buffer,
(*Water_molecule::energy_LJ_HH),

        (*Water_molecule::equilibrium_distance_LJ_HH)
                                        )) / (*this->array_size));
                                }
                                else
                                {
                                        force_buffer = this->lennard_jones_force
                                        (
                                                distance_buffer,
(*Water_molecule::energy_LJ_OH),

        (*Water_molecule::equilibrium_distance_LJ_OH)
                                        );
                                        potential_energy = potential_energy + 1.0*((this-
>lennard_jones_potential
                                        (
                                                distance_buffer,
(*Water_molecule::energy_LJ_OH),

        (*Water_molecule::equilibrium_distance_LJ_OH)
                                        )) / (*this->array_size));
                                }
                                // Coulomb
                                // Force
                                force_buffer = force_buffer + this-
>coulomb_force(distance_buffer, (*Water_molecule::k_coef_coulomb_potential),
charge_1_buffer, charge_2_buffer);
                                // Potential
                                potential_energy = potential_energy +
                                        (1.0*(this->coulomb_potential(distance_buffer,
(*Water_molecule::k_coef_coulomb_potential), charge_1_buffer, charge_2_buffer)) /
                                        (*this->array_size));

                                // Updating Force values
                                // OX
                                this->particle_array[outer_mol_i].upd_force(outer_atom_i,
0, dx/distance_buffer*force_buffer);
                                // OX - for pair
                                this->particle_array[inner_mol_i].upd_force(inner_atom_i,
0, -dx/distance_buffer*force_buffer);
                                // OY
                                this->particle_array[outer_mol_i].upd_force(outer_atom_i,
1, dy/distance_buffer*force_buffer);
                                // OY - for pair
                                this->particle_array[inner_mol_i].upd_force(inner_atom_i,
1, -dy/distance_buffer*force_buffer);
                                // OZ
                                this->particle_array[outer_mol_i].upd_force(outer_atom_i,
2, dz/distance_buffer*force_buffer);
                                // OZ - for pair
                                this->particle_array[inner_mol_i].upd_force(inner_atom_i,
2, -dz/distance_buffer*force_buffer);
                                // */
```

```
                    }
                }
            }
            // Calculating 'Kinetic Energy' and 'Accelerations'
            for (outer_atom_i = 0; outer_atom_i < (*Water_molecule::atom_count);
                    ++outer_atom_i)
            {
                // Getting atom type
                atom_1_type = this->particle_array[outer_mol_i].get_type(outer_atom_i);

                // Calculating velocity components
                vx = this->particle_array[outer_mol_i].get_velocity(outer_atom_i, 0);
                vy = this->particle_array[outer_mol_i].get_velocity(outer_atom_i, 1);
                vz = this->particle_array[outer_mol_i].get_velocity(outer_atom_i, 2);

                if (atom_1_type == (*Water_molecule::type_O))
                {
                    kinetic_energy = kinetic_energy + (
                        (this->get_kinetic_energy(vx*vx + vy * vy + vz * vz,
(*Water_molecule::mass_O))) /
                        (*this->array_size));

                    // OX
                    this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
0, this->particle_array[outer_mol_i].get_force(outer_atom_i, 0) /
(*Water_molecule::mass_O));
                    // OY
                    this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
1, this->particle_array[outer_mol_i].get_force(outer_atom_i, 1) /
(*Water_molecule::mass_O));
                    // OZ
                    this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
2, this->particle_array[outer_mol_i].get_force(outer_atom_i, 2) /
(*Water_molecule::mass_O));
                }
                else if (atom_1_type == (*Water_molecule::type_H))
                {
                    kinetic_energy = kinetic_energy +
                        ((this->get_kinetic_energy(vx*vx + vy * vy + vz * vz,
(*Water_molecule::mass_H))) /
                        (*this->array_size));

                    // OX
                    this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
0, this->particle_array[outer_mol_i].get_force(outer_atom_i, 0) /
(*Water_molecule::mass_H));
                    // OY
                    this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
1, this->particle_array[outer_mol_i].get_force(outer_atom_i, 1) /
(*Water_molecule::mass_H));
                    // OZ
                    this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
2, this->particle_array[outer_mol_i].get_force(outer_atom_i, 2) /
(*Water_molecule::mass_H));
                }
            }
        }

        return (kinetic_energy + potential_energy);
}


void MD_experiment::run_experiment_brute_force(long double & system_energy)
{
```

```
// Local variables defining
// ----------------------------------------------------------------------
Water_molecule * mol_ptr = this->particle_array,
        *mol_end_ptr = this->particle_array + (*this->array_size);
int atom_i = 0, dim_i = 0;
// ----------------------------------------------------------------------

// STEP #1: Calculating new coordinate values and
//          Velocity values partial update
// ----------------------------------------------------------------------
// Molecules cycle
for (; mol_ptr < mol_end_ptr; ++mol_ptr)
{
        // Atoms cycle
        for (atom_i = 0; atom_i < (*Water_molecule::atom_count); ++atom_i)
        {
                // Dimensions cycle
                for (dim_i = 0; dim_i < (*MD_experiment::dimension_count); ++dim_i)
                {
                        // Coordinate update
                        mol_ptr->upd_coordinate
                        (
                                atom_i, dim_i,
                                (*this->time_step)*(mol_ptr->get_velocity(atom_i, dim_i))
+
                                0.5 * powl(*this->time_step, 2) *
                                (mol_ptr->get_acceleration(atom_i, dim_i))
                        );

                        // Coordinate correction
                        this->coordinate_correction(mol_ptr, atom_i, dim_i);

                        // Partial velocity update
                        mol_ptr->upd_velocity
                        (
                                atom_i, dim_i,
                                0.5 * (*this->time_step) *
                                (mol_ptr->get_acceleration(atom_i, dim_i))
                        );
                }
        }
}
// ----------------------------------------------------------------------

// STEP #2: Accelerate values update
// ----------------------------------------------------------------------
system_energy = this->upd_particle_accelerations_brute_force();
// ----------------------------------------------------------------------

// STEP #3: Velocity values final partial update
// ----------------------------------------------------------------------
for (mol_ptr = this->particle_array; mol_ptr < mol_end_ptr; ++mol_ptr)
{
        // Atoms cycle
        for (atom_i = 0; atom_i < (*Water_molecule::atom_count); ++atom_i)
        {
                // Dimensions cycle
                for (dim_i = 0; dim_i < (*MD_experiment::dimension_count); ++dim_i)
                {
                        // Partial velocity update
                        mol_ptr->upd_velocity
                        (
                                atom_i, dim_i,
                                0.5 * (*this->time_step) *
```

```
                        (mol_ptr->get_acceleration(atom_i, dim_i))
                    );
                }
            }
        }
        // ----------------------------------------------------------------------
}
```

# ДОДАТОК У

Лістинг реалізації алгоритму моделювання динаміки системи з радіусом відсікання мовою програмування C++. Файл *MD_experiment_Brute_Force_iteration_with_R-cut.cpp*

```
#include "MD_experiment.cuh"

long double MD_experiment::upd_particle_accelerations_brute_force_with_r(long double r_cut)
{
	// Local variables defining
	// ---------------------------------------------------------------------
	int outer_mol_i = 0, inner_mol_i = 0;
	int outer_atom_i = 0, inner_atom_i = 0, dim_i = 0;
	long double reset_value = 0.0, force_buffer = 0.0, distance_buffer = 0.0;
	long double charge_1_buffer = 0.0, charge_2_buffer = 0.0, angle = 0.0;
	long double dx = 0.0, dy = 0.0, dz = 0.0, vx = 0.0, vy = 0.0, vz = 0.0;
	long double kinetic_energy = 0.0, potential_energy = 0.0;
	char atom_1_type = '\0', atom_2_type = '\0';
	// ---------------------------------------------------------------------

	// Setting 'Forces' and 'Accelerations' as 0.0
	for (outer_mol_i = 0; outer_mol_i < (*this->array_size); ++outer_mol_i)
	{
		for (outer_atom_i = 0; outer_atom_i < (*Water_molecule::atom_count);
			++outer_atom_i)
		{
			for (dim_i = 0; dim_i < (*MD_experiment::dimension_count); ++dim_i)
			{
				// Force
				this->particle_array[outer_mol_i].set_force(outer_atom_i, dim_i,
reset_value);

				// Acceleration
				this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
dim_i, reset_value);
			}
		}
	}

	// Calculating forces
	for (outer_mol_i = 0; outer_mol_i < (*this->array_size); ++outer_mol_i)
	{
		// Intramolecular forces

		// Angle Force
		// ---------------------------------------------------------------------
		outer_atom_i = 0;
		inner_atom_i = 2;

		// Distance calculating
		dx = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 0) -
			this->particle_array[outer_mol_i].get_coordinate(inner_atom_i, 0);
		dx = this->distance_correction(dx, 0);
		dy = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 1) -
			this->particle_array[outer_mol_i].get_coordinate(inner_atom_i, 1);
		dy = this->distance_correction(dy, 1);
		dz = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 2) -
```

```
			this->particle_array[outer_mol_i].get_coordinate(inner_atom_i, 2);
		dz = this->distance_correction(dz, 2);
		distance_buffer = powl(dx*dx + dy * dy + dz * dz, 0.5);

		// Calculating angle value
		angle = this->get_angle(this->particle_array[outer_mol_i]);

		// Atom-0 Force-OX
		force_buffer = this->angle_force_left_atom
		(
			this->particle_array[outer_mol_i],
			(*Water_molecule::equilibrium_angle),
			(*Water_molecule::k_coef_angle_poten),
			this->particle_array[outer_mol_i].get_coordinate(0, 0),
			this->particle_array[outer_mol_i].get_coordinate(1, 0),
			this->particle_array[outer_mol_i].get_coordinate(2, 0),
			angle
		);
		// Updating force-OX for atom-0
		this->particle_array[outer_mol_i].upd_force(outer_atom_i, 0, force_buffer);
		// Updating force-OX for atom-2
		this->particle_array[outer_mol_i].upd_force(inner_atom_i, 0, -force_buffer);

		// Atom-0 Force-OY
		force_buffer = this->angle_force_left_atom
		(
			this->particle_array[outer_mol_i],
			(*Water_molecule::equilibrium_angle),
			(*Water_molecule::k_coef_angle_poten),
			this->particle_array[outer_mol_i].get_coordinate(0, 1),
			this->particle_array[outer_mol_i].get_coordinate(1, 1),
			this->particle_array[outer_mol_i].get_coordinate(2, 1),
			angle
		);
		// Updating force-OY for atom-0
		this->particle_array[outer_mol_i].upd_force(outer_atom_i, 1, force_buffer);
		// Updating force-OY for atom-2
		this->particle_array[outer_mol_i].upd_force(inner_atom_i, 1, -force_buffer);

		// Atom-0 Force-OZ
		force_buffer = this->angle_force_left_atom
		(
			this->particle_array[outer_mol_i],
			(*Water_molecule::equilibrium_angle),
			(*Water_molecule::k_coef_angle_poten),
			this->particle_array[outer_mol_i].get_coordinate(0, 2),
			this->particle_array[outer_mol_i].get_coordinate(1, 2),
			this->particle_array[outer_mol_i].get_coordinate(2, 2),
			angle
		);
		// Updating force-OZ for atom-0
		this->particle_array[outer_mol_i].upd_force(outer_atom_i, 2, force_buffer);
		// Updating force-OZ for atom-2
		this->particle_array[outer_mol_i].upd_force(outer_atom_i, 2, -force_buffer);
		// Potential
		potential_energy = potential_energy + (1.0*(this->angle_potential(angle,
(*Water_molecule::equilibrium_angle), (*Water_molecule::k_coef_angle_poten))));
		// -------------------------------------------------------------------

		// Bond Force
		for (outer_atom_i = 0, inner_atom_i = 1; outer_atom_i < 2;
			++outer_atom_i, ++inner_atom_i)
		{
			// Distance calculating
```

```
                        dx = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 0)
-
                                this->particle_array[outer_mol_i].get_coordinate(inner_atom_i,
0);
                        dx = this->distance_correction(dx, 0);
                        dy = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 1)
-
                                this->particle_array[outer_mol_i].get_coordinate(inner_atom_i,
1);
                        dy = this->distance_correction(dy, 1);
                        dz = this->particle_array[outer_mol_i].get_coordinate(outer_atom_i, 2)
-
                                this->particle_array[outer_mol_i].get_coordinate(inner_atom_i,
2);
                        dz = this->distance_correction(dz, 2);
                        distance_buffer = powl(dx*dx + dy * dy + dz * dz, 0.5);

                        // Skip iteration
                        if (distance_buffer > r_cut) { continue; }

                        // Bond link: FORCE
                        force_buffer = this->bond_force(distance_buffer,
(*Water_molecule::equilibrium_distance_bond_poten), (*Water_molecule::k_coef_bond_poten));

                        // Bond link: Potential
                        potential_energy = potential_energy + 1.0*(this-
>bond_potential(distance_buffer, (*Water_molecule::equilibrium_distance_bond_poten),
(*Water_molecule::k_coef_bond_poten)) /
                                (*this->array_size));

                        // Updating Force values
                        // OX
                        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 0, dx /
distance_buffer * force_buffer);
                        // OX -- pair
                        this->particle_array[outer_mol_i].upd_force(inner_atom_i, 0, -dx /
distance_buffer * force_buffer);
                        // OY
                        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 1, dy /
distance_buffer * force_buffer);
                        // OY -- pair
                        this->particle_array[outer_mol_i].upd_force(inner_atom_i, 1, -dy /
distance_buffer * force_buffer);
                        // OZ
                        this->particle_array[outer_mol_i].upd_force(outer_atom_i, 2, dz /
distance_buffer * force_buffer);
                        // OZ -- pair
                        this->particle_array[outer_mol_i].upd_force(inner_atom_i, 2, -dz /
distance_buffer * force_buffer);
                }

                // Inner molecules cycle
                for (inner_mol_i = outer_mol_i + 1; inner_mol_i < (*this->array_size);
                        ++inner_mol_i)
                {
                        // Outer atom cycle
                        for (outer_atom_i = 0;
                                outer_atom_i < (*Water_molecule::atom_count); ++outer_atom_i)
                        {
                                // Checking first atom type
                                atom_1_type = this-
>particle_array[outer_mol_i].get_type(outer_atom_i);

                                // Checking first atom charge
```

```
                            charge_1_buffer = (atom_1_type == (*Water_molecule::type_O)) ?
                                   (*Water_molecule::q_charge_O) :
                                   (*Water_molecule::q_charge_H);

                            // Inner atom cycle
                            for (inner_atom_i = 0;
                                   inner_atom_i < (*Water_molecule::atom_count);
                                   ++inner_atom_i)
                            {
                                   // Rewritting
                                   force_buffer = 0.0;

                                   // Checking second atom type
                                   atom_2_type = this-
>particle_array[inner_mol_i].get_type(inner_atom_i);

                                   // Checking second atom charge
                                   charge_2_buffer = (atom_2_type ==
(*Water_molecule::type_O)) ?
                                          (*Water_molecule::q_charge_O) :
                                          (*Water_molecule::q_charge_H);

                                   // Distance calculating
                                   dx = this-
>particle_array[outer_mol_i].get_coordinate(outer_atom_i, 0) -
                                          this-
>particle_array[inner_mol_i].get_coordinate(inner_atom_i, 0);
                                   dx = this->distance_correction(dx, 0);
                                   dy = this-
>particle_array[outer_mol_i].get_coordinate(outer_atom_i, 1) -
                                          this-
>particle_array[inner_mol_i].get_coordinate(inner_atom_i, 1);
                                   dy = this->distance_correction(dy, 1);
                                   dz = this-
>particle_array[outer_mol_i].get_coordinate(outer_atom_i, 2) -
                                          this-
>particle_array[inner_mol_i].get_coordinate(inner_atom_i, 2);
                                   dz = this->distance_correction(dz, 2);
                                   distance_buffer = powl(dx*dx + dy * dy + dz * dz, 0.5);

                                   // Skip iteration
                                   if (distance_buffer > r_cut) { continue; }

                                   // /*                       // Lennard-Jones
                                   if ((atom_1_type == (*Water_molecule::type_O)) &&
                                          (atom_2_type == (*Water_molecule::type_O)))
                                   {
                                          force_buffer = this->lennard_jones_force
                                          (
                                                 distance_buffer,
(*Water_molecule::energy_LJ_OO),

       (*Water_molecule::equilibrium_distance_LJ_OO)
                                          );
                                          potential_energy = potential_energy + 1.0*((this-
>lennard_jones_potential
                                          (
                                                 distance_buffer,
(*Water_molecule::energy_LJ_OO),

       (*Water_molecule::equilibrium_distance_LJ_OO)
                                          )) / (*this->array_size));
                                   }
                                   else
```

```
                                                        if ((atom_1_type == (*Water_molecule::type_H)) &&
                                                                (atom_2_type == (*Water_molecule::type_H)))
                                                        {
                                                                force_buffer = this->lennard_jones_force
                                                                (
                                                                        distance_buffer,
(*Water_molecule::energy_LJ_HH),

        (*Water_molecule::equilibrium_distance_LJ_HH)
                                                                );
                                                                potential_energy = potential_energy +
1.0*((this->lennard_jones_potential
                                                                (
                                                                        distance_buffer,
(*Water_molecule::energy_LJ_HH),

        (*Water_molecule::equilibrium_distance_LJ_HH)
                                                                )) / (*this->array_size));
                                                        }
                                                        else
                                                        {
                                                                force_buffer = this->lennard_jones_force
                                                                (
                                                                        distance_buffer,
(*Water_molecule::energy_LJ_OH),

        (*Water_molecule::equilibrium_distance_LJ_OH)
                                                                );
                                                                potential_energy = potential_energy +
1.0*((this->lennard_jones_potential
                                                                (
                                                                        distance_buffer,
(*Water_molecule::energy_LJ_OH),

        (*Water_molecule::equilibrium_distance_LJ_OH)
                                                                )) / (*this->array_size));
                                                        }
                                                // Coulomb
                                                // Force
                                                force_buffer = force_buffer + this-
>coulomb_force(distance_buffer, (*Water_molecule::k_coef_coulomb_potential),
charge_1_buffer, charge_2_buffer);
                                                // Potential
                                                potential_energy = potential_energy +
                                                        (1.0*(this->coulomb_potential(distance_buffer,
(*Water_molecule::k_coef_coulomb_potential), charge_1_buffer, charge_2_buffer)) /
                                                        (*this->array_size));

                                                // Updating Force values
                                                // OX
                                                this->particle_array[outer_mol_i].upd_force(outer_atom_i,
0, dx / distance_buffer * force_buffer);
                                                // OX - for pair
                                                this->particle_array[inner_mol_i].upd_force(inner_atom_i,
0, -dx / distance_buffer * force_buffer);
                                                // OY
                                                this->particle_array[outer_mol_i].upd_force(outer_atom_i,
1, dy / distance_buffer * force_buffer);
                                                // OY - for pair
                                                this->particle_array[inner_mol_i].upd_force(inner_atom_i,
1, -dy / distance_buffer * force_buffer);
                                                // OZ
                                                this->particle_array[outer_mol_i].upd_force(outer_atom_i,
2, dz / distance_buffer * force_buffer);
```

```
                                // OZ - for pair
                                this->particle_array[inner_mol_i].upd_force(inner_atom_i,
2, -dz / distance_buffer * force_buffer);
                                // */
                        }
                }
        }
        // Calculating 'Kinetic Energy' and 'Accelerations'
        for (outer_atom_i = 0; outer_atom_i < (*Water_molecule::atom_count);
                ++outer_atom_i)
        {
                // Getting atom type
                atom_1_type = this->particle_array[outer_mol_i].get_type(outer_atom_i);

                // Calculating velocity components
                vx = this->particle_array[outer_mol_i].get_velocity(outer_atom_i, 0);
                vy = this->particle_array[outer_mol_i].get_velocity(outer_atom_i, 1);
                vz = this->particle_array[outer_mol_i].get_velocity(outer_atom_i, 2);

                if (atom_1_type == (*Water_molecule::type_O))
                {
                        kinetic_energy = kinetic_energy + (
                                (this->get_kinetic_energy(vx*vx + vy * vy + vz * vz,
(*Water_molecule::mass_O))) /
                                (*this->array_size));

                        // OX
                        this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
0, this->particle_array[outer_mol_i].get_force(outer_atom_i, 0) /
(*Water_molecule::mass_O));
                        // OY
                        this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
1, this->particle_array[outer_mol_i].get_force(outer_atom_i, 1) /
(*Water_molecule::mass_O));
                        // OZ
                        this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
2, this->particle_array[outer_mol_i].get_force(outer_atom_i, 2) /
(*Water_molecule::mass_O));
                }
                else if (atom_1_type == (*Water_molecule::type_H))
                {
                        kinetic_energy = kinetic_energy +
                                ((this->get_kinetic_energy(vx*vx + vy * vy + vz * vz,
(*Water_molecule::mass_H))) /
                                (*this->array_size));

                        // OX
                        this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
0, this->particle_array[outer_mol_i].get_force(outer_atom_i, 0) /
(*Water_molecule::mass_H));
                        // OY
                        this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
1, this->particle_array[outer_mol_i].get_force(outer_atom_i, 1) /
(*Water_molecule::mass_H));
                        // OZ
                        this->particle_array[outer_mol_i].set_acceleration(outer_atom_i,
2, this->particle_array[outer_mol_i].get_force(outer_atom_i, 2) /
(*Water_molecule::mass_H));
                }
        }
    }

    return (kinetic_energy + potential_energy);
}
```

```
void MD_experiment::run_experiment_brute_force_with_r(long double & system_energy, long
double r_cut)
{
	// Local variables defining
	// -----------------------------------------------------------------------
	Water_molecule * mol_ptr = this->particle_array,
		*mol_end_ptr = this->particle_array + (*this->array_size);
	int atom_i = 0, dim_i = 0;
	// -----------------------------------------------------------------------

	// STEP #1: Calculating new coordinate values and
	//          Velocity values partial update
	// -----------------------------------------------------------------------
	// Molecules cycle
	for (; mol_ptr < mol_end_ptr; ++mol_ptr)
	{
		// Atoms cycle
		for (atom_i = 0; atom_i < (*Water_molecule::atom_count); ++atom_i)
		{
			// Dimensions cycle
			for (dim_i = 0; dim_i < (*MD_experiment::dimension_count); ++dim_i)
			{
				// Coordinate update
				mol_ptr->upd_coordinate
				(
					atom_i, dim_i,
					(*this->time_step)*(mol_ptr->get_velocity(atom_i, dim_i))
+
					0.5 * powl(*this->time_step, 2) *
					(mol_ptr->get_acceleration(atom_i, dim_i))
				);

				// Coordinate correction
				this->coordinate_correction(mol_ptr, atom_i, dim_i);

				// Partial velocity update
				mol_ptr->upd_velocity
				(
					atom_i, dim_i,
					0.5 * (*this->time_step) *
					(mol_ptr->get_acceleration(atom_i, dim_i))
				);
			}
		}
	}
	// -----------------------------------------------------------------------

	// STEP #2: Accelerate values update
	// -----------------------------------------------------------------------
	system_energy = this->upd_particle_accelerations_brute_force_with_r
	(
		r_cut
	);
	// -----------------------------------------------------------------------

	// STEP #3: Velocity values final partial update
	// -----------------------------------------------------------------------
	for (mol_ptr = this->particle_array; mol_ptr < mol_end_ptr; ++mol_ptr)
	{
		// Atoms cycle
		for (atom_i = 0; atom_i < (*Water_molecule::atom_count); ++atom_i)
		{
```

```
                        // Dimensions cycle
                        for (dim_i = 0; dim_i < (*MD_experiment::dimension_count); ++dim_i)
                        {
                                // Partial velocity update
                                mol_ptr->upd_velocity
                                (
                                        atom_i, dim_i,
                                        0.5 * (*this->time_step) *
                                        (mol_ptr->get_acceleration(atom_i, dim_i))
                                );
                        }
                }
        }
        // -----------------------------------------------------------------------
}
```

## ДОДАТОК Ф

Лістинг реалізації алгоритму функції ініціалізації мовою програмування С++.

Файл *MD_experiment_Brute_Force_iteration_with_R-cut.cpp*

```
#include "MD_experiment_init_1.cuh"


void initialise_md_experiment_1
(
	MD_experiment * moldy_experiment,
	long double & system_energy
)
{
	// STEP #0: Preparing static constants
	// -----------------------------------------------------------------------
	moldy_experiment->setup_constants();
	// -----------------------------------------------------------------------

	// STEP #1: Setting modelling area sizes
	// -----------------------------------------------------------------------
	{
		// Local variables defining
		// -----------------------------------------------------------------------
		long double width = 0.0, length = 0.0, height = 0.0;
		// -----------------------------------------------------------------------

		// Getting modelling area parameters from the User
		while (true)
		{
			// User invitation
			// /*
			std::cout << "Please, type the values of modelling area shape."
				<< "\nMiminum values are"
				<< "\n\t- for width:  "
				<< (*MD_experiment::modelling_subarea_width)
				<< "\n\t- for length: "
				<< (*MD_experiment::modelling_subarea_length)
				<< "\n\t- for height: "
				<< (*MD_experiment::modelling_subarea_height);
			// */

			// User input process
			std::cout << "\n\nEnter Width: ";
			std::cin >> width;
			std::cout << "\nEnter Length: ";
			std::cin >> length;
			std::cout << "\nEnter Height: ";
			std::cin >> height;

			// Validation
			if ((width >= (*MD_experiment::modelling_subarea_width)) &&
				(length >= (*MD_experiment::modelling_subarea_length)) &&
				(height >= (*MD_experiment::modelling_subarea_height)))
			{
				moldy_experiment->set_modelling_area_sizes
				(
					width, length, height
				);
				break;
```

```
                }
                std::cout << "\nOne of the values is invalid!";
                std::cout << "\nPlease, try again.\n\n";
            }
        }
        // ----------------------------------------------------------------

        // STEP #2: Setting particle count
        // ----------------------------------------------------------------
        // Calculating maximum particle count
        moldy_experiment->calculate_max_particle_count();

        // Getting particle count value from the User
        {
            size_t user_input = 0;

            while (true)
            {
                // User invitation
                // /*
                std::cout << "\nPlease, type the value of particle count." << '\n';
                std::cout << "\nMax value is "
                    << (*moldy_experiment->max_particle_count) << ". "
                    << "Your value: ";
                // */
                // Getting user value
                std::cin >> user_input;

                if (user_input <= (*moldy_experiment->max_particle_count))
                {
                    moldy_experiment->set_particle_count(user_input);
                    break;
                }
                else
                {
                    std::cout << "\nThe value " << user_input
                        << " is too big! Try again..." << '\n';
                }
            }
        }
        // ----------------------------------------------------------------

        // STEP #3: Setting particles coordinates
        // ----------------------------------------------------------------
        moldy_experiment->set_particle_coordinates();
        // ----------------------------------------------------------------

        // STEP #4: Defining particle velocities
        // ----------------------------------------------------------------
        moldy_experiment->set_particle_velocities();
        // ----------------------------------------------------------------

        // STEP #5: Defining particle accelerations. Brute Force
        // ----------------------------------------------------------------
        system_energy = moldy_experiment->upd_particle_accelerations_brute_force();
        // ----------------------------------------------------------------

        // Showing params
        // moldy_experiment->show_experiment_parameters();
}
```

# ДОДАТОК Х

Таблиця результатів комп’ютерного експерименту з параметрами: 100 моделей молекул та розмірами контейнеру $100 \times 100 \times 100$

Таблиця Х.1 – Результати комп’ютерного експерименту розрахунку середнього значення енергії за кількістю частинок, що усереднено за кількістю кроків за часом

| Радіус відсікання | Контрольний параметр |
|---|---|
| — | -0.0356448 |
| 0.5 | -0.000395166 |
| 1.0 | -0.000395166 |
| 1.5 | -0.000395166 |
| 2.0 | -0.000395166 |
| 2.5 | -0.000395166 |
| 3.0 | -0.000395166 |
| 3.5 | -0.000395166 |
| 4.0 | -0.000395166 |
| 4.5 | -0.000395166 |
| 5.0 | -0.000395166 |
| 5.5 | -0.000395166 |
| 6.0 | -0.000395166 |
| 6.5 | -0.000395166 |
| 7.0 | -0.000395166 |
| 7.5 | -0.000395166 |
| 8.0 | -0.000395166 |
| 8.5 | -0.000395166 |
| 9.0 | 46.58 |
| 9.5 | -0.0193693 |
| 10.0 | -0.0193693 |
| 10.5 | 0.982245 |
| 11.0 | -0.0197128 |
| 11.5 | -0.0197128 |
| 12.0 | -0.0197128 |
| 12.5 | -38.006 |
| 13.0 | -7.24185 |
| 13.5 | -3.26333 |
| 14.0 | -0.687933 |
| 14.5 | -0.0254916 |
| 15.0 | -0.0254916 |
| 15.5 | -0.0254916 |

Продовження таблиці Х.1

| | |
|---|---|
| 16.0 | -0.0254916 |
| 16.5 | -0.0254916 |
| 17.0 | -0.0254916 |
| 17.5 | -0.0254916 |
| 18.0 | 20.6051 |
| 18.5 | -0.0272991 |
| 19.0 | 1.08993 |
| 19.5 | 1.11549 |
| 20.0 | -50.8193 |
| 20.5 | -4.61242 |
| 21.0 | 1.26972 |
| 21.5 | 3.6362 |
| 22.0 | 0.129749 |
| 22.5 | -0.0301018 |
| 23.0 | -0.0301018 |
| 23.5 | -0.0301018 |
| 24.0 | -0.0301018 |
| 24.5 | -0.0301018 |
| 25.0 | 3.00666 |
| 25.5 | 16.6115 |
| 26.0 | 0.606434 |
| 26.5 | 4.79822 |
| 27.0 | 6.05522 |
| 27.5 | 0.0972355 |
| 28.0 | 6.43231 |
| 28.5 | 19.9878 |
| 29.0 | 0.779153 |
| 29.5 | -3.27044 |
| 30.0 | -0.733887 |
| 30.5 | -0.109387 |
| 31.0 | -0.0323187 |
| 31.5 | -0.0323187 |
| 32.0 | 4.11681 |
| 32.5 | 16.138 |
| 33.0 | -3.21403 |
| 33.5 | -7.02686 |
| 34.0 | 0.106306 |
| 34.5 | 0.172333 |
| 35.0 | -0.0329413 |
| 35.5 | 2.06122 |
| 36.0 | 3.85373 |

Кінець таблиці Х.1

| | |
|---|---|
| 36.5 | -0.0330989 |
| 37.0 | 15.0765 |
| 37.5 | 2.35085 |
| 38.0 | -7.47845 |
| 38.5 | 2.98692 |
| 39.0 | -5.02012 |
| 39.5 | -0.808832 |
| 40.0 | -14.5911 |
| 40.5 | -3.28216 |
| 41.0 | 1.63619 |
| 41.5 | 4.75523 |
| 42.0 | 0.785603 |
| 42.5 | -0.0342765 |
| 43.0 | -0.0342765 |
| 43.5 | -0.0342765 |
| 44.0 | -0.0342765 |
| 44.5 | 3.63244 |
| 45.0 | 9.20737 |
| 45.5 | -7.35685 |
| 46.0 | 1.25447 |
| 46.5 | 2.27101 |
| 47.0 | -4.86071 |
| 47.5 | -0.853807 |
| 48.0 | -4.05494 |
| 48.5 | 4.07064 |
| 49.0 | 0.613717 |
| 49.5 | -7.3310 |
| 50.0 | -1.32776 |

# ДОДАТОК Ц

Таблиця результатів комп'ютерного експерименту з параметрами: 200 моделей молекул та розмірами контейнеру $100 \times 100 \times 100$

Таблиця Ц.1 – Результати комп'ютерного експерименту розрахунку середнього значення енергії за кількістю частинок, що усереднено за кількістю кроків за часом

| Радіус відсікання | Контрольний параметр |
|---:|---:|
| — | -0.0320192 |
| 0.5 | -0.00035528 |
| 1.0 | -0.00035528 |
| 1.5 | -0.00035528 |
| 2.0 | -0.00035528 |
| 2.5 | -0.00035528 |
| 3.0 | -0.00035528 |
| 3.5 | -0.00035528 |
| 4.0 | -0.00035528 |
| 4.5 | -0.00035528 |
| 5.0 | -0.00035528 |
| 5.5 | -0.00035528 |
| 6.0 | -0.00035528 |
| 6.5 | -0.00035528 |
| 7.0 | -0.00035528 |
| 7.5 | -0.00035528 |
| 8.0 | -0.00035528 |
| 8.5 | -0.00035528 |
| 9.0 | 46.739 |
| 9.5 | -0.0108361 |
| 10.0 | -0.0108361 |
| 10.5 | 1.61007 |
| 11.0 | -0.0117986 |
| 11.5 | -0.0117986 |
| 12.0 | -0.0117986 |
| 12.5 | -51.7596 |
| 13.0 | -7.21284 |
| 13.5 | -4.30574 |
| 14.0 | -1.7174 |
| 14.5 | -0.215273 |
| 15.0 | -0.0539041 |
| 15.5 | -30.56 |

Продовження таблиці Ц.1

| | |
|---|---|
| 16.0 | 0.866389 |
| 16.5 | 5.97013 |
| 17.0 | -0.049526 |
| 17.5 | -0.0486466 |
| 18.0 | 20.2447 |
| 18.5 | -0.0171677 |
| 19.0 | 1.10103 |
| 19.5 | 1.72709 |
| 20.0 | -66.0865 |
| 20.5 | -5.3685 |
| 21.0 | 2.43635 |
| 21.5 | 6.65494 |
| 22.0 | 39.1406 |
| 22.5 | -3.8307 |
| 23.0 | -4.66721 |
| 23.5 | -1.42214 |
| 24.0 | -0.11956 |
| 24.5 | -0.0221354 |
| 25.0 | 3.01601 |
| 25.5 | 16.7303 |
| 26.0 | 1.02453 |
| 26.5 | 10.0439 |
| 27.0 | 22.8943 |
| 27.5 | 1.52215 |
| 28.0 | 14.2292 |
| 28.5 | 25.3725 |
| 29.0 | 1.46545 |
| 29.5 | -16.3098 |
| 30.0 | -2.51022 |
| 30.5 | -0.52604 |
| 31.0 | -6.59209 |
| 31.5 | 0.534046 |
| 32.0 | 4.12496 |
| 32.5 | 16.1458 |
| 33.0 | -4.31698 |
| 33.5 | -30.2209 |
| 34.0 | 0.878354 |
| 34.5 | -1.46377 |
| 35.0 | -1.01568 |
| 35.5 | 3.35784 |
| 36.0 | 3.86023 |

Кінець таблиці Ц.1

| | |
|---|---|
| 36.5 | -0.027196 |
| 37.0 | 21.7819 |
| 37.5 | 2.53835 |
| 38.0 | -21.7034 |
| 38.5 | 5.23622 |
| 39.0 | -13.9325 |
| 39.5 | 1.87371 |
| 40.0 | -20.1461 |
| 40.5 | -4.70479 |
| 41.0 | -11.4546 |
| 41.5 | 2.03564 |
| 42.0 | 2.04176 |
| 42.5 | 4.40333 |
| 43.0 | 1.36428 |
| 43.5 | -0.0294622 |
| 44.0 | -0.0294622 |
| 44.5 | 3.63793 |
| 45.0 | 9.21234 |
| 45.5 | -16.2197 |
| 46.0 | -1.56465 |
| 46.5 | -7.10486 |
| 47.0 | -2.49948 |
| 47.5 | -2.47464 |
| 48.0 | 0.337649 |
| 48.5 | 4.5288 |
| 49.0 | -7.95544 |
| 49.5 | -9.40893 |
| 50.0 | -0.67984950 |

## ДОДАТОК Ш

Графіки результатів моделювання системи частинок як пружних куль з використанням графічного прискорювача (технологія CUDA)

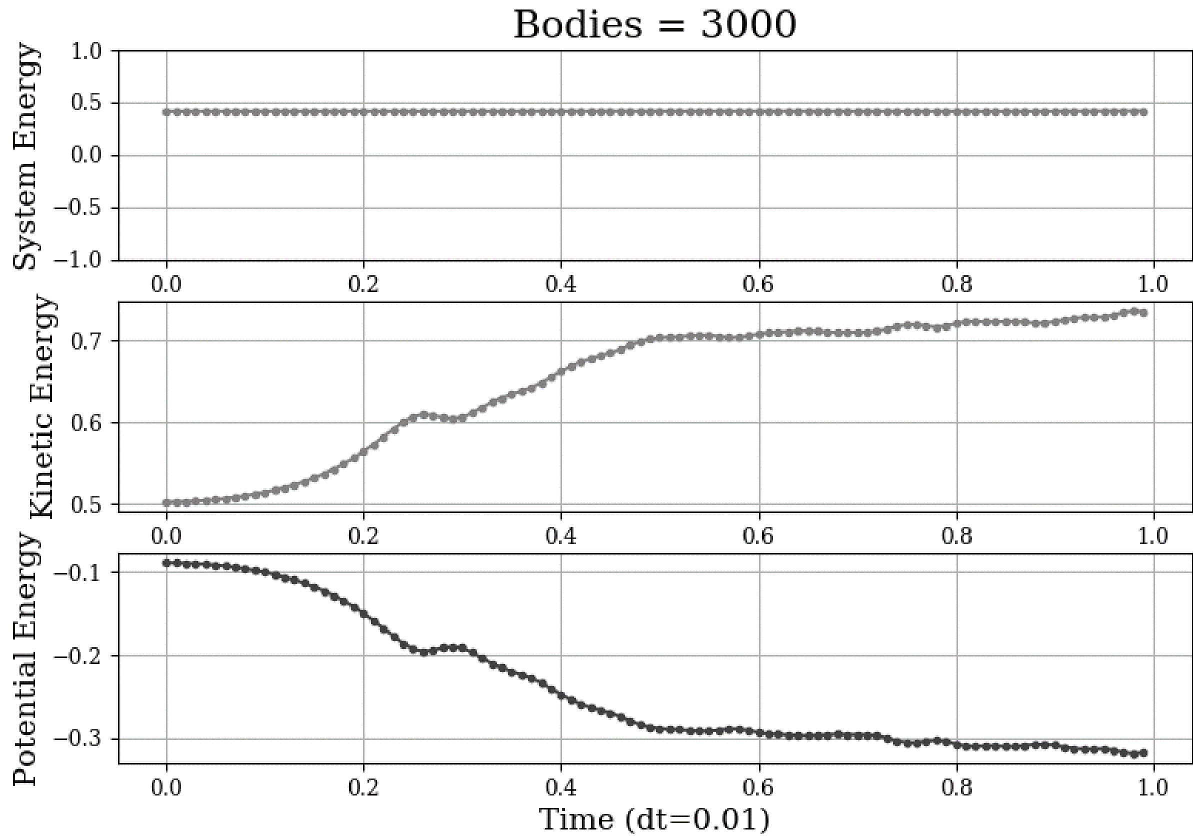



*Рисунок Ш.1 – Графік залежності енергії від часу моделювання для системи з 3000 гнучких куль у межах модельної області* 19 м.о. × 18 м.о. × 18 м.о., *де м.о. – модельні безрозмірні одиниці. System energy – повно енергія системи; kinetic energy – кінетична енергія системи, potential energy – потенціальна енергія системи*

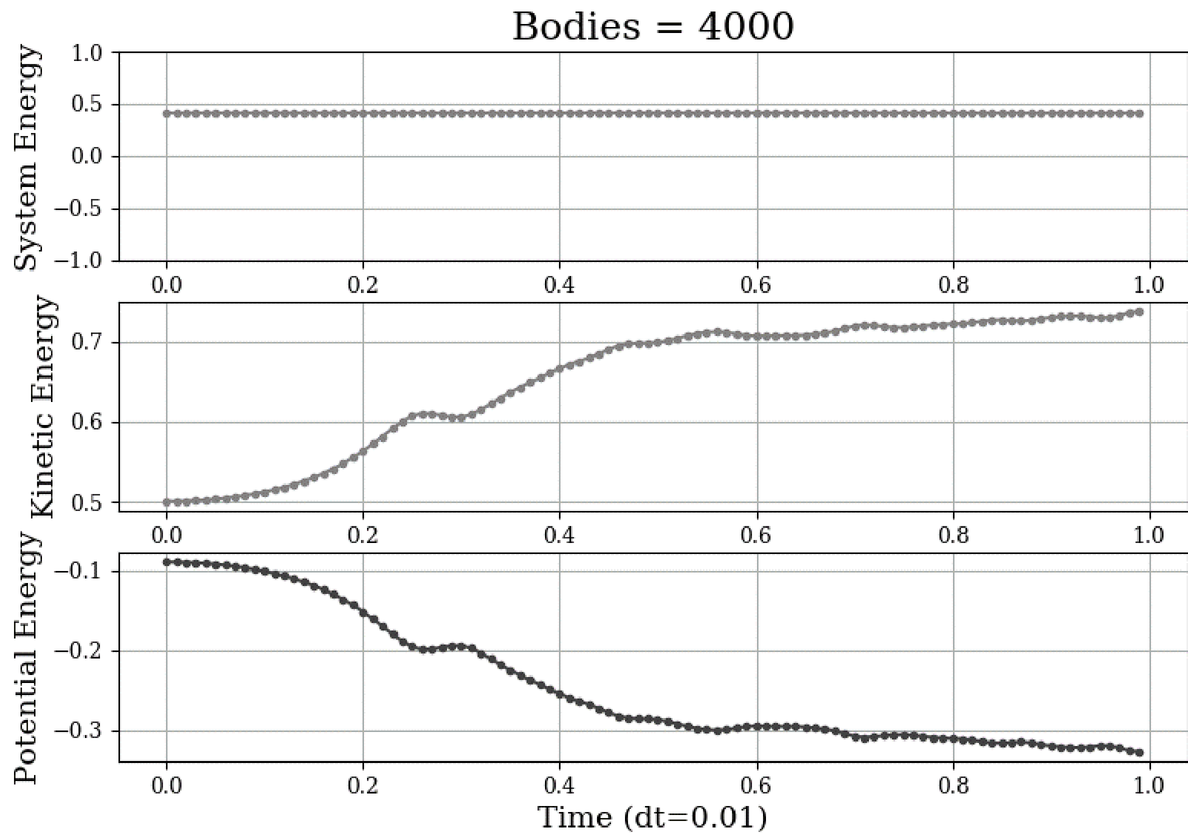



*Рисунок Ш.2 – Графік залежності енергії від часу моделювання для системи з 4000 гнучких куль у межах модельної області* 19 м.о. × 18 м.о. × 18 м.о., *де м.о. – модельні безрозмірні одиниці. System energy – повно енергія системи; kinetic energy – кінетична енергія системи, potential energy – потенціальна енергія системи*