МІНІСТЕРСТВО ОСВІТИ Й НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра прикладної математики та моделювання складних систем

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
спеціальності «Прикладна математика»

з теми: «Дослідження можливостей підвищення ефективності програмних алгоритмів для моделювання систем взаємодіючих частинок»

Завідуючий випускаючою кафедрою	д-р. фіз-мат. наук проф. О.В. Лисенко
Керівник роботи	канд. фіз.-мат наук, доц. І.О. Князь
Виконавець	студент факультету ЕлІТ гр. ПМ-51 О.І. Арнаутов

Затверджено на засіданні кафедри «__» __________ 2019 р.,
протокол №

Суми – 2019

РЕФЕРАТ

Кваліфікаційна робота бакалавра складається з 118 сторінок, 74 рисунки, 7 джерел.

Об’єкт дослідження – система взаємодіючих частинок.

Мета роботи – підвищення ефективності розрахунку характеристик системи, що складається з багатьох частинок за рахунок введення радіусу обмеження – значення відстані від деякої частинки системи, в межах якого інші частинки системи будуть враховуватися для обчислення нового прискорення для частинки, від котрої відраховується радіус. А також перевірка гіпотези існування залежності між радіусом обмеження та кількістю частинок, з яких складається система в умовах однакового габаритного контейнеру, у межах якого моделюється еволюція системи.

Рух системи частинок описується за допомогою методу молекулярної динаміки – розв’язуються рівняння руху для кожної частинки. Взаємодія частинок описується потенціалом Леннард-Джонса.

Будуються програми для моделювання руху, для центрального процесору та системи з центрального та графічного процесорів. Розрахунковий алгоритм для реалізації з використанням графічного процесору модифікується за допомогою введення поняття радіусу обмеження в систему частинок. Оцінено ефективність побудованих програмних реалізацій.

Досліджено гіпотезу про існування залежності між радіусом обмеження та кількістю частинок у системі, рух якої моделюється.

МЕТОД МОЛЕКУЛЯРНОЇ ДИНАМІКИ, СИСТЕМА ЧАСТИНОК, ПОТЕНЦІАЛ ЛЕННАРД-ДЖОНСА, РАДІУС ОБМЕЖЕННЯ, ЦЕНТРАЛЬНИЙ ПРОЦЕСОР (CPU), ГРАФІЧНИЙ ПРОЦЕСОР (GPU), CUDA, PYTHON.

ЗМІСТ

РЕФЕРАТ 2

ЗМІСТ 3

ВСТУП 5

ТЕОРЕТИЧНА ЧАСТИНА 8

1 Метод молекулярної динаміки 8

2 Особливості будови CPU та GPU, відмінності та переваги кожної архітектури 13

3 Існуючі методи прискорення розрахунків характеристик систем на основі методу молекулярної динаміки 14

ПРАКТИЧНА ЧАСТИНА 15

4 Математична модель 15

5 Програмна реалізація 19

5.1 Загальна схема програми 19

5.2 Реалізація функції ініціалізації системи 21

5.3 Реалізація функцій розрахунку характеристик системи 23

5.4 Реалізація функції, що будує таблиці, на основі яких обирається радіус обмеження 25

5.5 Реалізація функції, що будує графіки порівняння швидкодії реалізацій CPU, пари CPU, GPU без використання радіусу та пари CPU, GPU з використанням радіусу обмеження 53

ВИСНОВКИ 58

ПЕРЕЛІК ПОСИЛАНЬ 59

ДОДАТОК А 60

Лістинг програми побудови графіків залежності сили та потенціалу від відстані між частинками 60

ДОДАТОК Б 66

Лістинг програми моделювання системи частинок 66

ДОДАТОК В 67

Лістинг основного модуля *main_window* програми моделювання системи частинок 67

ДОДАТОК Г 105
Лістинг модуля *calc_coordinates_cpu_1ker* для розрахунку координат для CPU. 105

ДОДАТОК Ґ 106
Лістинг модуля *calc_speeds_cpu_1ker* для розрахунку швидкостей на CPU 106

ДОДАТОК Д 107
Лістинг модуля *calc_accelerate_cpu_1ker* для розрахунку прискорень на CPU ... 107

ДОДАТОК Е 108
Лістинг модуля *calc_force* для розрахунку сили на CPU 108

ДОДАТОК Є 109
Лістинг модуля *calc_distance* для розрахунку відстаней на CPU 109

ДОДАТОК Ж 110
Лістинг модуля *calc_system_energy* для розрахунку енергій на CPU 110

ДОДАТОК З 111
Лістинг модуля *GPU_calc_coords* для розрахунку координат на GPU 111

ДОДАТОК И 112
Лістинг модуля *GPU_calc_speeds* для розрахунку швидкостей на GPU 112

ДОДАТОК І 113
Лістинг модуля *GPU_calc_accelerates* для розрахунку прискорень на GPU 113

ДОДАТОК Ї 114
Лістинг модуля *GPU_calc_kinetic_energy* для розрахунку кінетичної енергії на GPU 114

ДОДАТОК Й 115
Лістинг модуля *GPU_calc_accelerates_with_rcut.py* для розрахунку прискорень на GPU з використанням радіусу обмеження 115

ДОДАТОК К 116
Лістинг програми для побудови графіків залежності усередненої енергії від радіусу обмеження 116

ДОДАТОК Л 118
Лістинг функції, що будує таблиці для побудови графіків залежності усередненої енергії від радіусу обмеження 118

ВСТУП

Мета роботи – є підвищення ефективності розрахунку характеристик системи, що складається з багатьох частинок за рахунок введення радіусу обмеження – значення відстані від деякої частинки системи, в межах якого інші частинки системи будуть враховуватися для обчислення нового прискорення для частинки, від котрої відраховується радіус. А також перевіряється гіпотеза існування залежності між радіусом обмеження та кількістю частинок, з яких складається система в умовах однакового габаритного контейнеру, у межах якого моделюється еволюція системи.

Завданнями кваліфікаційної роботи є:

- розробка комп'ютерної програми для моделювання динаміки системи взаємодіючих частинок:
    - реалізація програми на базі центрального процесора;
    - реалізація на базі гібридної системи центрального та графічного процесорів.
- модифікація алгоритму з метою підвищення ефективності розрахунку станів системи взаємодіючих частинок за допомогою введення радіусу обмеження;
- перевірка гіпотези про залежність між радіусом та розміром системи.

Актуальність. У даній роботі порівнюється ефективність реалізацій програм для моделювання системи частинок на базі центрального процесору (Central Processing Unit, CPU) та з використанням об'єднаної системи CPU та графічного процесору (Graphic Processing Unit, GPU), а також перевіряється гіпотеза про існування залежності між радіусом обмеження та кількістю частинок у системі, що досліджується.

Об'єкт дослідження: система взаємодіючих частинок.

Предмет дослідження: методи розрахунку станів системи взаємодіючих частинок.

Метод молекулярної динаміки (далі за текстом МД) використовується як зручний інструмент для дослідження властивостей об'єктів на рівні мікросвіту в масштабах мікро-, нано- та пікометрів. Він дозволяє «сповільнити» процеси, що відбуваються миттєво: відслідкувати траєкторію окремих молекул та атомів; явно відобразити взаємодію різних сполучень та сумішей, дослідити наслідки впливу сил на структурну будову. А також базуючись на цьому методі є можливість побудови нових структур, що будуть мати, властивості та параметри, що необхідні для рішення інших задач.

Особливий інтерес полягає у можливості вивчати властивості та вимірювати характеристики не тільки за допомогою спостереження, а й безпосереднього впливу на об'єкт дослідження та вивчення його відповідних реакцій та наслідків взаємодії.

Метод моделювання об’єктів як великої сукупності взаємодіючих частинок є поширеним засобом опису деформацій тій. Результати досліджень, у яких використовувався метод МД, дозволяють говорити про тісний взаємозв’язок внутрішньої будови та процесу деформації об’єкта. Даний приклад підтверджує користь методу для моделювання об’єктів, що розглядається у роботі [1].

Основною ціллю МД є представлення деякого досліджуваного об’єкту у вигляді єдиної системи взаємодіючих частин (наприклад: матеріальних точок, молекул, атомів, певних структурних елементів), що описуються за допомогою класичної механіки. Взаємодія між самими частинками виражається через співвідношення, що називаються потенціалами взаємодії. Досліджені потенціали взаємодії частинок дозволяють моделювати поведінку об’єктів з високою точністю [1].

Усі ці можливості стали доступними завдяки розвитку обчислювальної техніки. Задачі, що не мали до цього аналітичного розв’язку з доповненням у вигляді комп’ютерної моделі отримали наближені рішення, що базуються на чисельних методах. Комп’ютерне моделювання є об’єднуючим елементом, що пов’язує теоретичні знання та реальний експеримент. Користуючись теоретичними надбаннями є можливість побудови комп’ютерних моделей, складність та точність котрих обмежується лише технічними потужностями, а також дозволяє уникнути складнощів, що пов’язанні з проведенням реального експерименту [1].

Основною складністю використання методу МД є обчислення великої кількості рівнянь, що представляють взаємозв’язки між частинами досліджуваного об’єкту. Саме через це швидкість розрахунків є важливим питанням при побудові комп’ютерної моделі системи багатьох взаємодіючих частинок.

Пришвидшити виконання обчислень можливо двома шляхами:

- апаратною складовою (оновлення складових на більш продуктивні або використання додаткової потужності);
- алгоритмічним способом (використання сучасних чисельних алгоритмів або модифікація наявних).

До прискорення за допомогою заміни або оновлення апаратної бази можна віднести використання спеціалізованих розрахункових кластерів. А також використання дискретних графічних процесорів як паралельних обчислювальних платформ за допомогою спеціалізованого програмного забезпечення. Кожен з цих підходів існує та використовується, і вони мають свої особливості у використанні.

Щодо кластерного обчислення, то слабким місцем у цьому випадку є обмін даним між комп’ютерами мережею, яка їх з’єднує (зазвичай швидкість комп’ютерних мереж поступається в часі навіть зверненням процесора до пам’яті жорсткого диску). Кращими задачами для розрахунку за допомогою кластеру є ті, що можна поділити на окремі локалізовані частини, виконання яких не залежить від результату інших

складових та час розрахунку є порівняно більшим, ніж час обміну інформацією мережею.

У випадку використання графічних процесорів (GPU) тип задач, що є для них найбільш підходящим визначено історично, так як їх будова направлена у першу чергу на швидку обробку максимально великого об’єму даних одночасно нескладними операціями (зміна яскравості або кольору пікселів на екрані: операції додавання, множення тощо). Але потреба нових рішень проблем з швидкістю виконання команд дозволила розвинути графічні процесори з іншої сторони – у якості паралельних багатоядерних розрахункових пристроїв.

ТЕОРЕТИЧНА ЧАСТИНА

1 Метод молекулярної динаміки

Для дослідження поведінки системи частинок та її властивостей використовується підхід безпосереднього моделювання законів руху, взаємодії та зовнішніх сил, що діють у її межах. Математична постановка задачі молекулярної динаміки заснована на рівняннях класичної механіки з використанням певних законів взаємодії між частинками. Ці закони описуються за допомогою потенціалів [1, 2].

Потенціал Леннард-Джонса: дозволяє розрахувати значення потенціальної енергії між двома частинками, що розташовані на деякій відстані.

$$U(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right], \quad (1.1)$$

$$F(r) = 24\frac{\varepsilon}{\sigma}\left[2\left(\frac{\sigma}{r}\right)^{13} - \left(\frac{\sigma}{r}\right)^{7}\right], \quad (1.2)$$

де $U(r)$ – потенціальна енергія взаємодії, ε – це енергія зв'язку, σ – рівноважна відстань для пари атомів ($U(\sigma)=0$), r – відстань між парою частинок, $F(r)$ – сила взаємодії. Потенціал Леннард-Джонса широко використовується завдяки простоті розрахунку (див. Рис. 1.1, Рис. 1.2) [1, 2, 3].

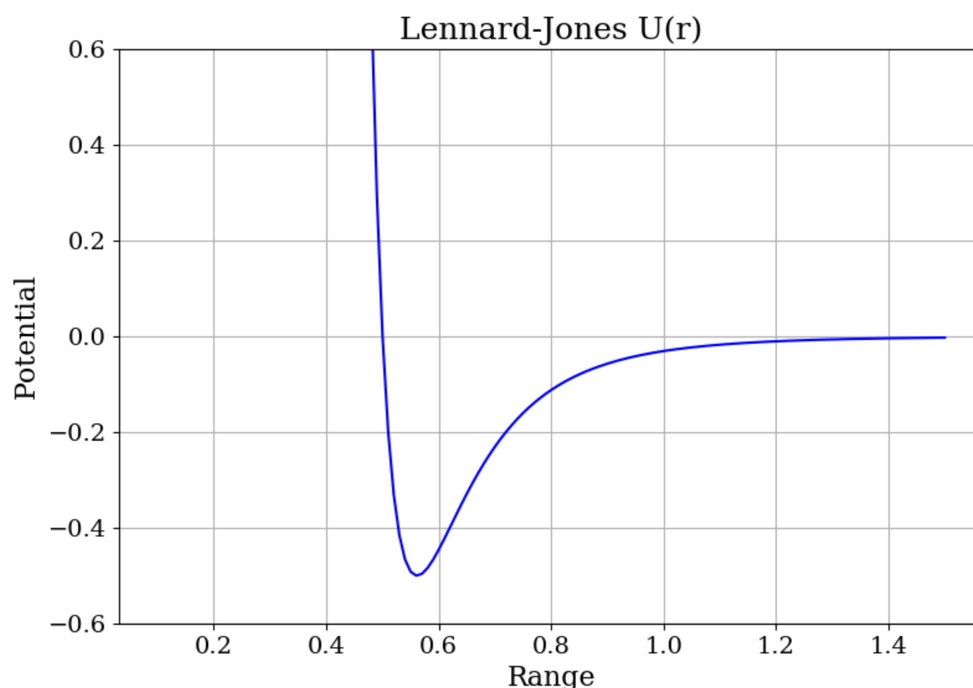

Рис. 1.1 – форма залежності потенціалу Леннард-Джонса від відстані між частинками

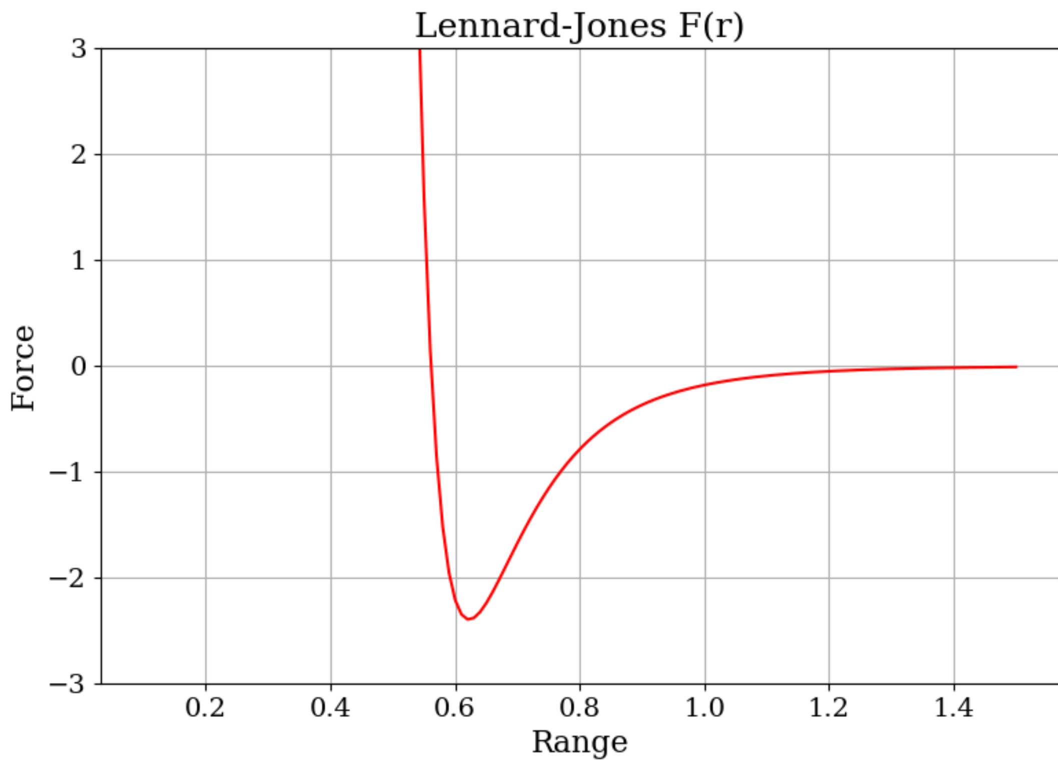

Рис. 1.2 – форма залежності сили Леннард-Джонса від відстані між частинками

Потенціал Мі: потенціал та сила мають наступний вигляд:

$$U(r) = \frac{\varepsilon}{n-m}\left[m\left(\frac{\sigma}{r}\right)^{n} - n\left(\frac{\sigma}{r}\right)^{m}\right], \tag{1.3}$$

$$F(r) = \frac{nm}{n-m}\frac{\varepsilon}{\sigma}\left[\left(\frac{\sigma}{r}\right)^{n+1} - \left(\frac{\sigma}{r}\right)^{m+1}\right], \tag{1.4}$$

де n, m – параметри взаємодії. Потенціал Мі можна використовувати у випадках, коли існує необхідність більш гнучкого налаштування правила взаємодії частинок у системі через те, що він має вже чотири параметри (див. Рис. 1.3, Рис. 1.4) [1, 2].

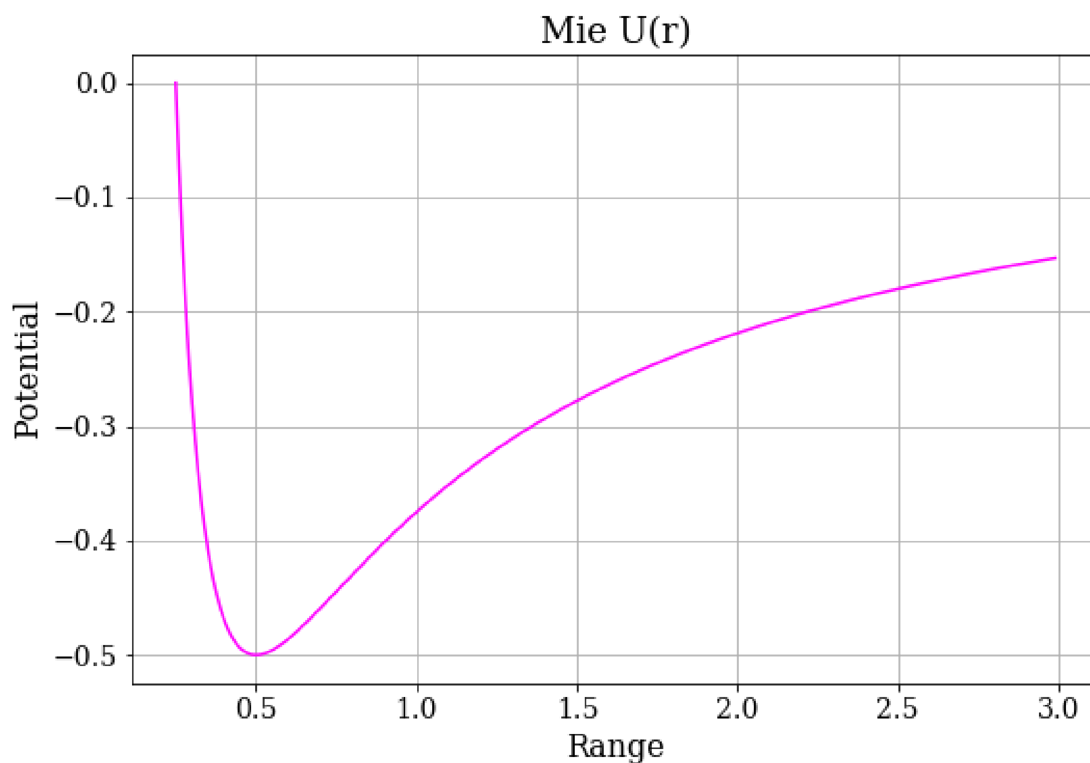

Рис. 1.3 – форма залежності потенціалу Мі від відстані між частинкам

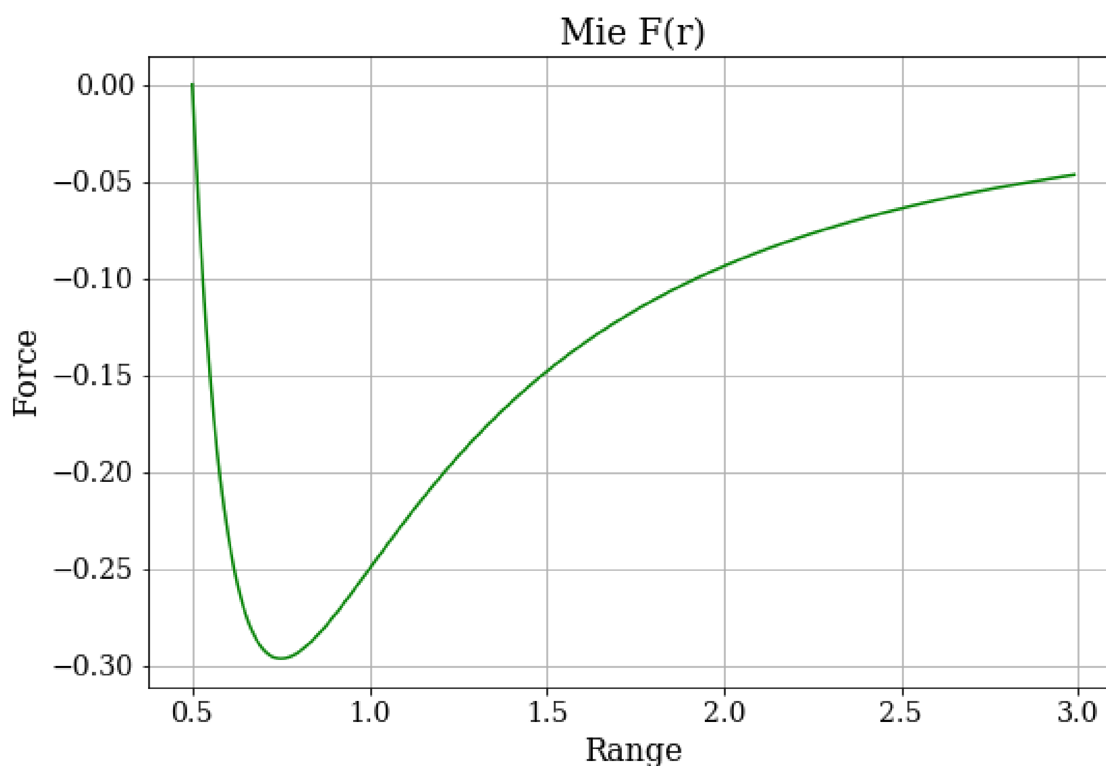

Рис. 1.4 – форма залежності сили Мі від відстані між частинкам

Модифікований потенціал використовуються у разі необхідності змінити радіус дії потенціалу, але зберегти його властивості. З цією метою можна додати наступні зміни в формулу:

$$\hat{U}(r)=U(k(r-\sigma)+\sigma), \tag{1.5}$$

де $\hat{U}(r)$ – модифікований потенціал, а $U(k(r-\sigma)+\sigma)$ – це деякий потенціал взаємодії частинок (наприклад: Леннард-Джонса або Мі). Якщо $k=1$, то вихідний та змінений потенціали будуть однаковими $\left(\hat{U}(r)=U(r)\right)$. У випадку $k>1$ потенціал $\hat{U}(r)$ буде «стиснений», відносно початкового $U(r)$. При $k<1$ потенціал навпаки «буде розтягнений» (див. Рис. 1.5) [1, 2].

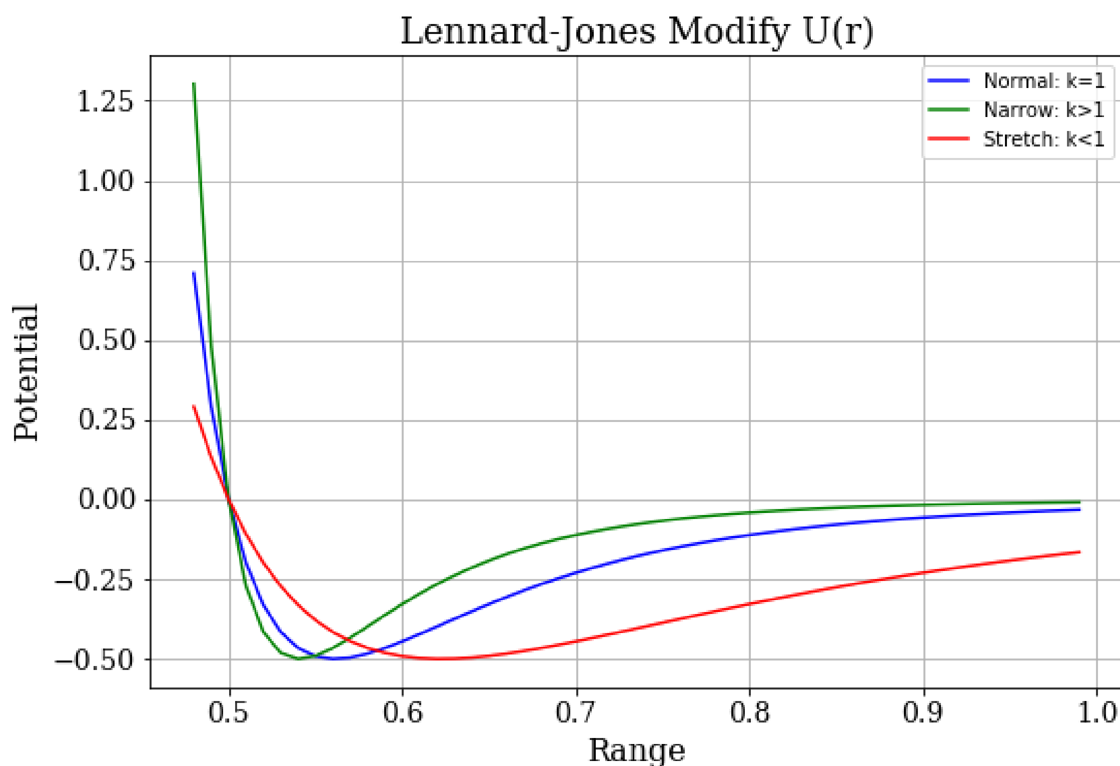

Рис. 1.5 – форма залежності модифікованого потенціалу Леннард-Джонса від відстані між частинкам

Відповідно модифікована сила буде виглядати (див. Рис. 1.6) [1, 2]:

$$\hat{F}(r)=kF(k(r-\sigma)+\sigma). \tag{1.6}$$

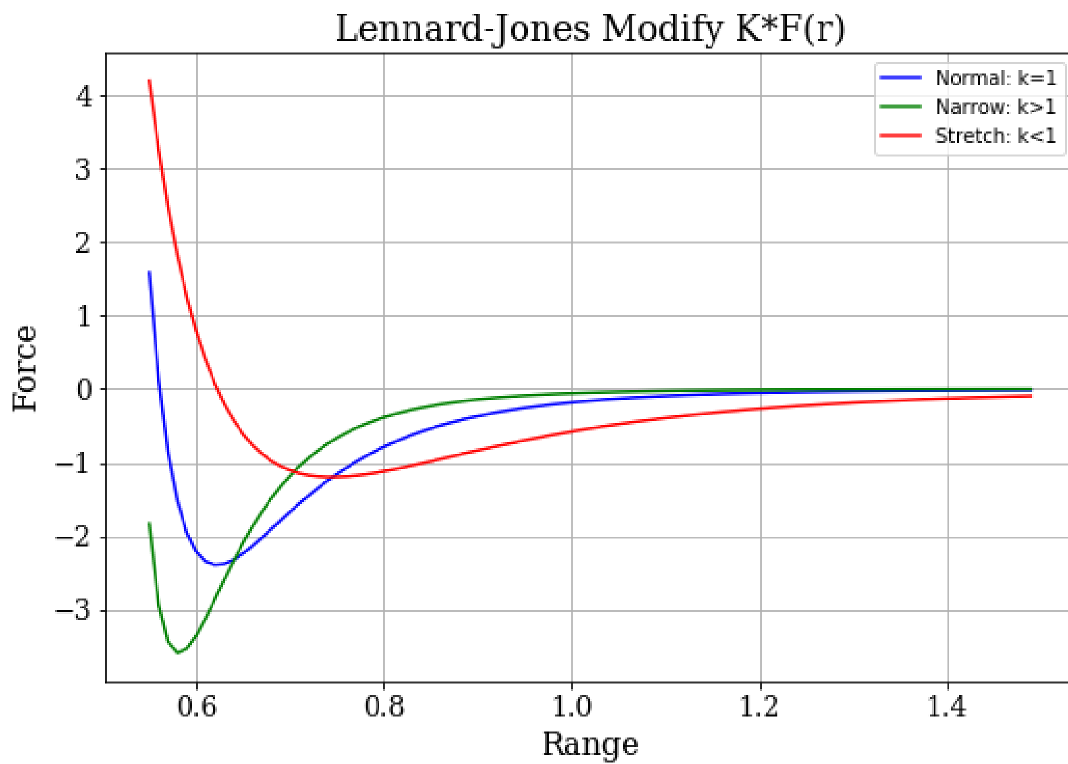

Рис. 1.6 – форма залежності сили від відстані між частинкам (модифікований потенціал Леннард-Джонса)

Програма для побудови графіків залежності сили та потенціалу від відстані знаходиться в розділі ДОДАТОК А.

2 Особливості будови CPU та GPU, відмінності та переваги кожної архітектури

Центральний процесор розроблено таким чином, щоб була змога виконувати якомога ширший спектр завдань, а також, щоб за максимально короткий час виконати команди та мати змогу миттєво перемикатися. Швидкість виконання інструкцій в основному збільшується за рахунок підвищення тактової частоти та збільшенням об'єму кеш-пам'яті, час доступу до якої є набагато меншим, ніж до оперативної пам'яті комп'ютера. З іншого боку GPU створено так, щоб за короткий проміжок обробляти максимально великий об'єм даних. Такий ефект досягається за рахунок того, що графічний процесор складається з великої кількості процесорів, але меншої потужності та оптимізації пам'яті для максимальної пропускної здатності [4].

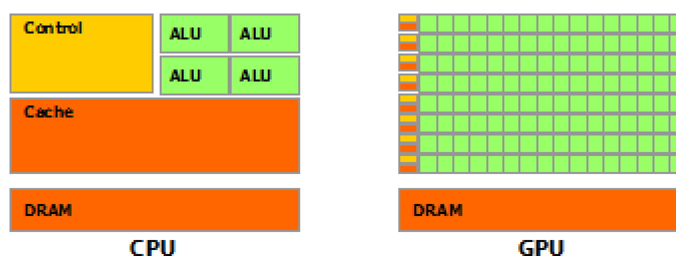

Рис. 2.1 – порівняння архітектури CPU та GPU. Зеленим позначені транзистори, що виконують обчислення; жовтим – ті, що керують потоком; помаранчевим область, яка відповідає пам'яті [5]

Зараз тенденція підвищення тактової частоти CPU замінилася на користь мультипроцесорів (всередині центрального процесору знаходиться декілька окремих ядер, які теж можуть одночасно виконувати інструкції). Але цей факт не зменшив ваги GPU як пристрою, що дозволяє проводити розрахунки, що пов'язані не тільки з графікою [4].

3 Існуючі методи прискорення розрахунків характеристик систем на основі методу молекулярної динаміки

Існує декілька способів, що дозволяють оптимізувати розрахунки. Ці методи спираються на деякий радіус, що дозволяє зменшити кількість частинок, що необхідні для розрахунку нового прискорення кожної частинки.

1. **Список Верле** – перед розрахунком сил взаємодії для кожної частинки створюється окремий список сусідніх частинок, що ввійшли в межі сфери з радіусом r_c, що за значенням менший, ніж область, що використовується для моделювання. Оновлення списку відбувається кожні 10 – 20 кроків за часом [6].
2. **Метод зв’язаних списків** – область розрахунку поділяється на частини (комірки), розмір яких визначається за допомогою радіусу обмеження r_c. На початку розрахунків створюється масив, що містить список номерів сусідніх комірок для кожної окремо. Частинки взаємодіють лише з частинками контейнеру, до якого належать, а також із тими, що знаходяться в найближчих [6].

У даній роботі для підвищення ефективності розрахунків теж використовується радіус обмеження. Але спосіб, у який він використовується дещо модифіковано: частинки, що використовуються для обрахування прискорення, відбираються за допомогою радіусу безпосередньо на етапі розрахунку без використання додаткових списків.

ПРАКТИЧНА ЧАСТИНА

4 Математична модель

Завданнями даної роботи є побудова комп'ютерної моделі системи частинок, які взаємодіють між собою, за допомогою методу молекулярної динаміки. Порівняння реалізацій на базі CPU (Central Processing Unit) та об'єднання архітектури CPU та GPU (Graphics Processing Unit). Прискорення розрахунків характеристик за допомогою введення радіусу, що буде розділяти частинки на дві частини: ті, що впливають на деяку частинку, прискорення якої обраховується, та ті, впливом котрих можна нехтувати. Для досягнення цієї мети спочатку необхідно побудувати математичну модель системи взаємодіючих частинок.

Система складається з N хімічно нейтральних частинок, які можуть рухатися у будь-якому напрямку вздовж осей OX, OY, OZ відповідно до другого закону Ньютона:

$$\vec{F}(x, y, z) = m\vec{a}(x, y, z), \tag{4.1}$$

де $\vec{F}$ – це сумарний вектор сил, що діють на частинку, m – маса частинки, $\vec{a}$ – вектор прискорення частинки.

Тіла взаємодіють між собою, цей взаємозв'язок описується за допомогою парного потенціалу Леннард-Джонса (1.1) [1, 2, 3, 7].

У рамках даного моделювання приймаємо наступні припущення:

- маса m всіх частинок є однаковою та рівна умовній 1 одиниці маси;
- енергія взаємозв'язку ε між молекулами дорівнює 0.5 умовних одиниць;
- рівноважна відстань σ рівна 0.5 умовних одиниць.

Наслідком таких припущень є повне приведення системи до умовних величин.

Крайові умови – періодичні. Тобто система частинок обмежена прямокутним контейнером з деякими значеннями висоти, ширини й довжини. При потраплянні частинки на одну зі стінок (границь) боксу, вона «миттєво» з'явиться на протилежній без зміни своїх параметрів швидкості та прискорення. Наслідком такого обмеження є наступний факт: максимальна відстань між частинками може бути рівною [3]:

$$r_{\max} = \sqrt{0.5h^2 + 0.5w^2 + 0.5d^2}, \tag{4.2}$$

де $r_{\max}$ – максимальна відстань між частинками, h – висота контейнеру (в умовних одиницях координат), w – ширина контейнеру (в умовних одиницях координат), d – глибина контейнеру (в умовних одиницях координат).

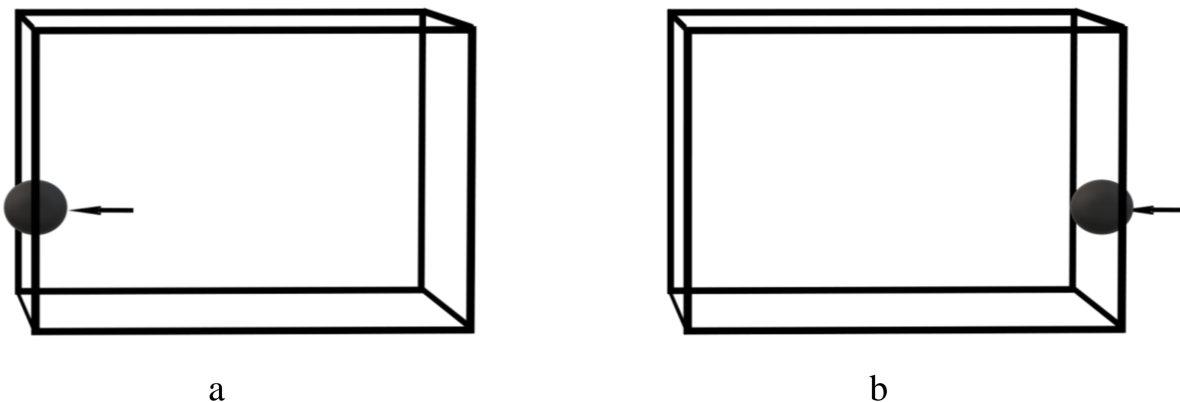

Рис. 4.1 – приклад періодичних умов: a – момент часу t; b – момент часу $t + k \cdot \Delta t$, де Δt – крок за часом

Для розрахунку траєкторії частинок скористаємося алгоритмом Верле:

$$x_{n+1} = 2x_n - x_{n-1} + a_n^x \left(\Delta t\right)^2, \quad (4.3)$$

$$v_n^x = \frac{x_{n+1} - x_{n-1}}{2\Delta t}, \quad (4.4)$$

де x_{n+1} – це значення компоненти x в момент часу $n+1$, x_n – значення компоненти x в момент часу n, x_{n-1} – значення компоненти x у момент часу $n-1$, a_n^x – прискорення за компонентом x у момент часу n, Δt – крок за часом, v_n^x – швидкість за компонентом x у момент часу n [3].

Формули (4.3) і (4.4) можна отримати з розкладу функції $x\left(t_n + \Delta t\right)$ і $x\left(t_n - \Delta t\right)$ у ряд Тейлора:

$$x_{n+1} = x_n + v_n^x \Delta t + \frac{1}{2} a_n \left(\Delta t\right)^2 + O\left[\left(\Delta t\right)^3\right], \quad (4.5)$$

$$x_{n-1} = x_n - v_n^x \Delta t + \frac{1}{2} a_n \left(\Delta t\right)^2 + O\left[\left(\Delta t\right)^3\right], \quad (4.6)$$

де $O\left[\left(\Delta t\right)^3\right]$ – залишковий член (похибка).

Отримати формулу (4.3) можна додавши (4.5) до (4.6):

$$x_{n+1} + x_{n-1} = 2x_n - x_{n-1} + a_n^x \left(\Delta t\right)^2 + 2O\left[\left(\Delta t\right)^3\right]. \quad (4.7)$$

Результат допомагає зрозуміти, що локальна похибка розрахунку координати на кожному кроці має третій порядок [3].

Формулу (4.4) дає різниця (4.5) і (4.6):

$$x_{n+1} - x_{n-1} = 2v_n^x \Delta t \Rightarrow v_n^x = \frac{x_{n+1} - x_{n-1}}{2\Delta t} + O\left[\left(\Delta t\right)^2\right]. \quad (4.8)$$

Отже, локальна похибка розрахунку швидкості має другий порядок [3].

Вигляд формул (4.3) і (4.4) є не досить зручним для використання, адже для розрахунку нової координати, потрібно окрім поточної знати й попередню, а значення швидкості розраховується лише для поточного кроку. Але є спосіб перевести формули (4.3) і (4.4) в іншу еквівалентну форму. Достатньо для (4.3) у праву частину додати пару $\frac{x_{n+1}}{2}-\frac{x_{n+1}}{2}$:

$$x_{n+1}=2x_n-x_{n-1}+a_n^x\left(\Delta t\right)^2+\frac{x_{n+1}}{2}-\frac{x_{n+1}}{2}\Rightarrow$$

$$x_{n+1}=2x_n-\frac{x_{n-1}}{2}-\frac{x_{n-1}}{2}+a_n^x\left(\Delta t\right)^2+\frac{x_{n+1}}{2}-\frac{x_{n+1}}{2}\Rightarrow$$

$$x_{n+1}=2x_n-\frac{x_{n-1}}{2}-\frac{x_{n+1}}{2}+a_n^x\left(\Delta t\right)^2+\frac{x_{n+1}-x_{n-1}}{2}\Rightarrow$$

$$x_{n+1}=2x_n-\frac{x_{n-1}}{2}-\frac{x_{n+1}}{2}+a_n^x\left(\Delta t\right)^2+v_n^x\Delta t\Rightarrow$$

$$x_{n+1}=x_n+v_n^x\Delta t+a_n^x\left(\Delta t\right)^2-\frac{x_{n+1}}{2}+x_n-\frac{x_{n-1}}{2}\Rightarrow$$

$$x_{n+1}=x_n+v_n^x\Delta t+a_n^x\left(\Delta t\right)^2-\frac{1}{2}a_n^x\Delta t\Rightarrow$$

$$x_{n+1}=x_n+v_n^x\Delta t+\frac{1}{2}a_n^x\Delta t \tag{4.9}$$

У викладці (4.9) було використано формулу (4.4) та виражено прискорення з формули (4.3).

А для того щоб перетворити (4.4) необхідно використати наступні кроки за n у (4.4) та (4.3):

$$v_{n+1}^x=\frac{x_{n+2}-x_n}{2\Delta t}, \tag{4.10}$$

$$x_{n+2}=2x_{n+1}+-x_n+a_{n+1}^x\left(\Delta t\right)^2. \tag{4.11}$$

Отже:

$$v_{n+1}^x=\frac{2x_{n+1}+a_{n+1}\left(\Delta t\right)^2-x_n-x_n}{2\Delta t}\Rightarrow v_{n+1}^x=\frac{2x_{n+1}+a_{n+1}\left(\Delta t\right)^2-2x_n}{2\Delta t}\Rightarrow$$

$$v_{n+1}^x=\frac{x_{n+1}+\left(x_{n+1}-2x_n+x_{n-1}\right)-x_{n-1}+a_{n+1}^x\left(\Delta t\right)^2}{2\Delta t}\Rightarrow$$

$$v_{n+1}^x = \frac{x_{n+1} - x_{n-1}}{2\Delta t} + \frac{x_{n+1} - 2x_n + x_{n-1}}{2\Delta t} + \frac{1}{2} a_{n+1}^x \Delta t \Rightarrow$$

$$v_{n+1}^x = v_n^x + \frac{1}{2} a_n^x \Delta t + \frac{1}{2} a_{n+1}^x \Delta t \Rightarrow v_{n+1}^x = v_n^x + \frac{1}{2}\left(a_n^x + a_{n+1}^x\right)\Delta t. \quad (4.12)$$

Результатами (4.9) та (4.12) є наступні формули:

$$x_{n+1} = x_n + v_n^x \Delta t + \frac{1}{2} a_n^x \left(\Delta t\right)^2, \quad (4.13)$$

$$v_{n+1}^x = v_n^x + \frac{1}{2}\left(a_{n+1}^x + a_n^x\right)\Delta t, \quad (4.14)$$

Отримані рівняння дозволяють розраховувати координати та швидкості частинок маючи лише початкові значення координат, швидкостей і прискорень, у той час як у вигляді формул (4.3) і (4.4) потрібно розрахувати перший крок за допомогою іншого методу. Вирази для розрахунку швидкості та координати за компонентами y та z тотожні зазначеним (4.13) та (4.14) з точністю до зміни позначення компоненти [3].

5 Програмна реалізація

5.1 Загальна схема програми

Для побудови програми використано наступні інструменти:

- Мова програмування Python:
  - інтегроване середовище розробки (Integrated Development Environment, IDE) PyCharm;
  - набір бібліотек:
    - Numpy – для роботи з масивами та типами даних;
    - Panda3D – для роботи з вікнами програм, кнопками, формами, текстом а також графікою, зокрема 3D;
    - Matplotlib – для роботи з 2D графікою;
    - PyCuda – для доступу до паралельної архітектури CUDA, що дозволяє використовувати графічний процесор для обчислень;
    - Time та Timeit – для роботи з часом;
    - Copy – для доступу до інструментів копіювання;
    - Random – для доступу до функцій генерування псевдовипадкових чисел.
- Архітектура для паралельних обчислень компанії NVIDIA – CUDA.

Загальний алгоритм роботи програми:

1. Запуск:

1.1. Відкриття вікна програми, на якому розміщено:

1.1.1. Поля для введення параметрів системи з підписами:

1.1.1.1. Кількість частинок в системі (натуральне число в діапазоні $\left[2;2^{32}-1\right]$, а також обмеження накладаються з боку апаратної складової, наприклад: може закінчитися пам’ять для зберігання необхідних даних).

1.1.1.2. Інтервал початкових швидкостей частинок (приймає на вхід діапазон значень, наприклад, максимальний інтервал $\left[-1;1\right]$).

1.1.1.3. Кількість частинок у рядку (довжина обмежуючого контейнера, натуральне число в діапазоні $\left[2;2^{32}-1\right]$, а також накладаються обмеження пам’яті).

1.1.1.4. Кількість частинок у стовпчику (ширина обмежуючого контейнера, натуральне число в діапазоні $\left[2;2^{32}-1\right]$, а також накладаються обмеження пам’яті).

1.1.1.5. Кількість частинок вглибину (глибина обмежуючого контейнера, натуральне число в діапазоні $\left[2;2^{32}-1\right]$, а також накладаються обмеження пам'яті).

1.1.2. Кнопки для запуску виконання симуляції та побудови графіків:

1.1.2.1. **Read with output** – кнопка для ініціалізації системи з виводом на екран частинок у вигляді 3D-об'єкту у вигляді сфери.

1.1.2.2. **Read without output** – кнопка для ініціалізації системи без виводу на екран 3D-об'єктів.

1.1.2.3. **GPU vs CPU** – кнопка для проведення тесту швидкості моделювання системи частинок для окремих реалізацій CPU та гібридної системи CPU та GPU та побудови відповідних графіків.

1.1.2.4. **Create table with Radius –** кнопка для проведення експериментів та побудови таблиць, що необхідні для виявлення оптимального радіусу обмеження.

2. Вибір кнопки:

2.1. **Read with output** – читання введених даних, побудова системи та виведення її на екран, відображення на екрані варіантів запуску комп'ютерного моделювання:

2.1.1. **CPU** – запуск розрахунку на базі центрального процесора.

2.1.2. **GPU** – запуск розрахунку на базі графічного процесора.

2.2. **Read without output** – читання введених даних, побудова системи без виводу на екран, відображення кнопок для вибору обчислювальної платформи:

2.2.1. **CPU** – запуск розрахунку на базі центрального процесора.

2.2.2. **GPU** – запуск розрахунку на базі графічного процесора.

2.3. **GPU vs CPU** – запуск проведення тесту швидкості розрахунків станів системи окремо на базі CPU та з використанням GPU, побудова графіку та його збереження.

2.4. **Create table with Radius** – запуск експерименту для побудови таблиць, що є основою для визначення оптимального радіусу обмеження.

3. Запуск моделювання системи частинок:

   Можливі 2 режими роботи:
   - розрахунок за допомогою CPU;
   - розрахунок на базі CPU та GPU.

4. Можливість збереження графіку енергій у будь-який момент часу після початку моделювання за допомогою кнопки **SaveSysEnergyPlot** у файл *Plot.png*.

Лістинг основного модуля програми знаходиться в ДОДАТОК В. Лістинг програми для запуску знаходиться в ДОДАТОК Б.

5.2 Реалізація функції ініціалізації системи

Функції ініціалізації системи працюють за наступною послідовністю:

1. Зчитування введених значень з полів у відповідні змінні.
2. Підготовка необхідних констант.
3. Перевірка розмірів контейнеру та кількості частинок, у разі якщо кількість частинок більша за добуток введених параметрів контейнера, функція генерації зупиниться й можна буде змінити параметри.
4. У випадку генерації з відображенням частинок:
   1. Маскування початкових полів, написів та кнопок.
   2. Відображення системи в початковому стані.
   3. Відображення кнопок для початку моделювання.
   4. Переміщення камери приблизно в центр по двом осям (*OX*, *OZ*) та на деякій відстані відносно третьої (*OY*).
5. У випадку без відображення частинок:
   1. Відображення кнопок для запуску моделювання.

Створення початкових масивів є однаковим для обох реалізацій, як окремо CPU так і для взаємодії CPU з GPU.

Функції ініціації системи знаходяться в основному модулі програми, що знаходиться в ДОДАТОК В.

Координати генеруються наступним чином:

- для компоненти *x* зберігаються результати виразу:

$$x_i = i\,\mathrm{mod}(r), \tag{5.1}$$

де x_i – компонент вектору координат частинки з номером i, i – номер частинки $\left(i=\overline{0,N}\right)$, r – кількість частинок у рядку.

- для компоненти *y* зберігаються результати виразу:

$$y_i = \left[i / \left[r / c\right]\right]\mathrm{mod}(d), \tag{5.2}$$

де y_i – компонент вектору координат частинки з номером i, c – кількість частинок у колонці, d – кількість частинок вглиб контейнера, $\left[r / c\right]$ – ціле число від ділення.

- для компоненти z:

$$z_i = \left[r / c\right]\mathrm{mod}(d), \tag{5.3}$$

де z_i – компонент вектору координат частинки з номером i.

Останнім кроком усі отримані значення множаться на значення відстані між частинками (цей хід викликаний особливостями відображення: щоб частинки не «проникали» одна в одну).

Функція генерації координат знаходиться в основному модулі програми, що знаходиться в ДОДАТОК В.

Початкові швидкості генеруються наступним чином:

1. Генерується масив псевдовипадкових значень у діапазоні, що заданий користувачем, але в межах: $[-1;1]$, розміром половини кількості тіл, якщо кількість тіл – не парне число, то береться ціла частина від ділення.
2. Потім створюється один масив на основі двох однакових за модулем, але різними за знаком (це забезпечує виконання умови: сумарний імпульс системи повинен бути рівним 0).
3. Останнім кроком масив перемішується.
4. Тільки у випадку непарної кількості частинок: до загального масиву додається ще одна комірка, у яку записується половина значення, що знаходиться в першій комірці з протилежним знаком.

Функція генерації швидкостей знаходиться в основному модулі програми, що знаходиться в ДОДАТОК В.

5.3 Реалізація функцій розрахунку характеристик системи

Різниця в реалізаціях для окремо CPU та пари CPU, GPU полягає в наступних фактах:

- CPU:
  - координати, швидкості та прискорення являють собою спеціальну структуру даних, яка дозволяє проводити матричні операції, у свою чергу кінетична та потенціальна енергії представляються у вигляді векторів-рядків;
  - координати та швидкості обчислюються відповідно до методу Верле (4.13)-(4.14) у вигляді матричних виразів.
  - Прискорення розраховується за допомогою циклу. Перед початком циклу створюється матриця нулів розміром 3 (кількість орт у системі координат) на N (кількість частинок у системі). За один прохід розраховуються усі компоненти прискорення для однієї частинки:
    - розраховується матриця компонент відстаней відніманням вектору-стовпця від матриці координат (у результаті отримуємо 1 стовпчик нулів);
    - розраховується вектор відстаней (множенням вектору-рядка, який містить лише одиниці, довжиною в кількість вимірів системи координат (у даному випадку 3) на матрицю компонент відстаней, кожне значення котрої піднесене до квадрату), а потім розраховується корінь зі значень отриманого вектору.
    - у позицію, якій відповідає номер частинки записуємо 1 (щоб позбавитися помилки, що пов'язана з діленням на 0).
    - розраховується вектор сил відповідно до формули (1.2).
    - розраховується матриця компонентів сил: вектор сил множиться на кожен рядок матриці компонентів відстаней, а потім кожен рядок матриці розділяється на вектор-рядок відстаней (за рахунок стовпчика нулів у матриці компонентів відстаней отримуємо стовпчик нулів у матриці компонентів сил).
    - враховуємо третій закон Ньютона (сила взаємодії тіл рівна за модулем, але протилежна за знаком), а так як маса в даній моделі рівна умовній одиниці, то від матриці прискорень просто віднімається матриця компонент сил.
    - до відповідного стовпчика матриці прискорень (залежить від номера частинки в циклі) додається вектор-стовпчик, що складається з сум сил у кожному рядку матриці компонент сил.

  - Після прискорення розраховується потенціальна енергія взаємодії частинки відповідного індексу та усіх інших, за допомогою вектору розрахованих відстаней.
- пара CPU, GPU без використання радіусу обмеження:
  - координати, швидкості та прискорення частинок являють собою одновимірні масиви (вектори), так як і масиви енергій потенціальної та кінетичної.
  - розрахунки проводяться по одному елементу, але відразу на всіх доступних для розрахунків потоках графічного процесора. Кількість потоків залежить від моделі відеокарти. У даній роботі використовується графічний процесор компанії NVIDIA модель GEFORCE GTX 1050, яка має 1024 потоки для одного розрахункового блоку. У свою чергу блоки складають мультипроцесор, кожен з яких має 2 блоки в даній моделі відеокарти. Загальна кількість мультипроцесорів складає 5 штук. У результаті отримуємо 10 блоків по 1024 потоки, що дозволяє одночасно обробляти 10240 елементів.
  - розрахунки координати та швидкості проводяться відповідно до формул методу Верле (4.13)-(4.14).
- пара CPU, GPU з використанням радіусу обмеження:
  - послідовність відрізняється від попереднього підпункту лише наявністю додаткової величини (радіусу обмеження), що керує розрахунком прискорень для кожної частинки:
    - частинки, що розташовані на відстані радіусу або ж менше використовуються для оновлення прискорення, а всі інші опускаються (їх вплив вважається рівним 0).

Функції розрахунку координат, швидкостей, прискорень та енергій для CPU-версії знаходяться відповідно в ДОДАТОК Г, ДОДАТОК Ґ, ДОДАТОК Д та ДОДАТОК Ж.

Відповідно для функції для розрахунків на GPU знаходяться в ДОДАТОК З, ДОДАТОК И, ДОДАТОК І, ДОДАТОК Ї, ДОДАТОК Й.

5.4 Реалізація функції, що будує таблиці, на основі яких обирається радіус обмеження

Загальний алгоритм роботи:

1. Формується масиви з кількостями частинок у системі, параметрами контейнера для кожної системи. Початкові швидкості генеруються на основі інтервалу $\left[-1,1\right]$.
2. Формується система частинок та масив радіусів обмеження; початковий стан системи зберігається окремо: координати та швидкості.
3. Проводиться моделювання за допомогою GPU без використання радіусу обмеження.
4. Зберігаються результати моделювання, будуються графіки середньої енергії на частинку, кінетичної середньої енергії на частинку та потенціальної енергії на частинку.
5. Проводиться моделювання за допомогою GPU з використанням радіусу обмеження. Експеримент проводиться для кожного значення радіусу з підготованого раніше масиву. По закінченню результати зберігаються в окремі масиви. Будується таблиця зі значеннями радіусу обмеження, середніми значеннями повної, кінетичної та потенціальної енергій, а також записуються значення, що були отримані в ході моделювання без використання радіусу.

Було проведено два комп'ютерні експерименти з наступними параметрами:

Експеримент 1:

- габаритні параметри контейнера $19\times18\times18$ (у частинках: 19 – ширина, 18 – висота, 18 – глибина – максимальна кількість частинок в контейнері 6156);
- початкові швидкості обрані в межах інтервалу: $\left[-1;1\right]$;
- масив радіусів обмеження сформовано наступним чином: $\left\{r_1, r_2 = r_1 + \Delta r, r_3 = r_2 + \Delta r, ..., r_{\max}\right\}$, де $r_1 = 2\sigma - 2\Delta r$, σ – це рівноважна відстань, що рівна 0.5 умовних одиниць, Δr – це крок значення радіусу обмеження, що дорівнює 0.1, $r_{\max}$ – це максимальне значення радіусу, що визначається за формулою (4.2) і округлене в більшу сторону, у випадку якщо число не ціле;
- час моделювання: 100 кроків при $\Delta t = 0.01$.

У результаті проведення моделювання було отримано наступні графіки повної, кінетичної та потенціальної енергій (усередненої за всіма частинками):

- для системи з 1000 частинок:
  - без використання радіусу:

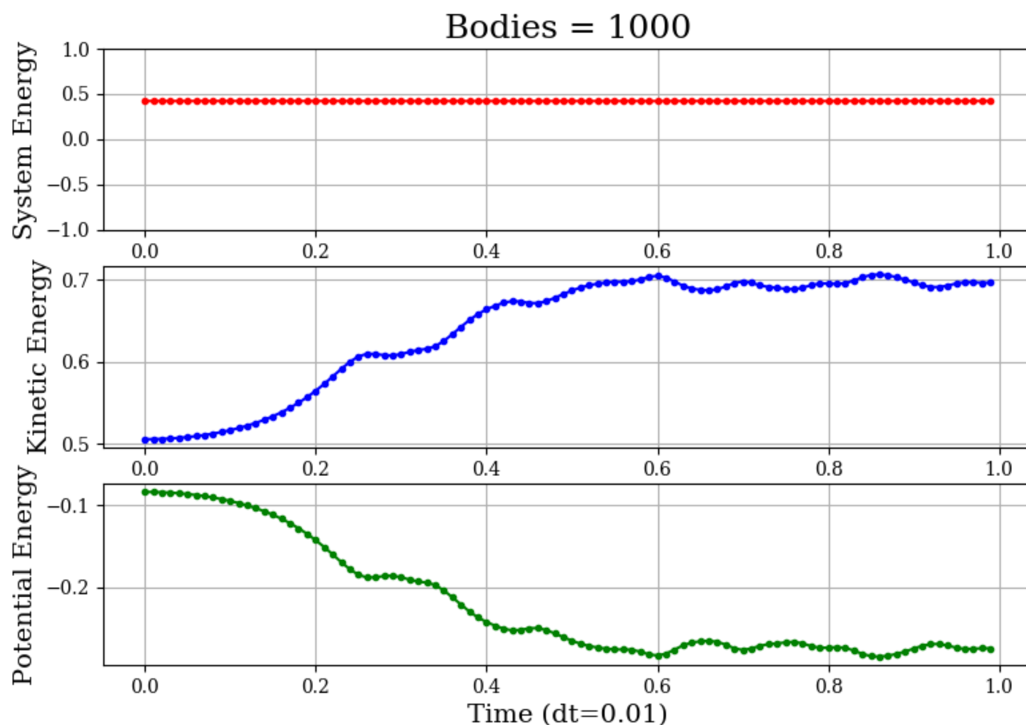

Рис. 5.1 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок без використання радіусу обмеження

- використовуючи радіус обмеження:

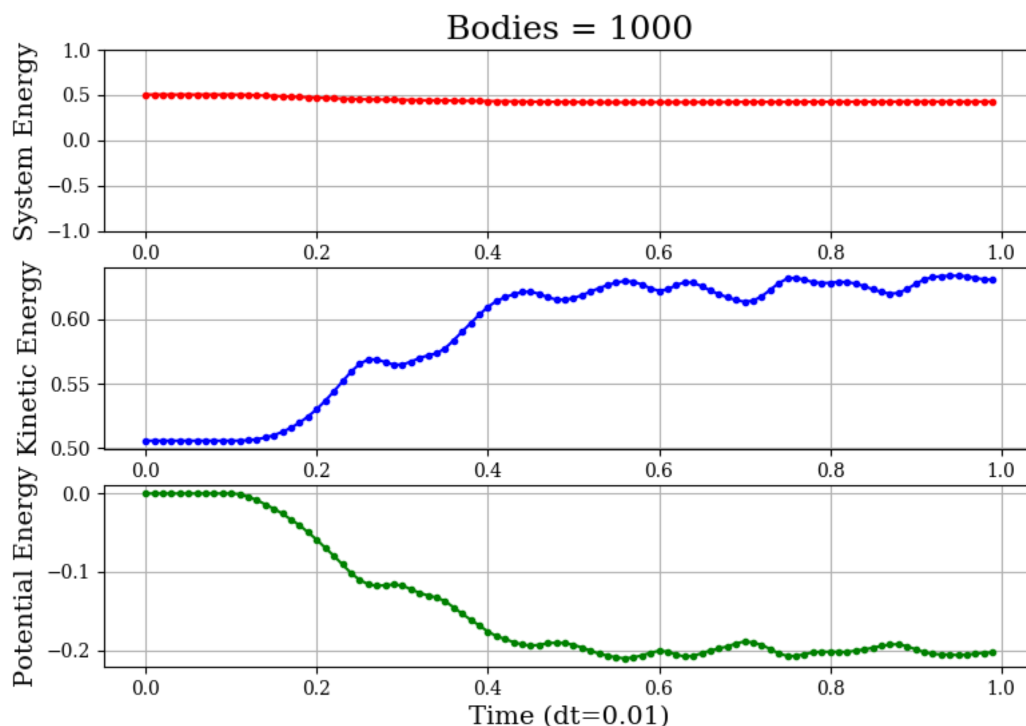

Рис. 5.2 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок, значення радіусу обмеження рівне 0.8 умовних одиниць

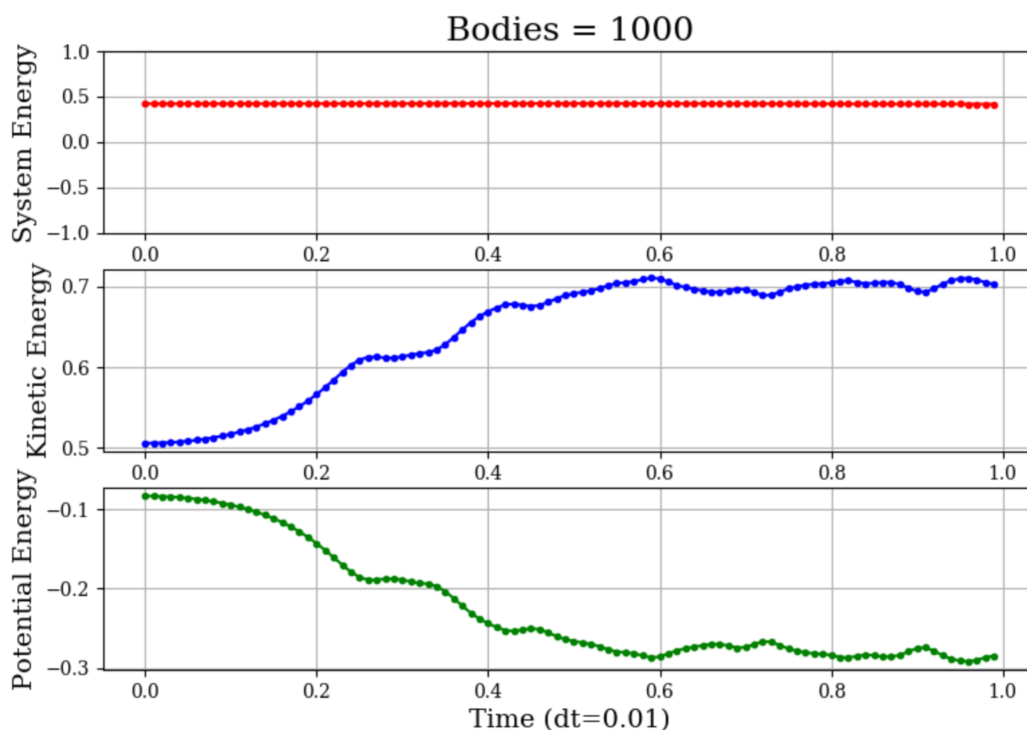

Рис. 5.3 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок, значення радіусу обмеження рівне 2.3 умовних одиниць

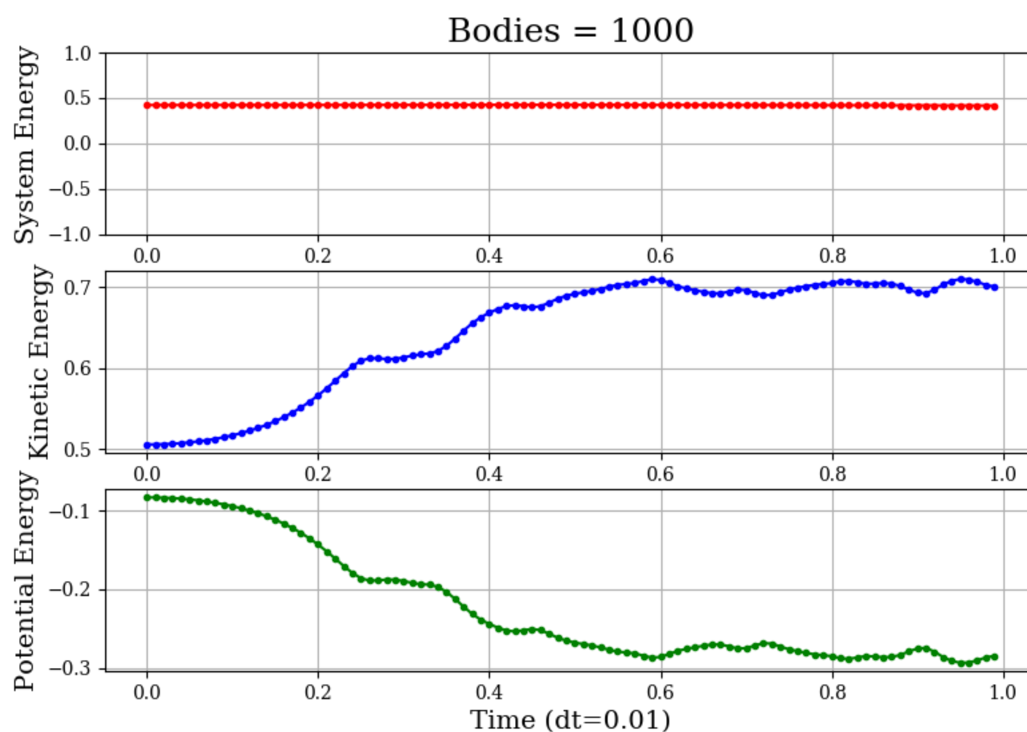

Рис. 5.4 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок, значення радіусу обмеження рівне 28.4 умовних одиниць

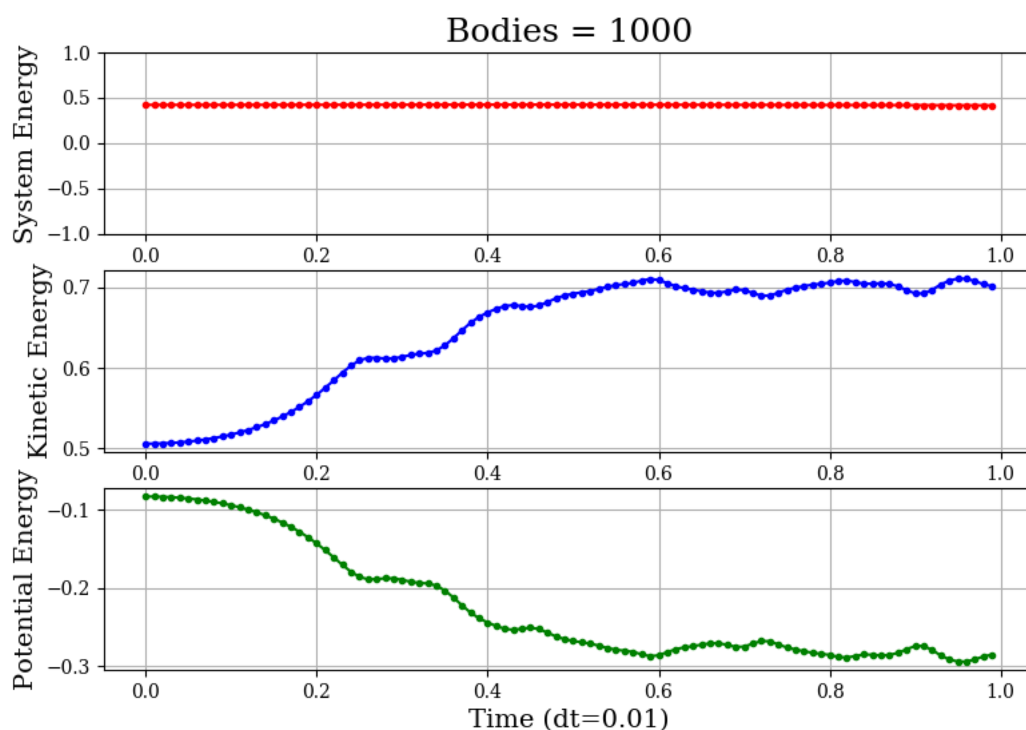

Рис. 5.5 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок, значення радіусу обмеження рівне 40.0 умовних одиниць

- для системи з 2000 частинок:
  - без використання радіусу

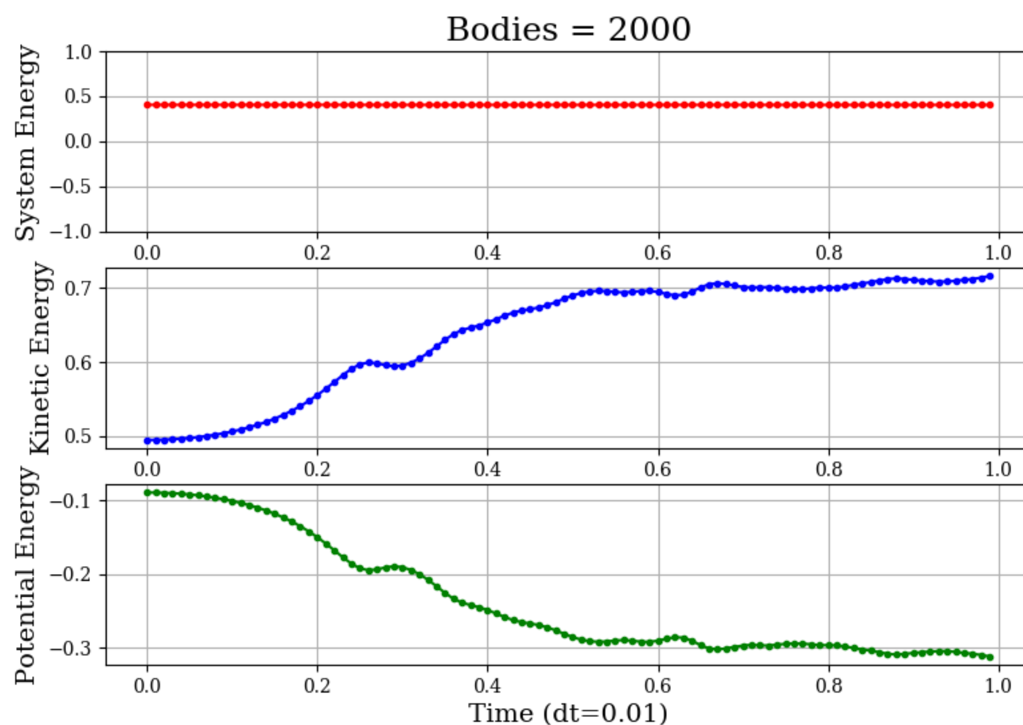

Рис. 5.6 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 2000 частинок без використання радіусу обмеження

- використовуючи радіус обмеження:

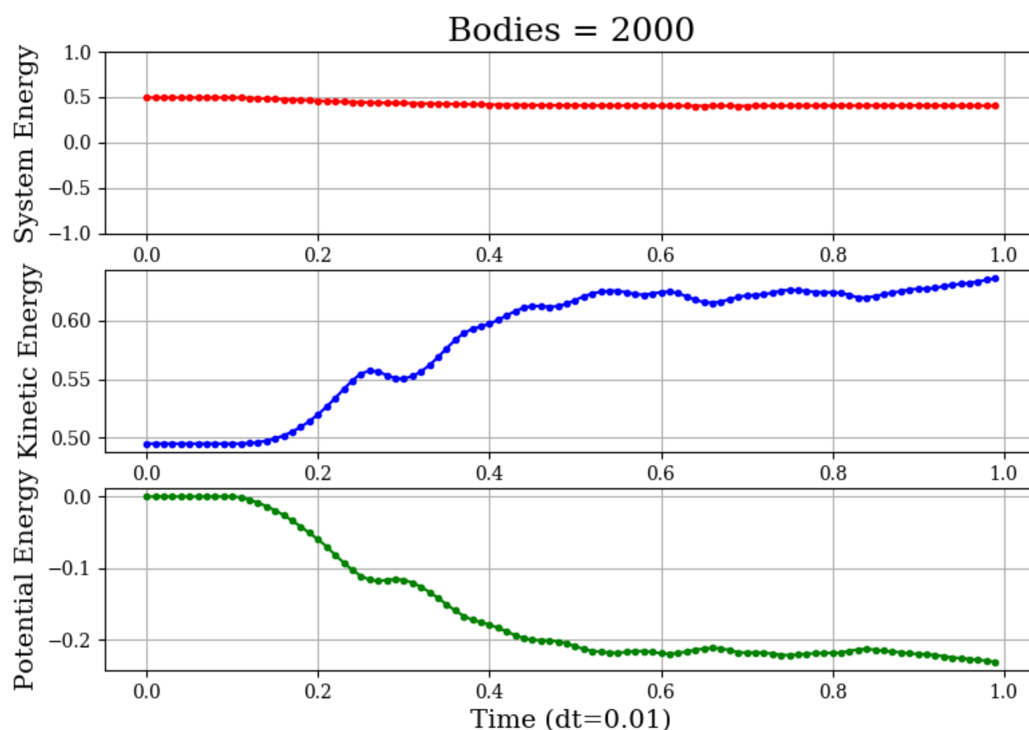

Рис. 5.7 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 2000 частинок, значення радіусу обмеження рівне 0.8 умовних одиниць

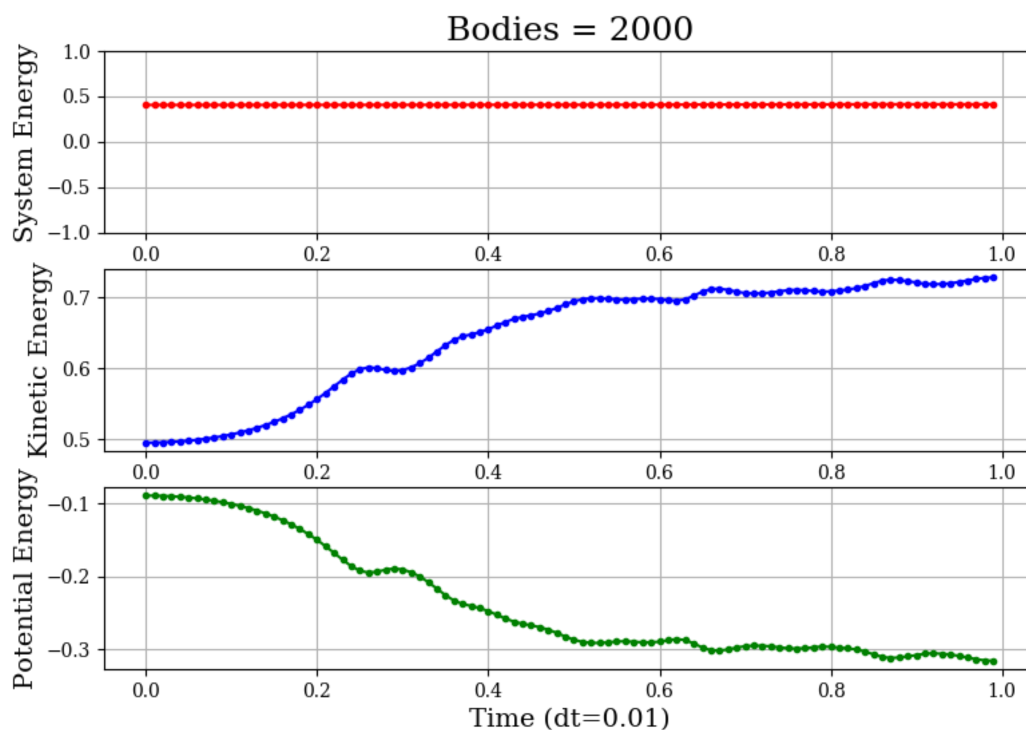

Рис. 5.8 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 2000 частинок, значення радіусу обмеження рівне 2.3 умовних одиниць

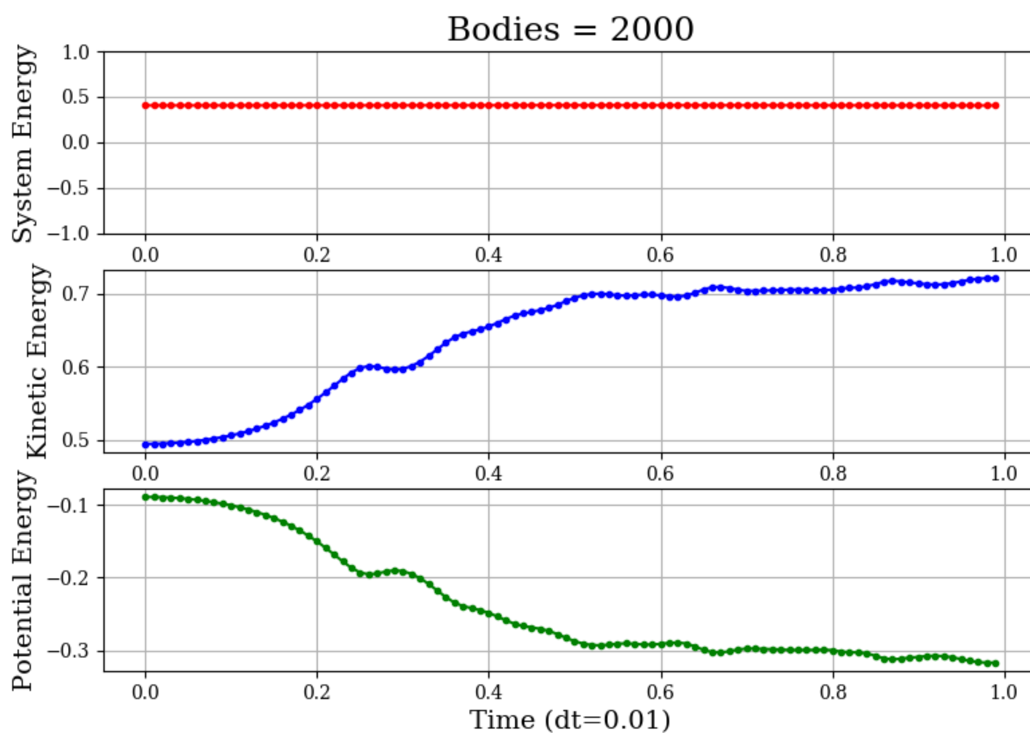

Рис. 5.9 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 2000 частинок, значення радіусу обмеження рівне 28.4 умовних одиниць

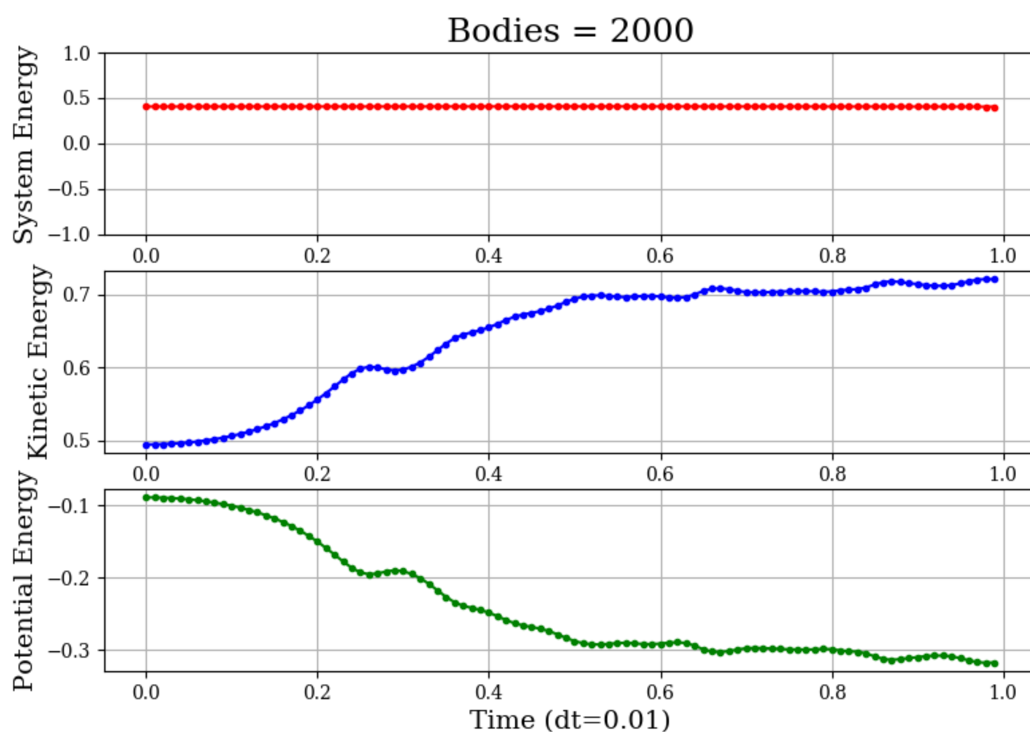

Рис. 5.10 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 2000 частинок, значення радіусу обмеження рівне 40.0 умовних одиниць

- для системи з 3000 частинок:
    - без використання радіусу

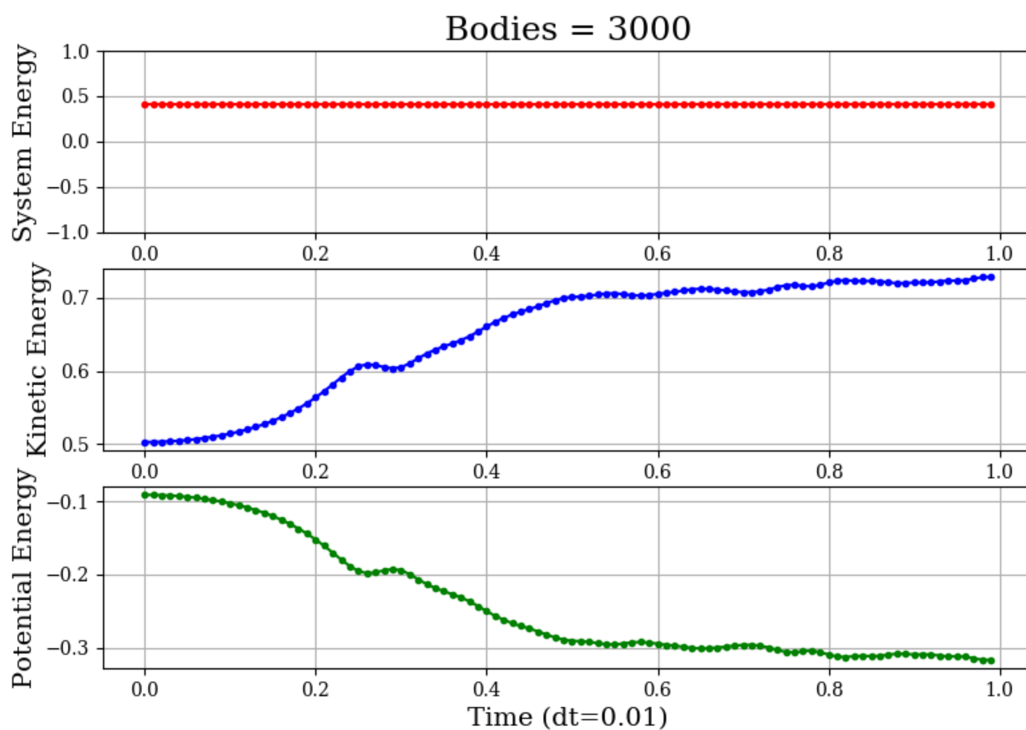

Рис. 5.11 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 3000 частинок без використання радіусу обмеження

- використовуючи радіус обмеження:

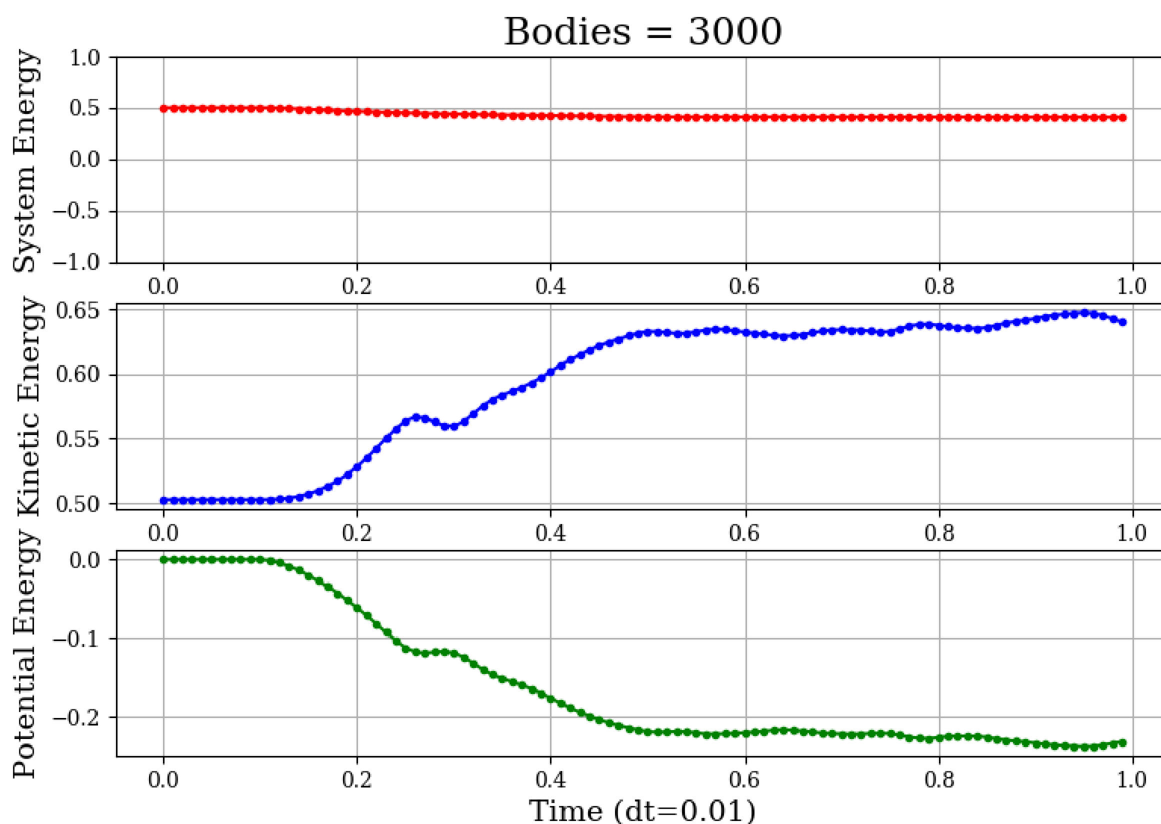

Рис. 5.12 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 3000 частинок, значення радіусу обмеження рівне 0.8 умовних одиниць

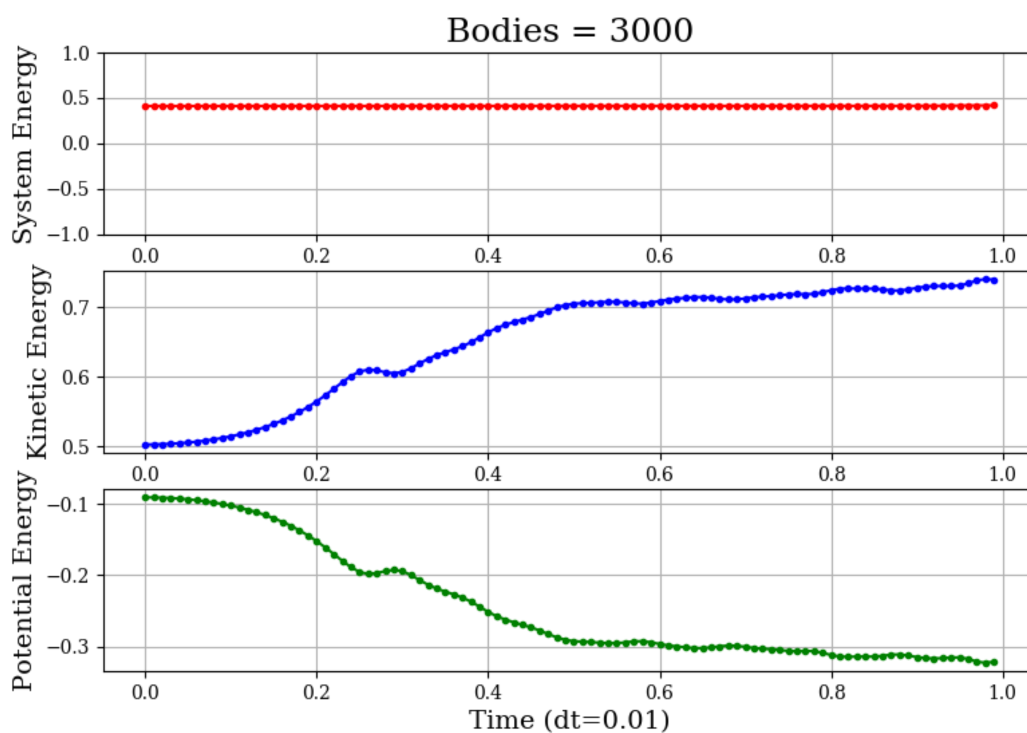

Рис. 5.13 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 3000 частинок, значення радіусу обмеження рівне 2.3 умовних одиниць

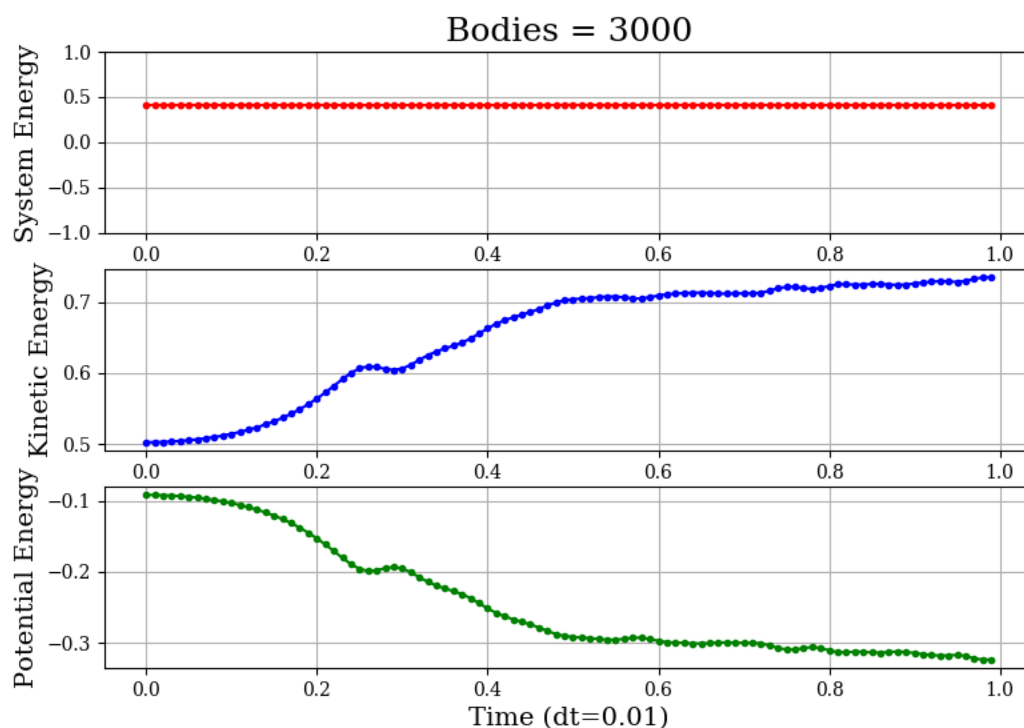

Рис. 5.14 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 3000 частинок, значення радіусу обмеження рівне 28.4 умовних одиниць

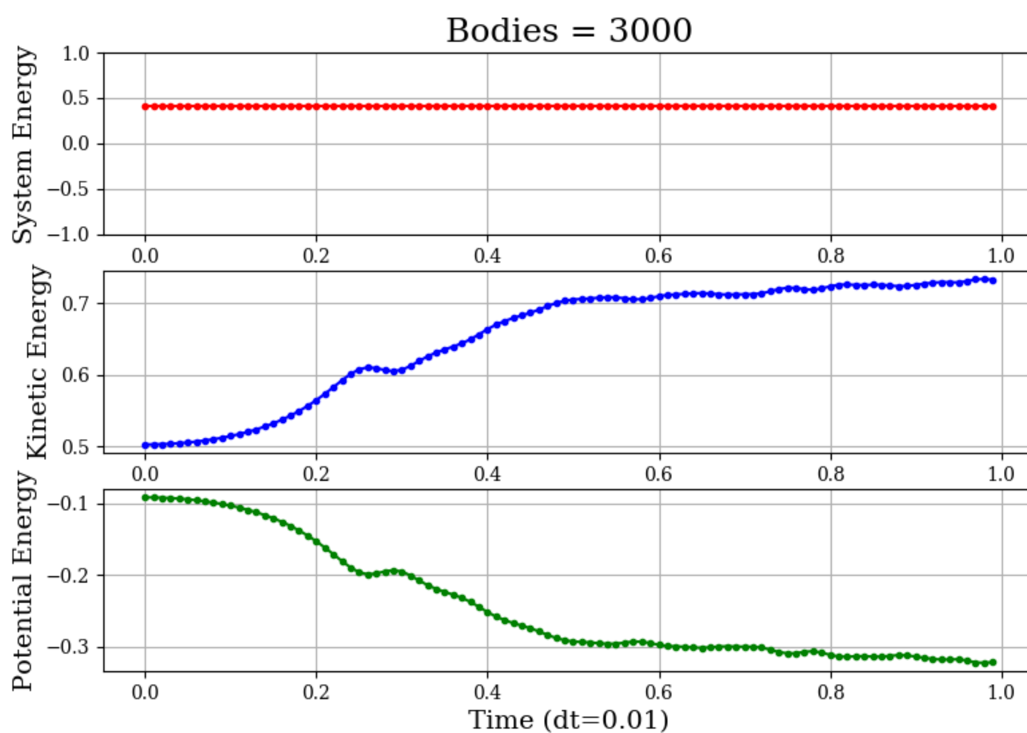

Рис. 5.15 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 3000 частинок, значення радіусу обмеження рівне 40.0 умовних одиниць

- для системи з 4000 частинок:
    - без використання радіусу

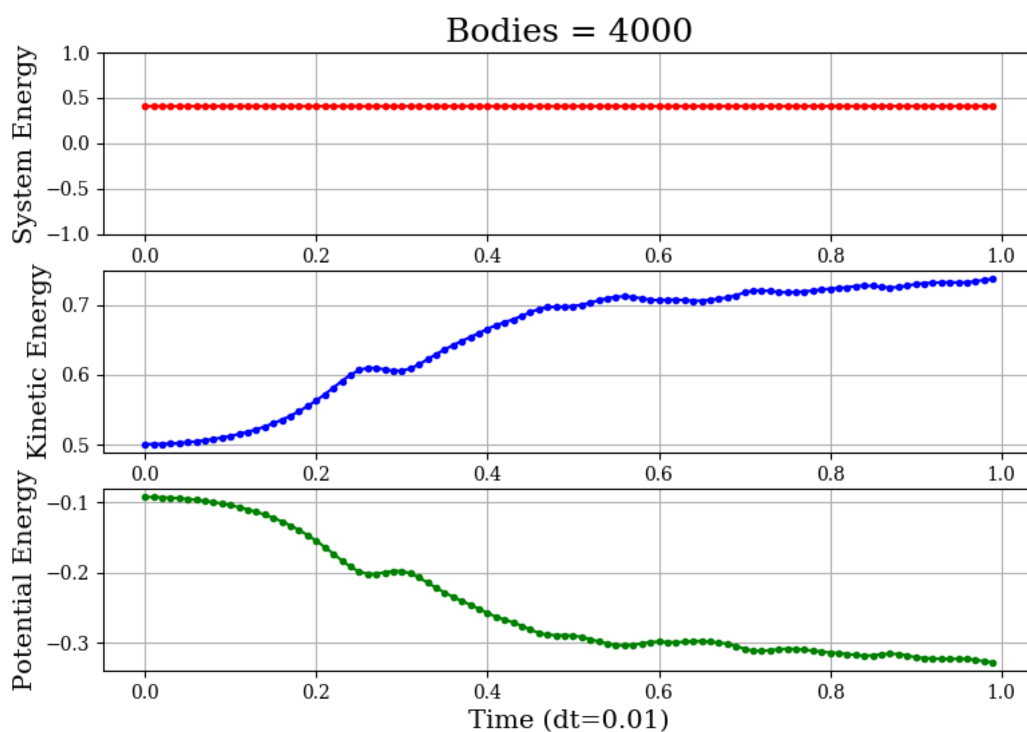

Рис. 5.16 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 4000 частинок без використання радіусу обмеження

- використовуючи радіус обмеження:

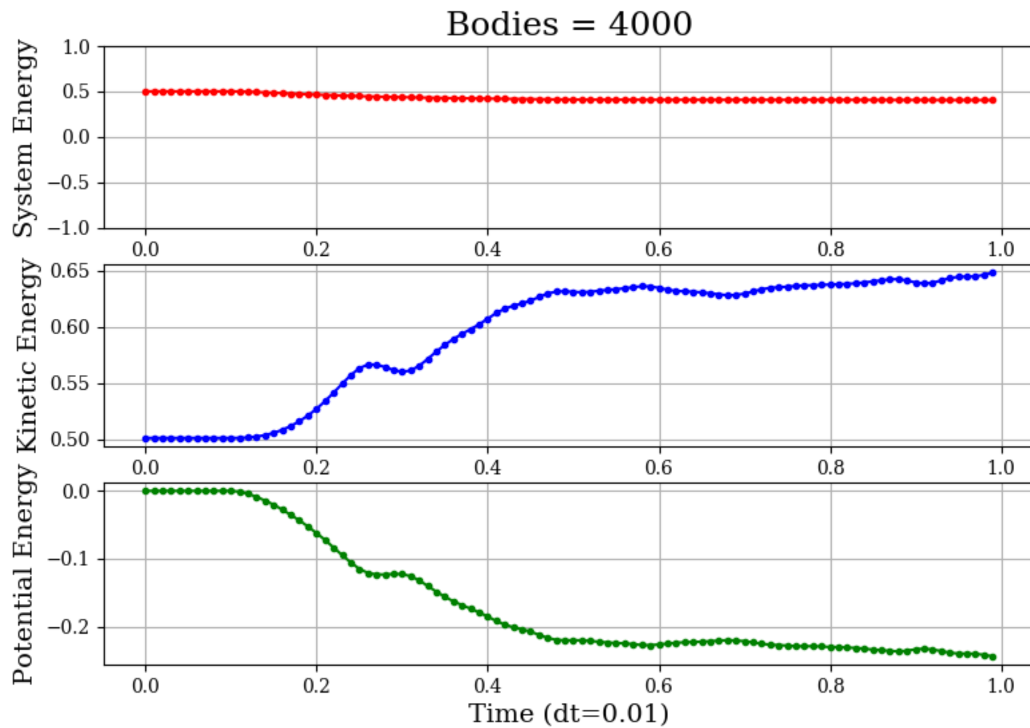

Рис. 5.17 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 4000 частинок, значення радіусу обмеження рівне 0.8 умовних одиниць

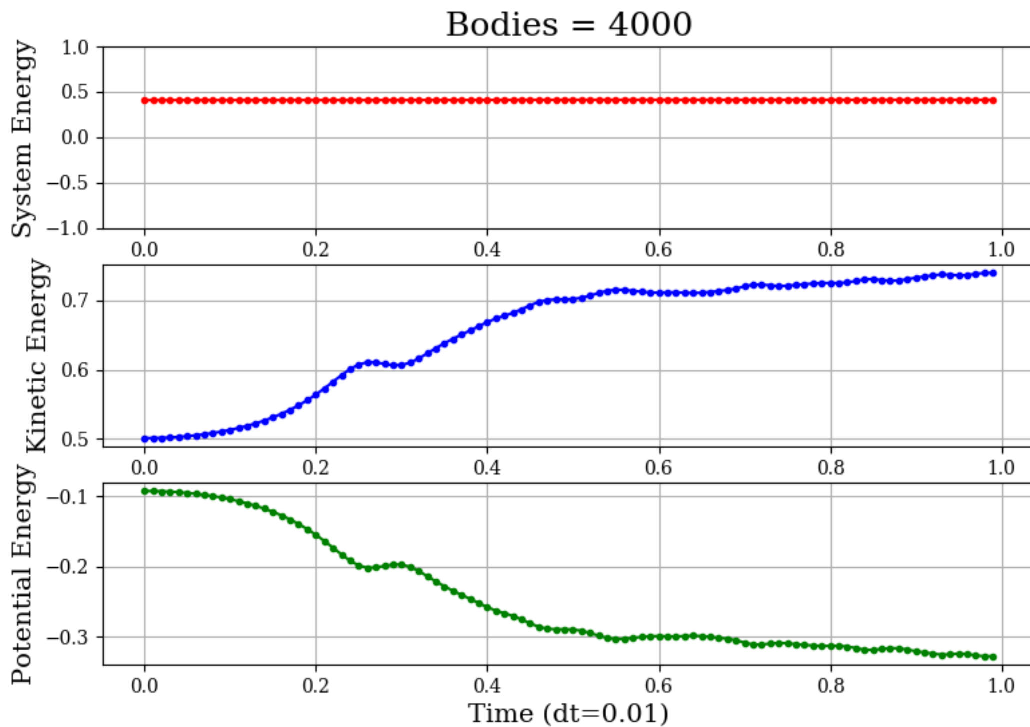

Рис. 5.18 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 4000 частинок, значення радіусу обмеження рівне 2.3 умовних одиниць

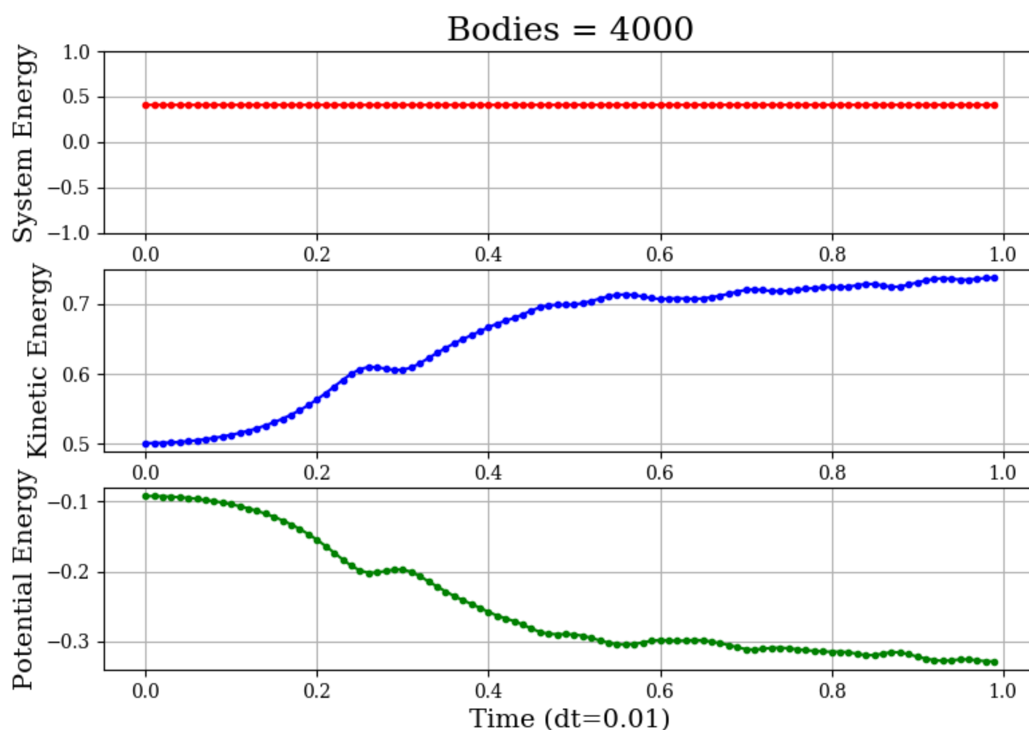

Рис. 5.19 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 4000 частинок, значення радіусу обмеження рівне 28.4 умовних одиниць

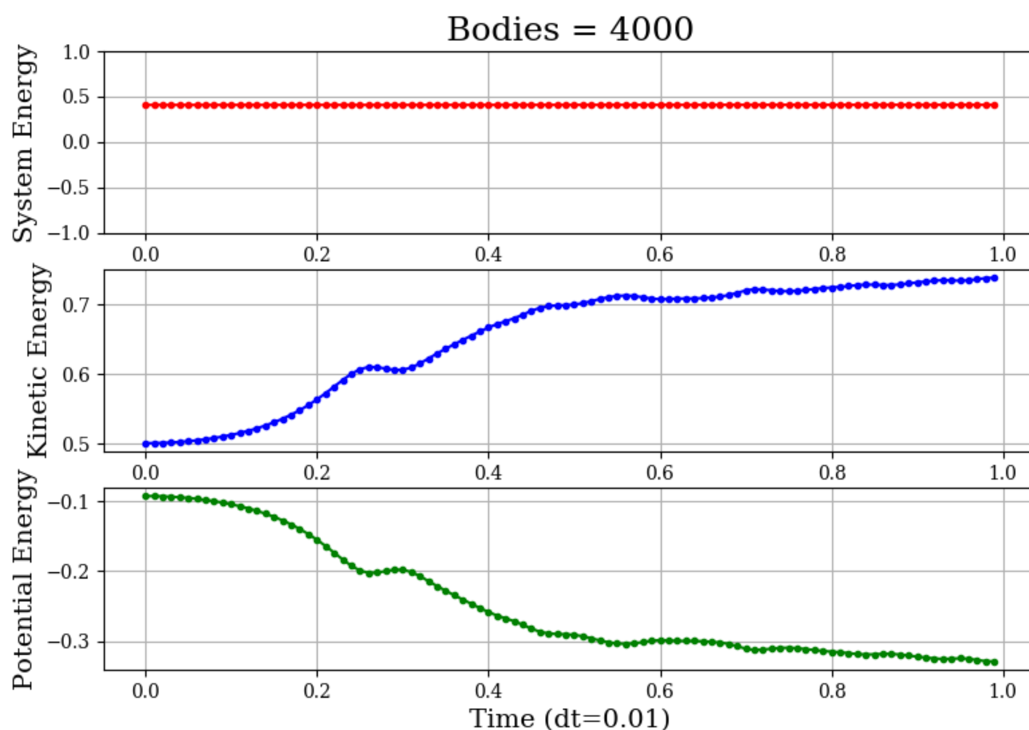

Рис. 5.20 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 4000 частинок, значення радіусу обмеження рівне 40.0 умовних одиниць

- для системи з 5000 частинок:
  - без використання радіусу

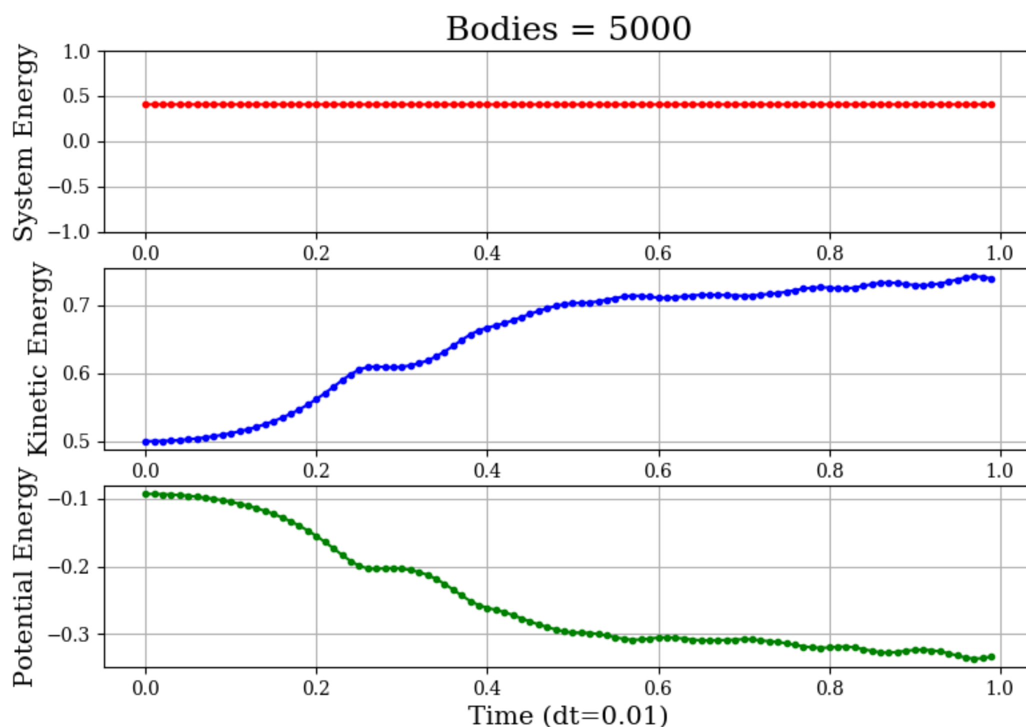

Рис. 5.21 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 5000 частинок без використання радіусу обмеження

- використовуючи радіус обмеження:

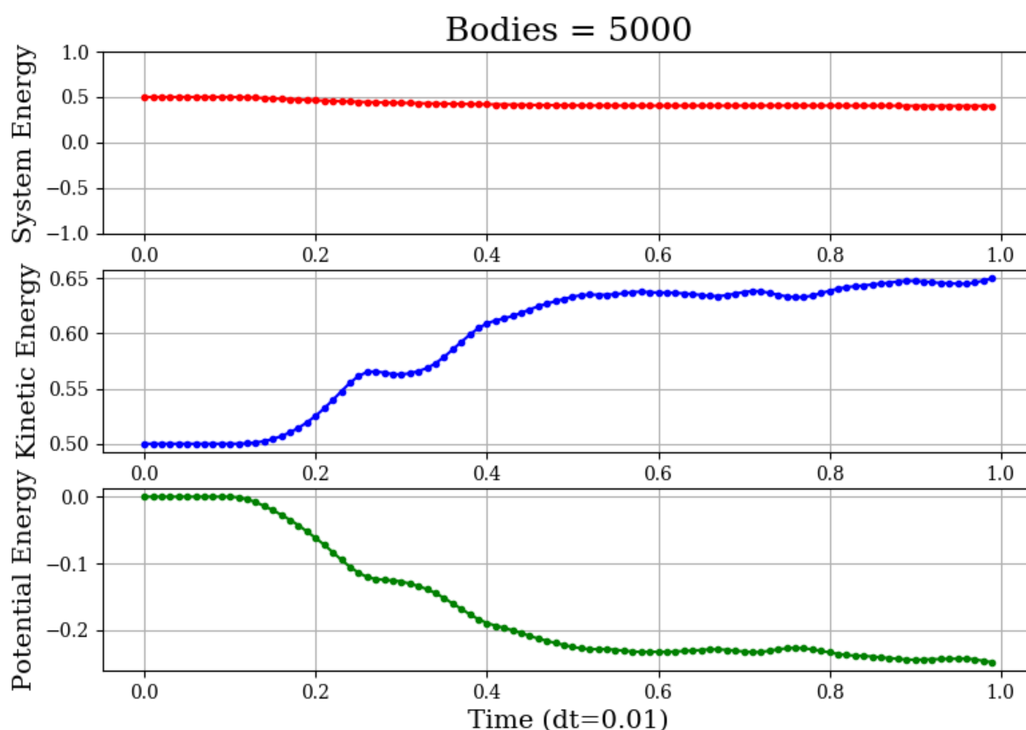

Рис. 5.22 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 5000 частинок, значення радіусу обмеження рівне 0.8 умовних одиниць

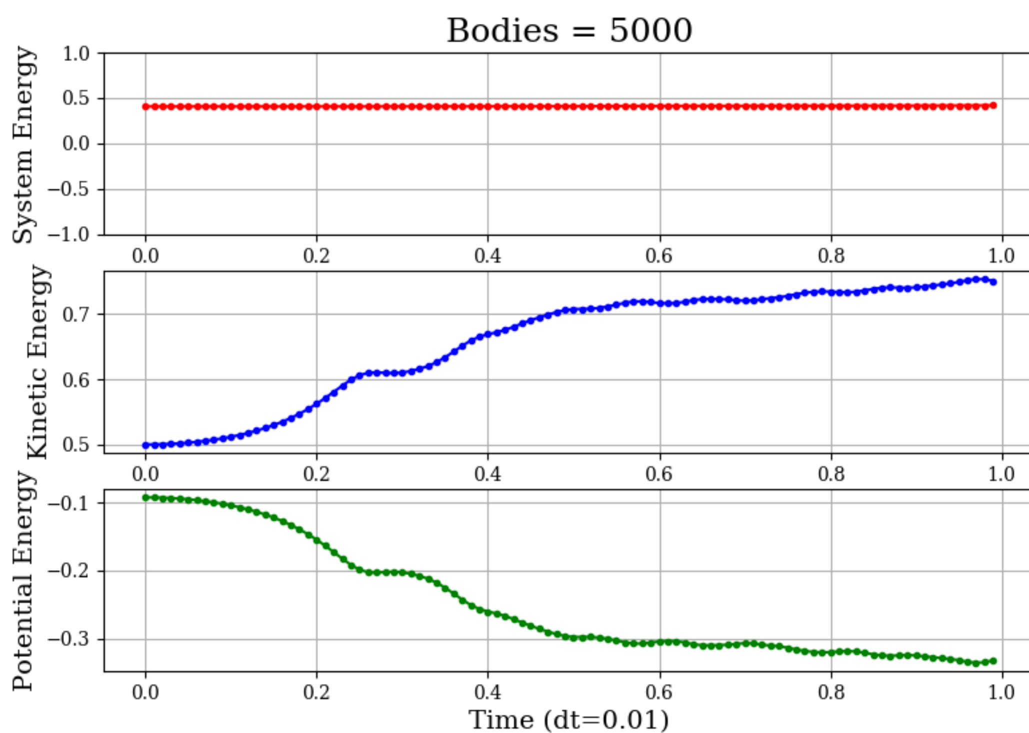

Рис. 5.23 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 5000 частинок, значення радіусу обмеження рівне 2.3 умовних одиниць

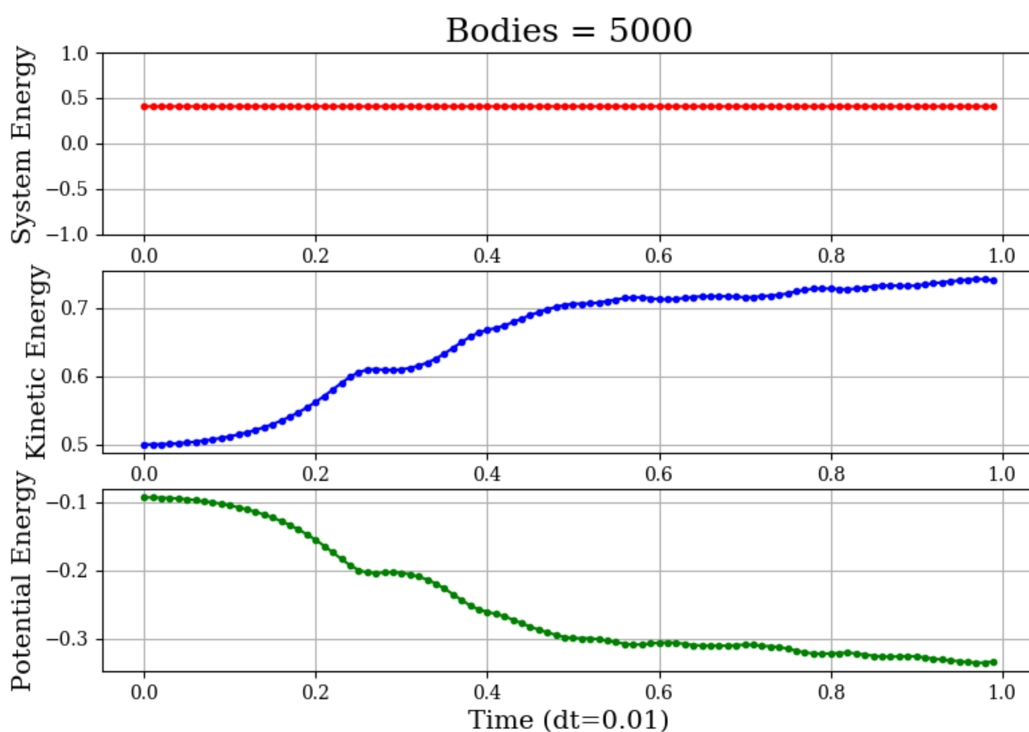

Рис. 5.24 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 5000 частинок, значення радіусу обмеження рівне 28.4 умовних одиниць

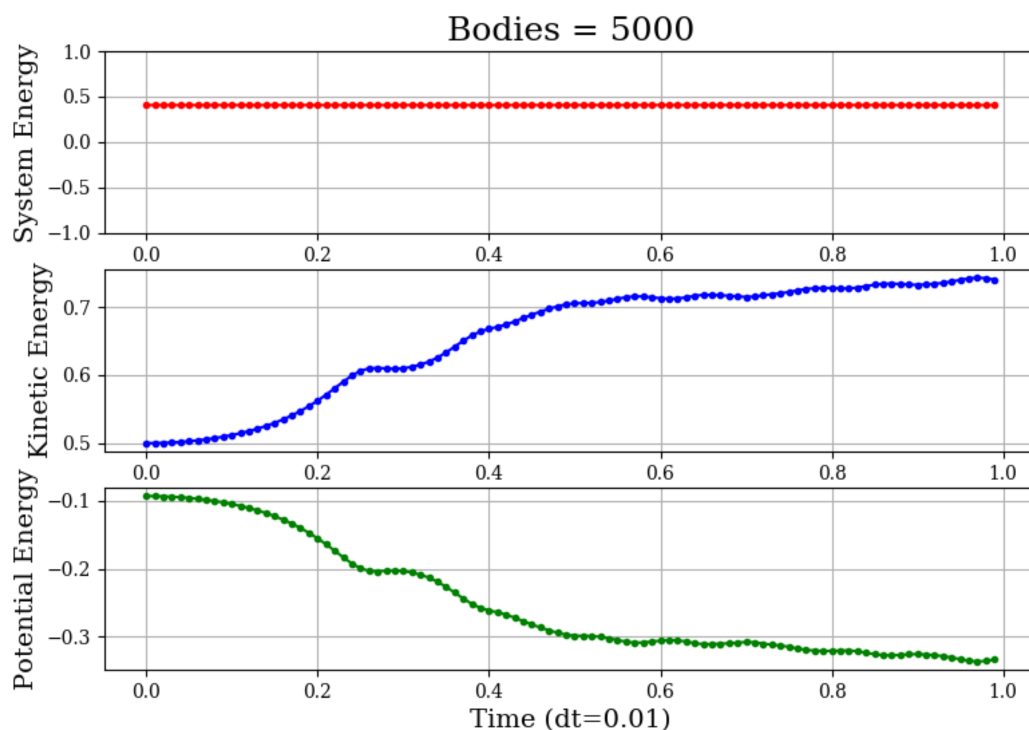

Рис. 5.25 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 5000 частинок, значення радіусу обмеження рівне 40.0 умовних одиниць

- для системи з 6000 частинок:
  - за допомогою GPU без використання радіусу

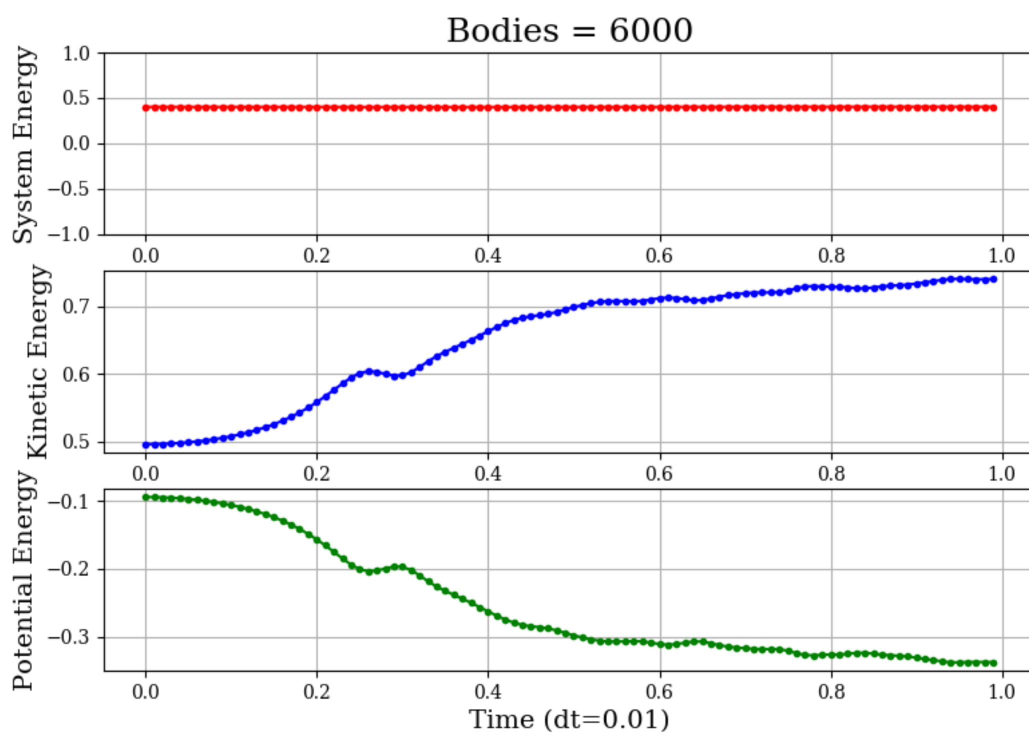

Рис. 5.26 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 6000 частинок без використання радіусу обмеження

- використовуючи радіус обмеження:

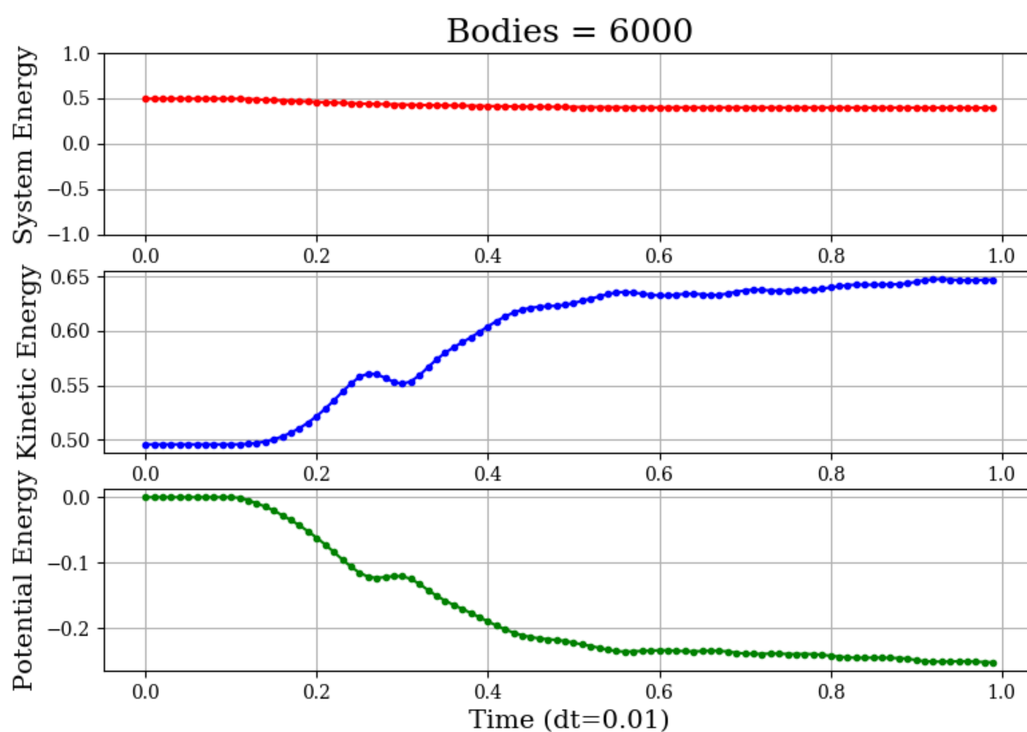

Рис. 5.27 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 6000 частинок, значення радіусу обмеження рівне 0.8 умовних одиниць

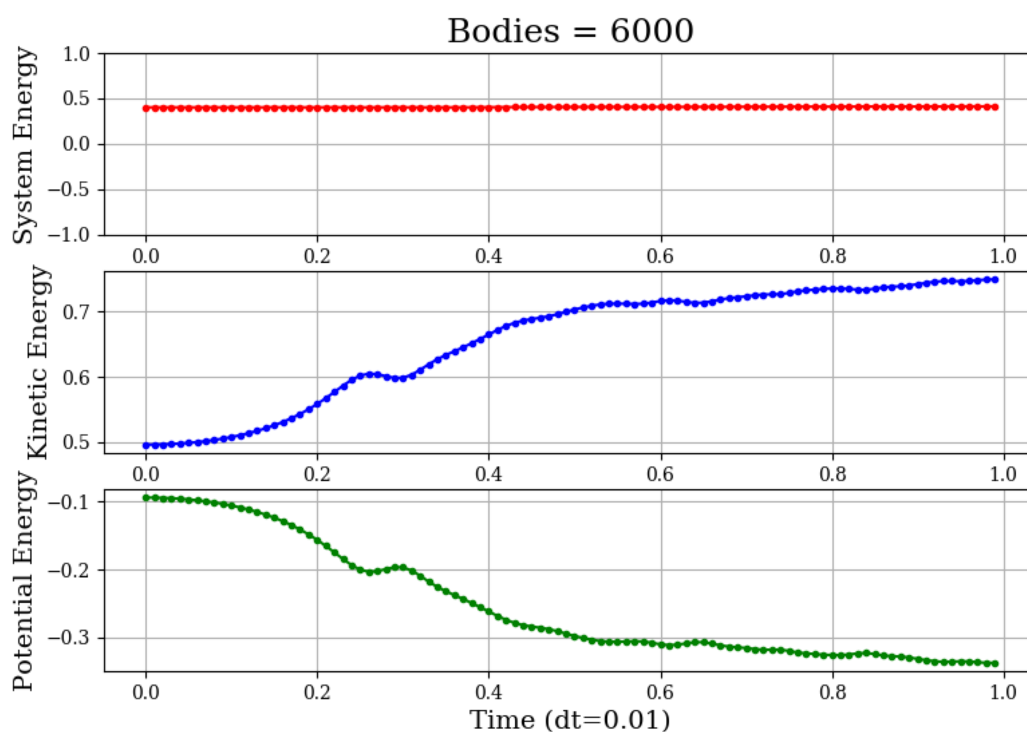

Рис. 5.28 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 6000 частинок, значення радіусу обмеження рівне 2.3 умовних одиниць

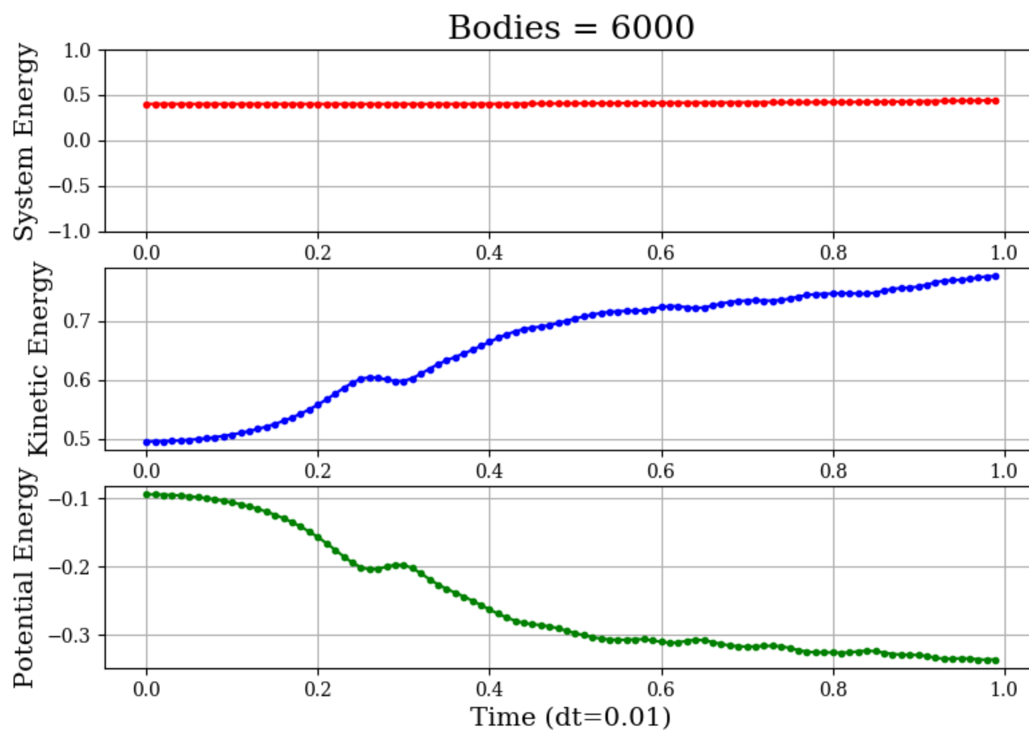

Рис. 5.29 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 6000 частинок, значення радіусу обмеження рівне 28.4 умовних одиниць

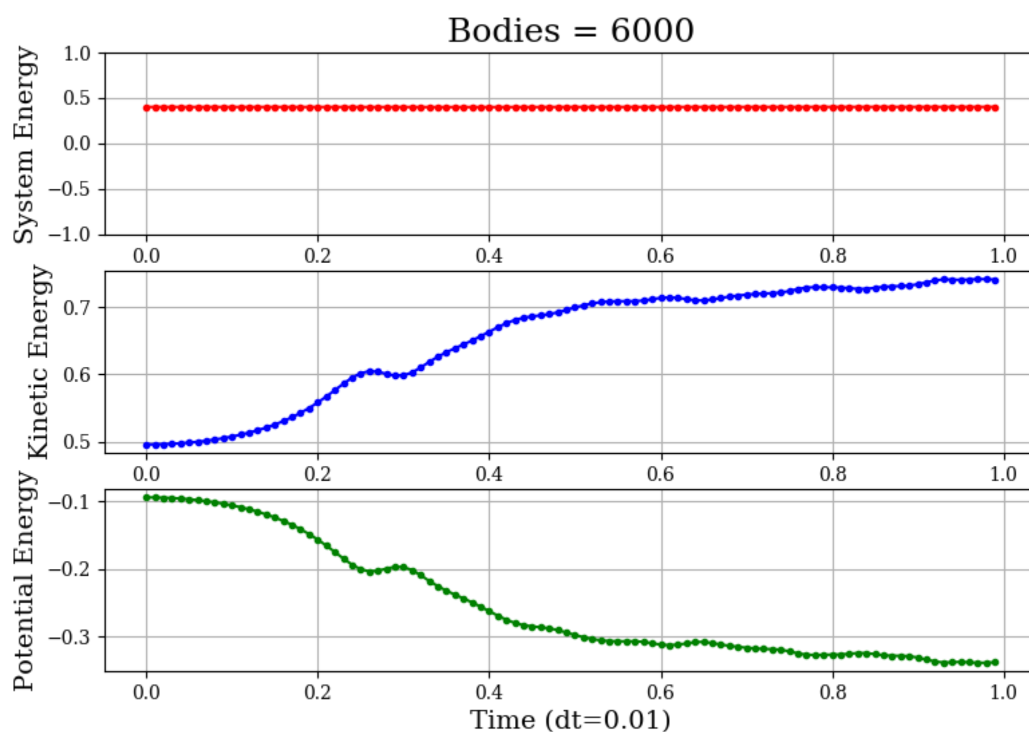

Рис. 5.30 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 6000 частинок, значення радіусу обмеження рівне 40.0 умовних одиниць

На основі отриманих значень повної, кінетичної та потенціальної енергій будувалися таблиці для перевірки існування залежності між радіусом обмеження та

кількістю частинок у системі, які використовувалися для побудови графіку залежності енергії від радіусу обмеження:

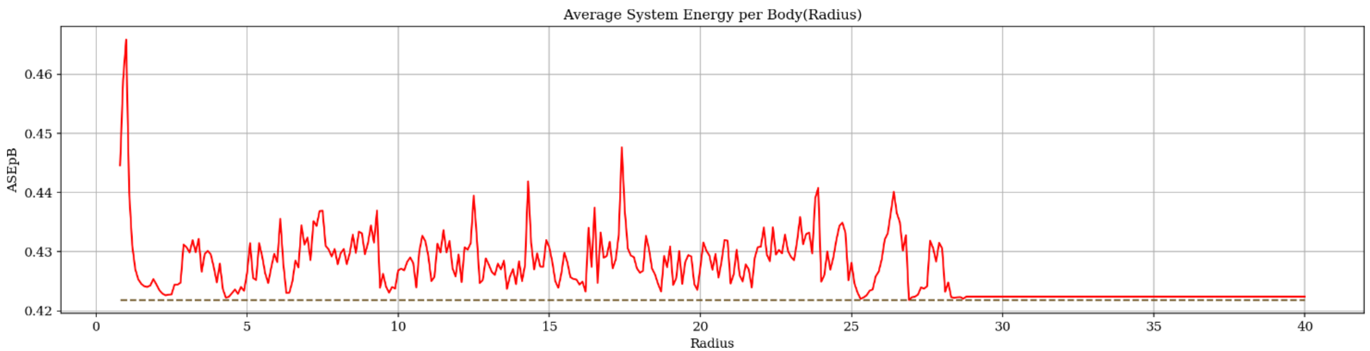

Рис. 5.31 – графік залежності середньої повної енергії на частинку (для системи з 1000 частинок): червоним позначено значення енергії відповідно для кожного значення радіусу обмеження, а пунктирною лінією – значення енергії без використання радіусу обмеження (*ASEpB, Average System Energy per Body* – середня повна енергія на частинку, що усереднена за кількістю кроків на кожне значення радіусу)

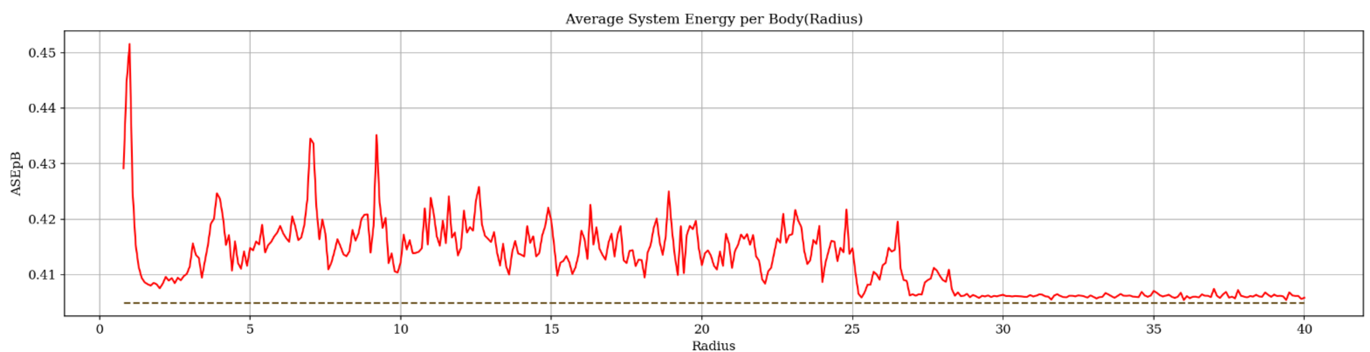

Рис. 5.32 – графік залежності середньої повної енергії на частинку (для системи з 2000 частинок): червоним позначено значення енергії відповідно для кожного значення радіусу обмеження, а пунктирною лінією – значення енергії без використання радіусу обмеження

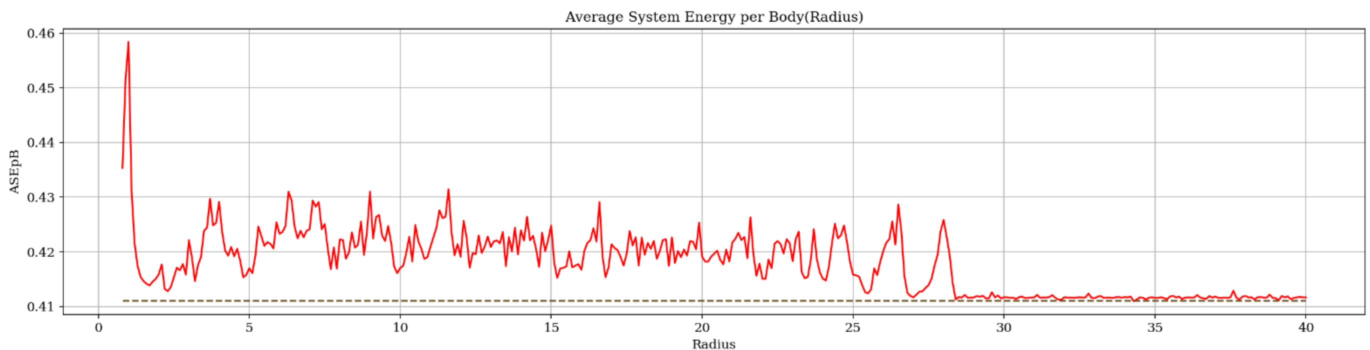

Рис. 5.33 – графік залежності середньої повної енергії на частинку (для системи з 3000 частинок): червоним позначено значення енергії відповідно для кожного значення радіусу обмеження, а пунктирною лінією – значення енергії без використання радіусу обмеження

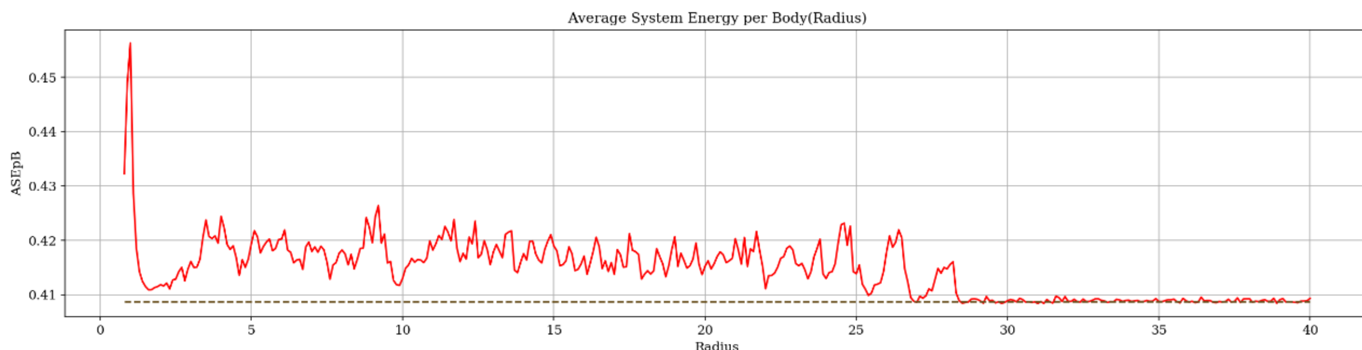

Рис. 5.34 – графік залежності середньої повної енергії на частинку (для системи з 4000 частинок): червоним позначено значення енергії відповідно для кожного значення радіусу обмеження, а пунктирною лінією – значення енергії без використання радіусу обмеження

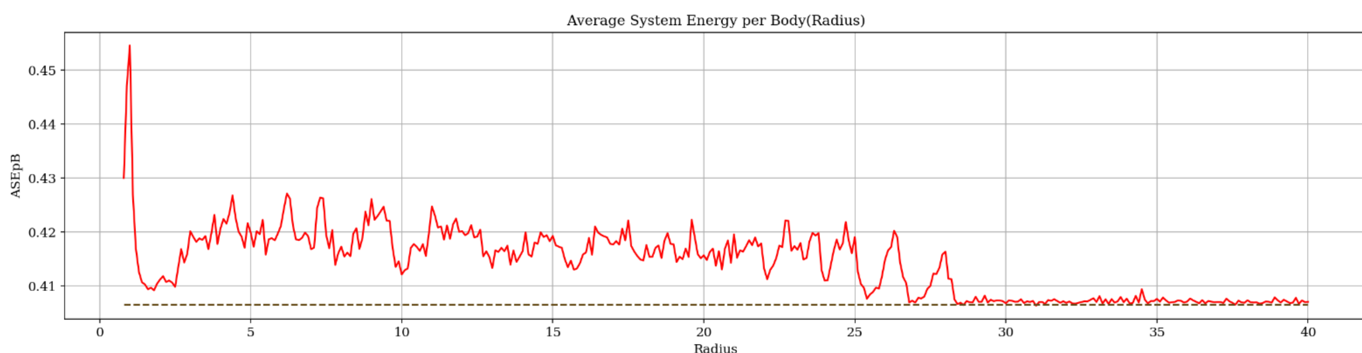

Рис. 5.35 – графік залежності середньої повної енергії на частинку (для системи з 5000 частинок): червоним позначено значення енергії відповідно для кожного значення радіусу обмеження, а пунктирною лінією – значення енергії без використання радіусу обмеження

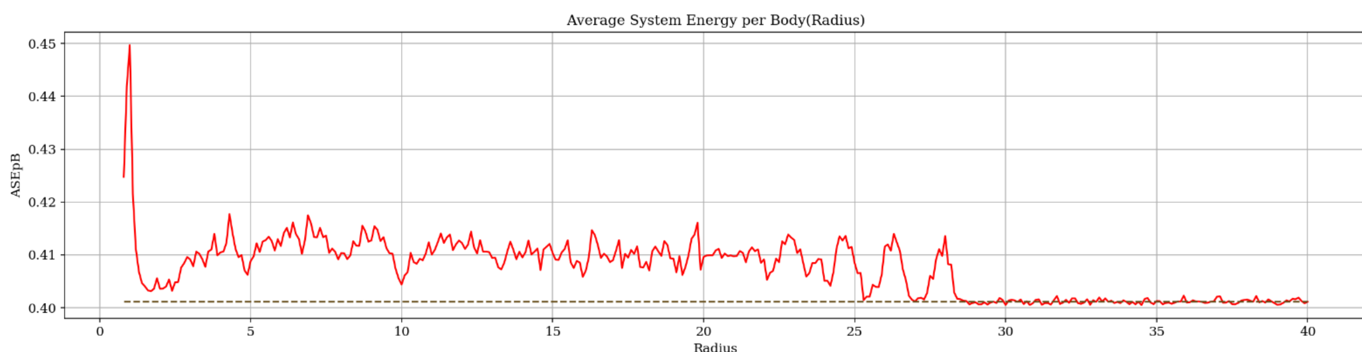

Рис. 5.36 – графік залежності середньої повної енергії на частинку (для системи з 6000 частинок): червоним позначено значення енергії відповідно для кожного значення радіусу обмеження, а пунктирною лінією – значення енергії без використання радіусу обмеження

Найбільший інтерес представляють місця, що розташовані найближче до лінії, що визначає значення енергії, яка обрахована без використання радіусу обмеження. На всіх шести графіках це місце знаходиться між значеннями 25 та 30. А на графіку для 1000 частинок ще між значеннями 0 та 5.

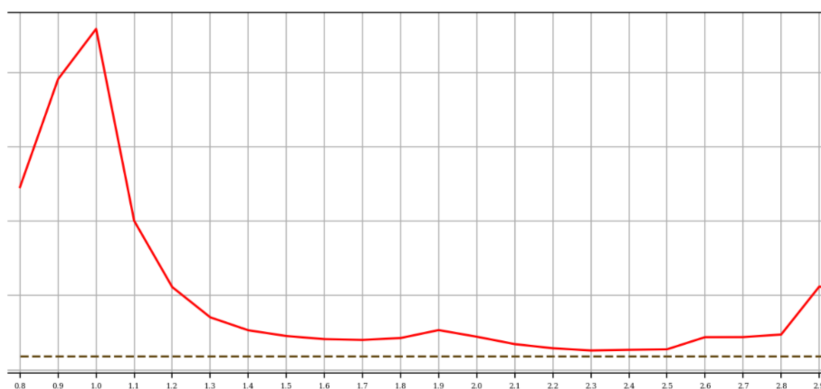

Рис. 5.37 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [0.8, 2.9] для системи з 1000 частинок)

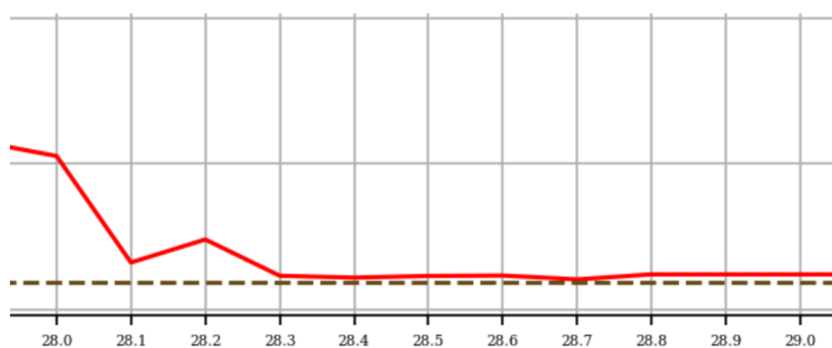

Рис. 5.38 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [28, 29] для системи з 1000 частинок)

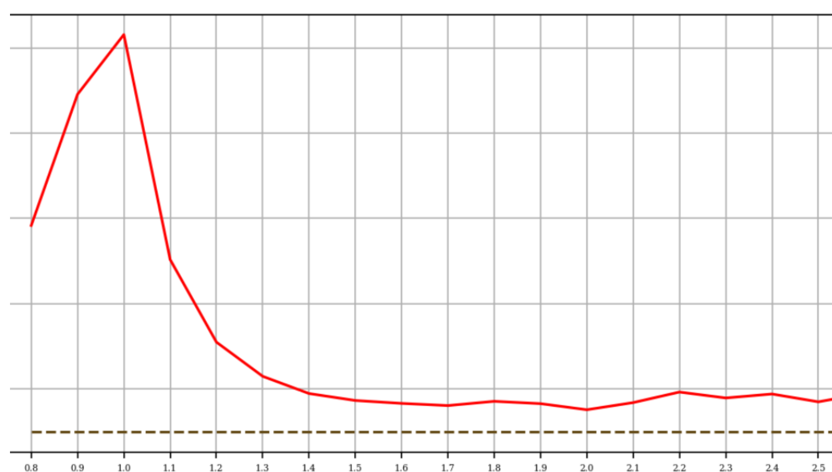

Рис. 5.39 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [0.8, 2.5] для системи з 2000 частинок)

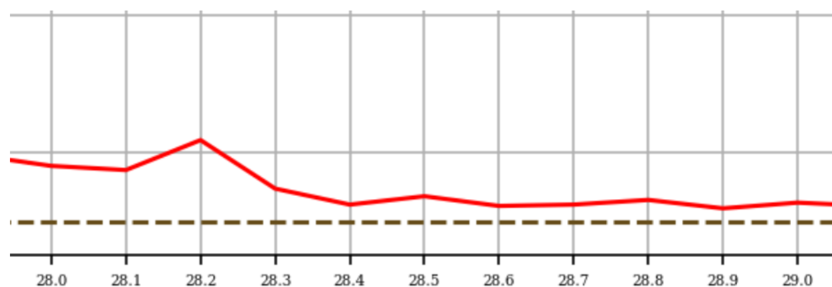

Рис. 5.40 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [28, 29] для системи з 2000 частинок)

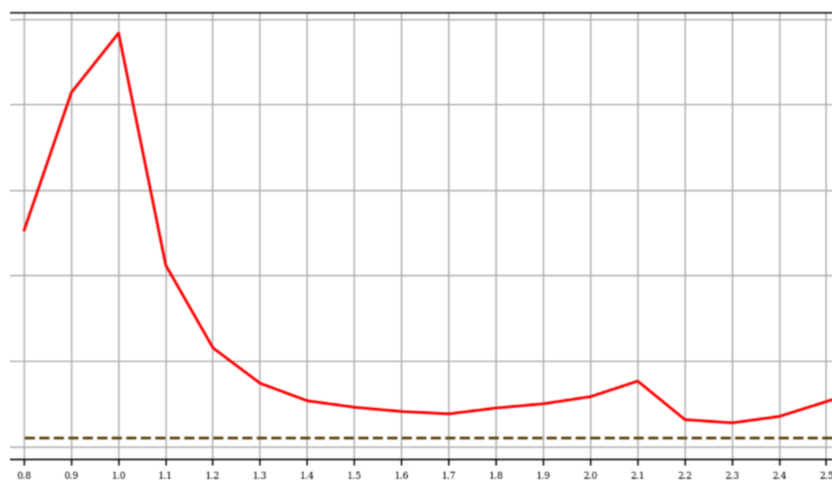

Рис. 5.41 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [0.8, 2.5] для системи з 3000 частинок)

Рис. 5.42 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [28, 29] для системи з 3000 частинок)

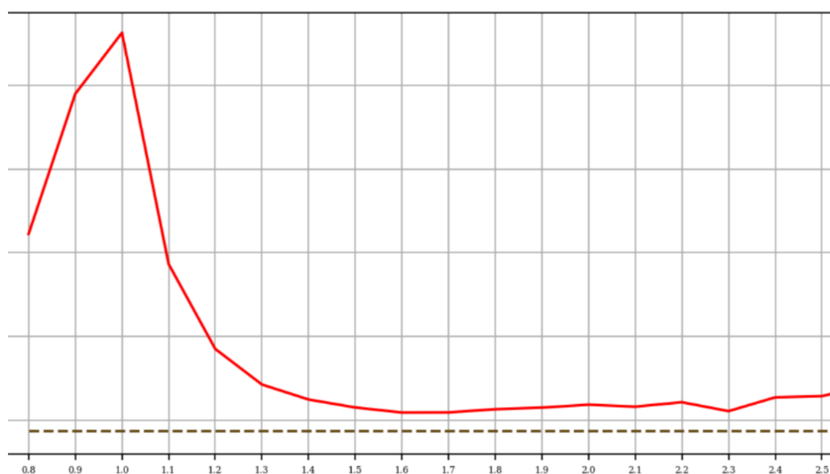

Рис. 5.43 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [0.8, 2.5] для системи з 4000 частинок)

Рис. 5.44 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [28, 29] для системи з 4000 частинок)

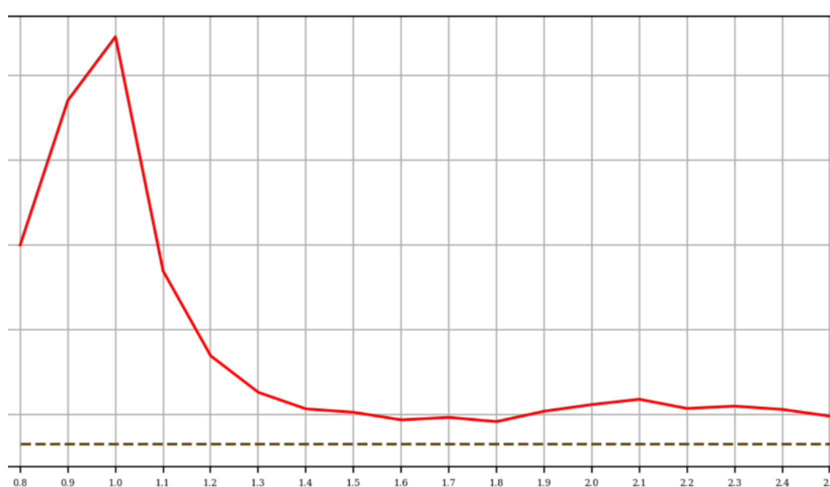

Рис. 5.45 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [0.8, 2.5] для системи з 5000 частинок)

Рис. 5.46 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [28, 29] для системи з 5000 частинок)

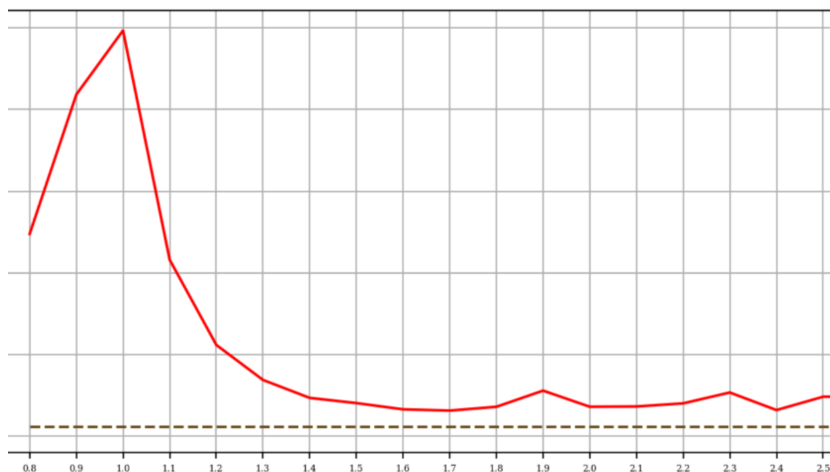

Рис. 5.47 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [0.8, 2.5] для системи з 6000 частинок)

Рис. 5.48 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [28, 29] для системи з 6000 частинок)

З графіків, що зображені на Рис. 5.37 – Рис. 5.48 можна зробити висновок, що значення радіусів обмеження 2.3 та 28.4 є найбільш підходящим для даних систем.

Експеримент 2:

- габаритні параметри контейнера $47 \times 47 \times 46$ (у частинках: 47 – ширина, 47 – висота, 46 – глибина – максимальна кількість частинок в контейнері 101614);
- системи з 1000 та 10 000 частинок.

Усі інші параметри відповідають попередньому експерименту.

У результаті отримано наступні графіки:

- для системи з 1000 частинок:

- без використання радіусу:

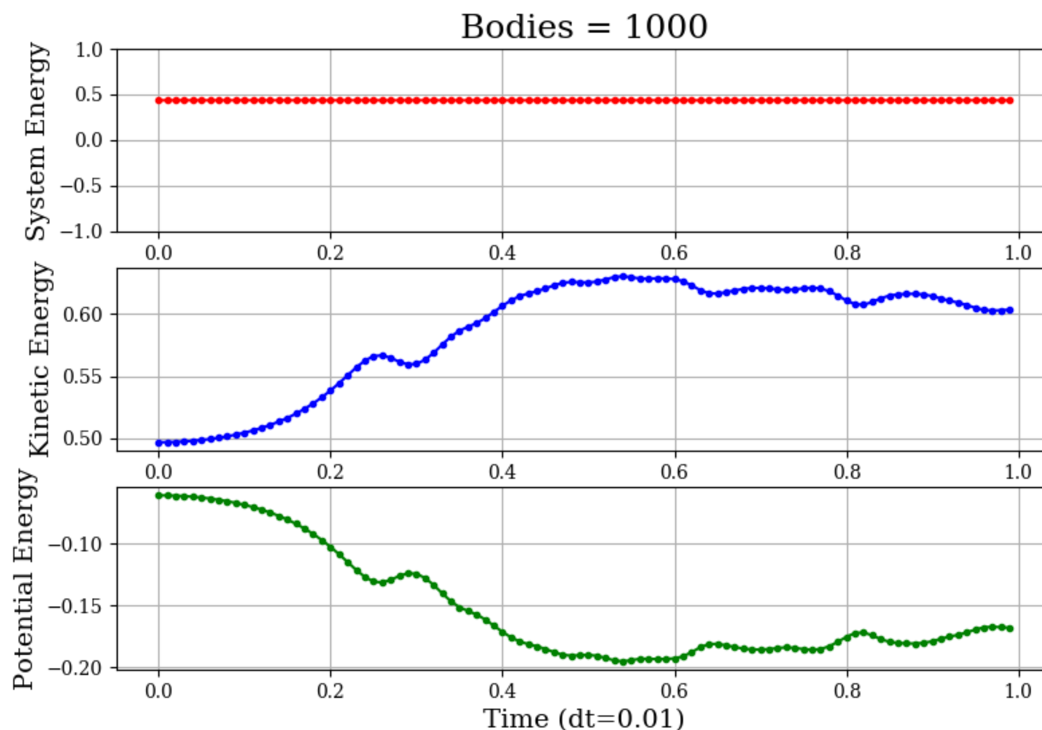

Рис. 5.49 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок без використання радіусу обмеження

- використовуючи радіус обмеження:

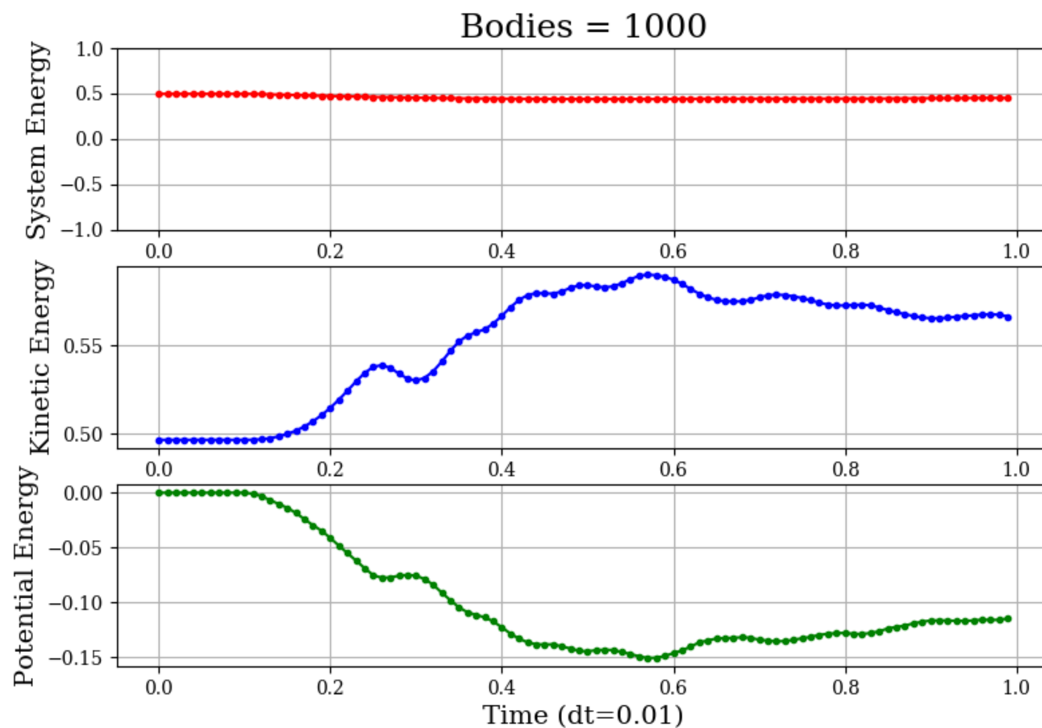

Рис. 5.50 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок, значення радіусу обмеження рівне 0.8 умовних одиниць

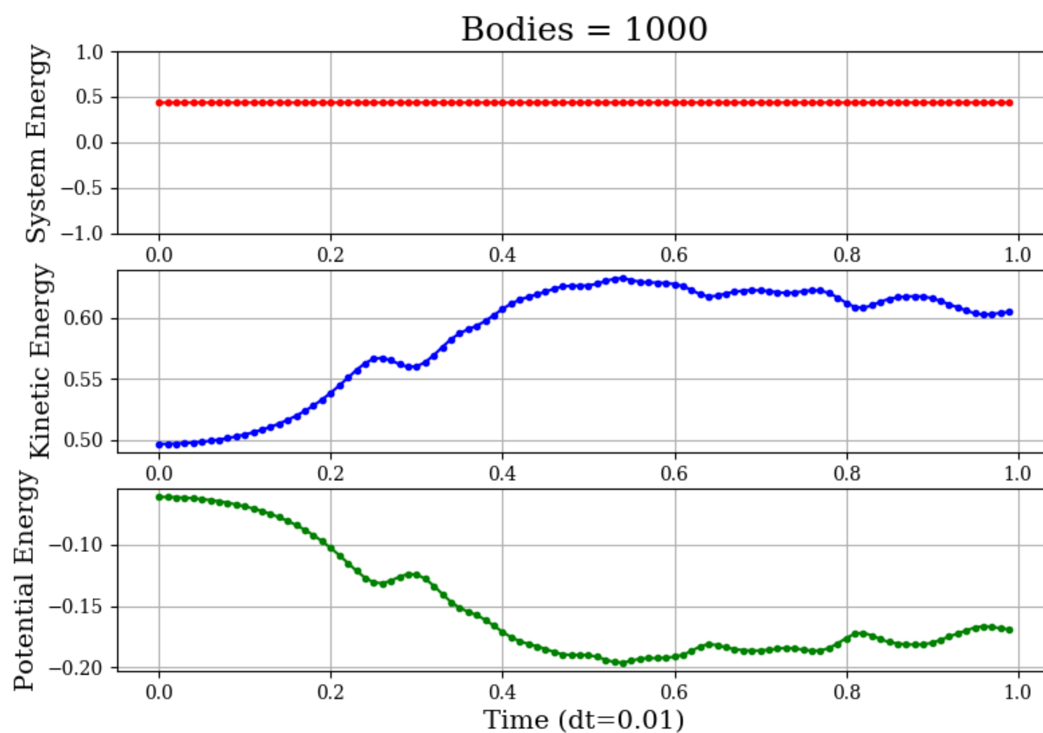

Рис. 5.51 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок, значення радіусу обмеження рівне 2.3 умовних одиниць

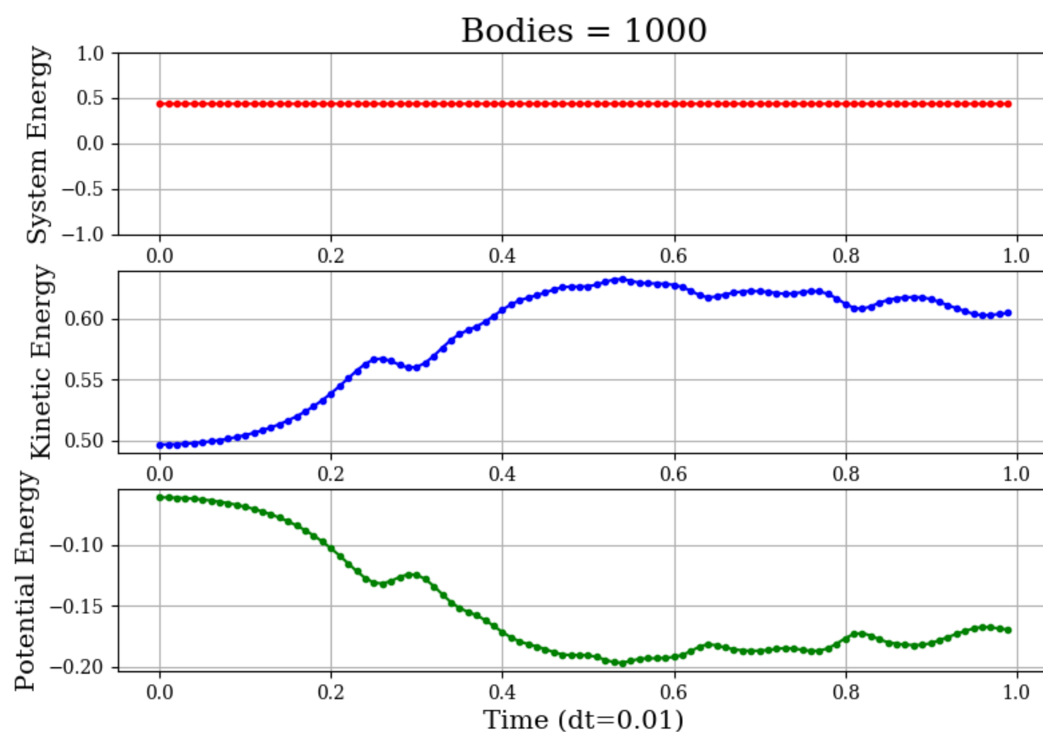

Рис. 5.52 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 1000 частинок, значення радіусу обмеження рівне 102.0 умовних одиниць

- для системи з 10 000 частинок:
  - без використання радіусу:

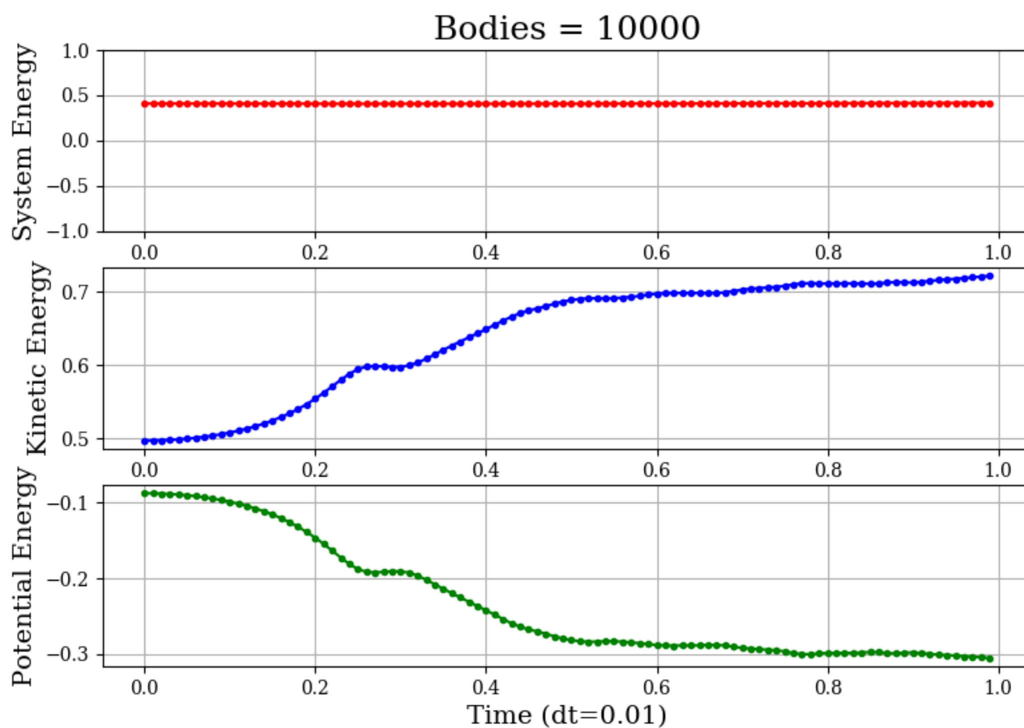

Рис. 5.53 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 10 000 частинок без використання радіусу обмеження

- використовуючи радіус обмеження:

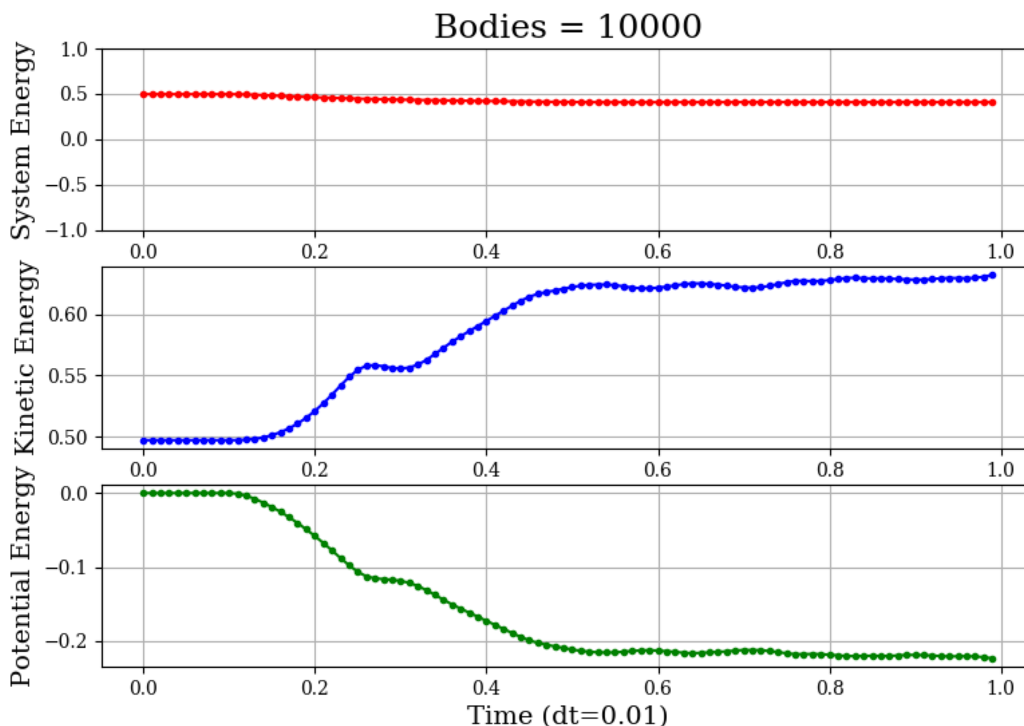

Рис. 5.54 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 10 000 частинок, значення радіусу обмеження рівне 0.8 умовних одиниць

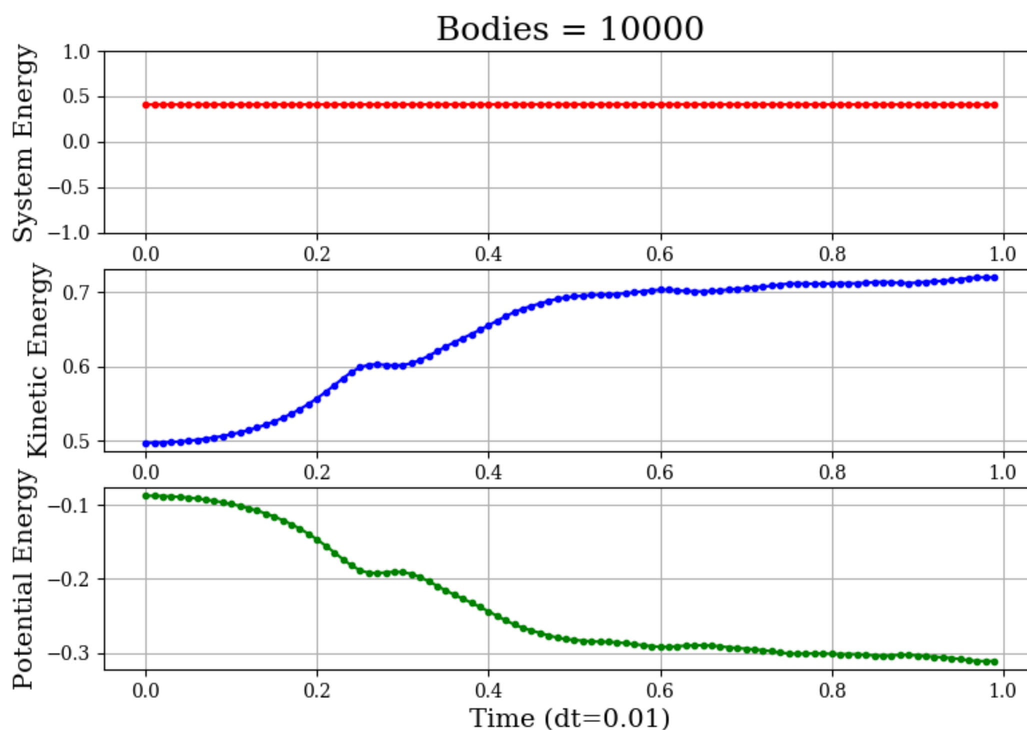

Рис. 5.55 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енерігй для системи з 10 000 частинок, значення радіусу обмеження рівне 2.3 умовних одиниць

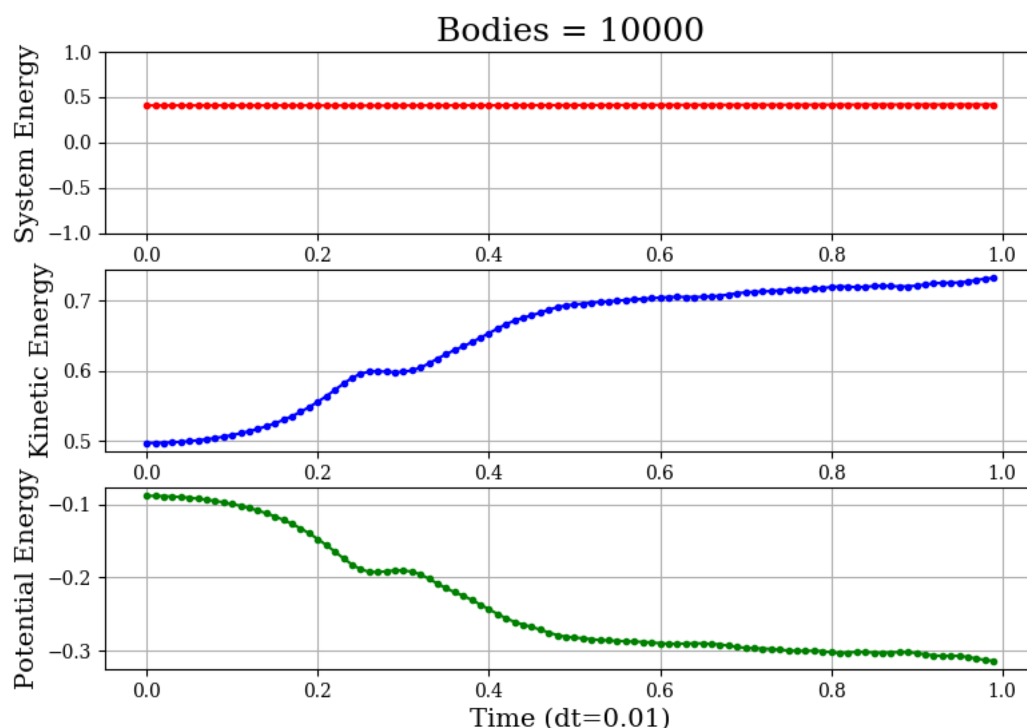

Рис. 5.56 – графік повної (червоним), кінетичної (синім), потенціальної (зеленим) енергій для системи з 10 000 частинок, значення радіусу обмеження рівне 102.0 умовних одиниць

На основі отриманих значень повної, кінетичної та потенціальної енергій будувалися таблиці для перевірки існування залежності між радіусом обмеження та

кількістю частинок у системі, які використовувалися для побудови графіку залежності енергії від радіусу обмеження:

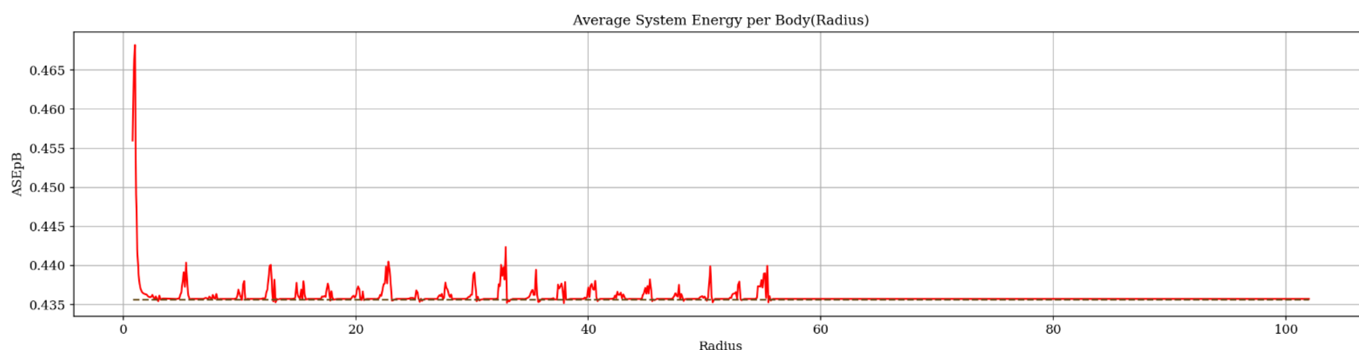

Рис. 5.57 – графік залежності середньої повної енергії на частинку (для системи з 1000 частинок): червоним позначено значення енергії відповідно для кожного значення радіусу обмеження, а пунктирною лінією – значення енергії без використання радіусу обмеження (*ASEpB, Average System Energy per Body* – середня повна енергія на частинку, що усереднена за кількістю кроків на кожне значення радіусу)

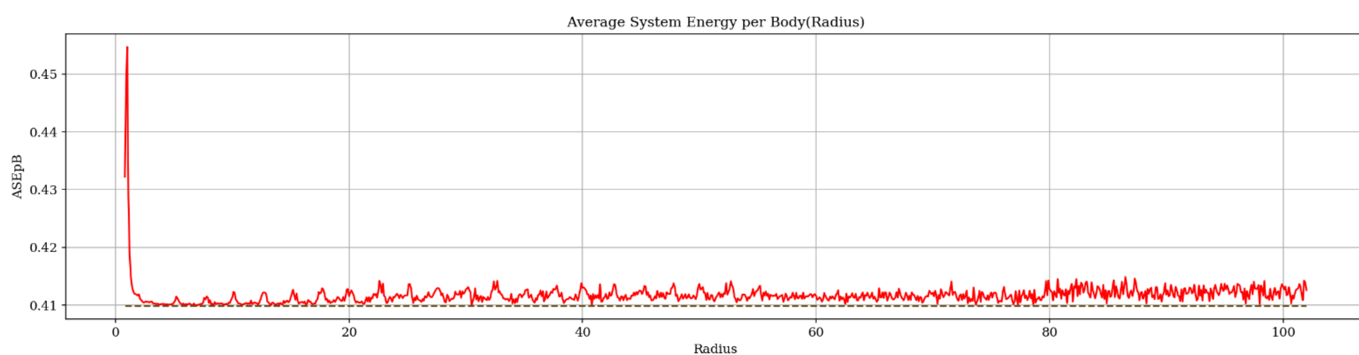

Рис. 5.58 – графік залежності середньої повної енергії на частинку (для системи з 10 000 частинок): червоним позначено значення енергії відповідно для кожного значення радіусу обмеження, а пунктирною лінією – значення енергії без використання радіусу обмеження

У даному випадку найбільш підходящі значення радіусу обмеження знаходяться у межах інтервалу [0, 5] для обох систем. За меншого кроку за радіусом отримуємо:

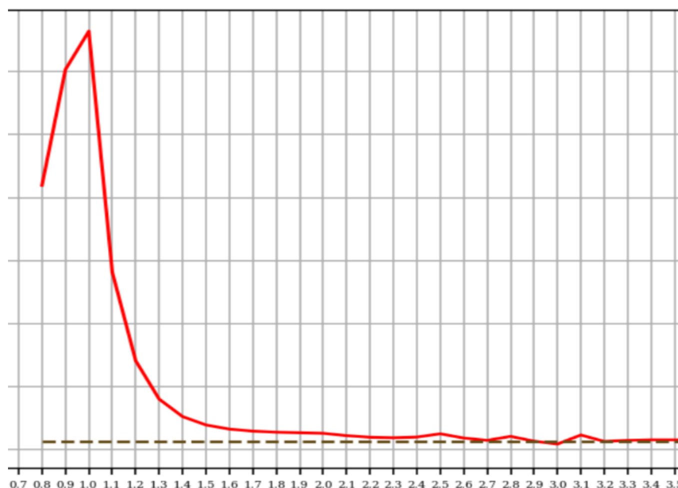

Рис. 5.59 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [0.8, 3.5] для системи з 1000 частинок)

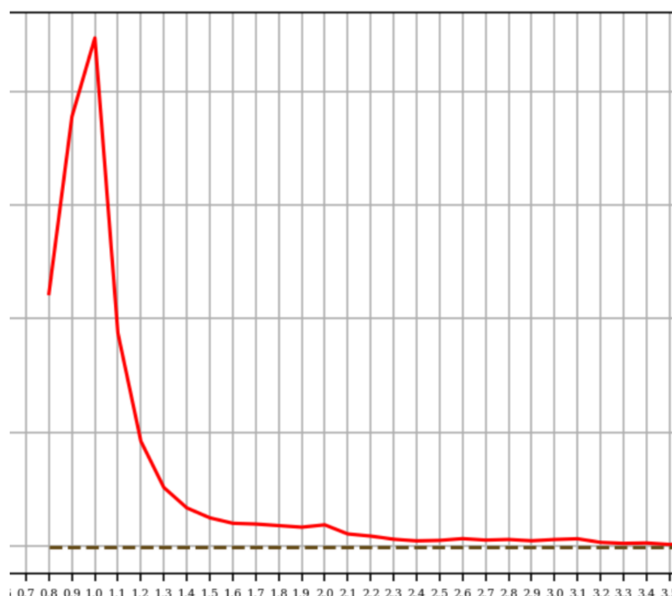

Рис. 5.60 – графік залежності середньої повної енергії на частинку (більш дрібне розбиття за радіусом проміжок [0.8, 3.5] для системи з 10 000 частинок)

З отриманих вище графіків (Рис. 5.59, Рис. 5.60) видно, що оптимальним радіусом обмеження можна вважати значення 3.3.

Загальний висновок з проведених комп'ютерних експериментів наступний: радіус обмеження більше залежить від розмірів контейнера, аніж від кількості частинок у системі. Базуючись на результатах також можна зробити висновок, що радіус обмеження краще обирати більшим за значення 4σ, у той час як літературні джерела вказують на значення 2σ, що на всіх отриманих графіках указує на найбільше відхилення від середнього значення системи без використання радіусу обмеження. Лістинг програми для побудови графіків залежності середньої повної енергії на частинку від радіусу обмеження знаходиться в розділі ДОДАТОК К.

5.5 Реалізація функції, що будує графіки порівняння швидкодії реалізацій CPU, пари CPU, GPU без використання радіусу та пари CPU, GPU з використанням радіусу обмеження

Для порівняння швидкодії розроблено функцію яка проводить комп'ютерну симуляцію за наступним алгоритмом:

1. Формується масив значень від 2 до 10 включно.
2. Перед початком формування системи зафіксовується час. По черзі перебираються значення, кожного разу формується система з параметрами контейнера (наприклад перша ітерація 2x2x2) та відповідною кількістю частинок рівною 8 (продовження попереднього прикладу). Діапазон швидкостей сталий для всіх систем: $[-1;1]$.
3. Розрахувати 1000 кроків еволюції системи. По закінченню розрахунків зафіксувати час.
4. В окремий масив зберігати різницю кінцевого й початкового часу.
5. Виконати кроки 1-4 для реалізацій окремо CPU, пари CPU, GPU без використання радіусу обмеження та CPU, GPU з радіусом обмеження.
6. Сформовані масиви часу використати для побудови графіків залежності кількості елементів від часу моделювання та відношення часу розрахунку на центральному процесорі до часу розрахунку на графічному процесорі від кількості елементів, щоб дізнатися на скільки реалізація для GPU є більш ефективною.

Функція для тестування знаходиться в основному модулі програми, що знаходиться в ДОДАТОК В. Результатом проведення тесту при значенні радіусу обмеження в 2.3 умовні одиниці (це значення взяте з результатів, що описані в попередньому розділі) є наступні графіки:

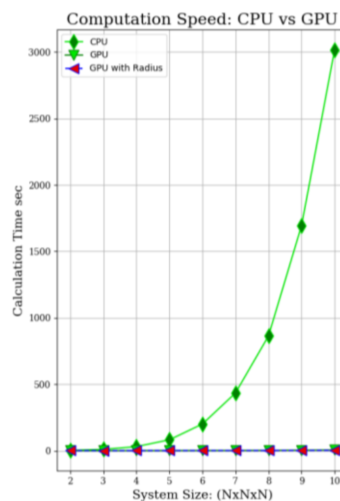

Рис. 5.61 – форма залежності часу розрахунку від кількості частинок: зелена лінія з зеленими ромбами залежності часу розрахунку на CPU, зелена лінія з зеленими

трикутниками відповідає часу розрахунку на GPU без використання радіусу обмеження, синя лінія з червоними трикутниками відповідає часу розрахунку на GPU з використанням радіусу обмеження

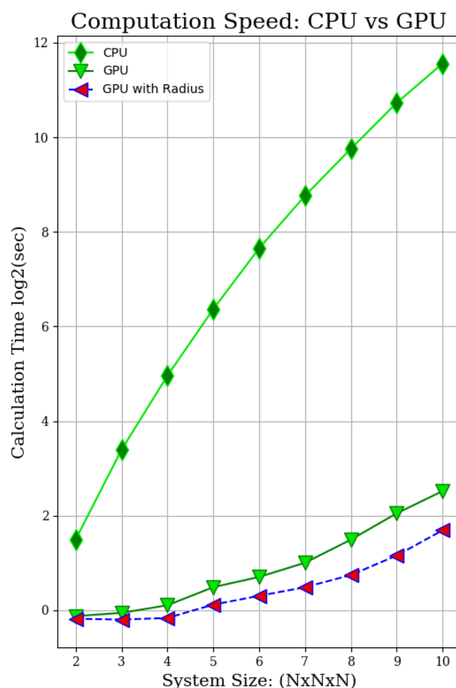

Рис. 5.62 – форма залежності часу розрахунку від кількості частинок у системі. Час логарифмовано за основою 2. Позначення відповідають попередньому графіку

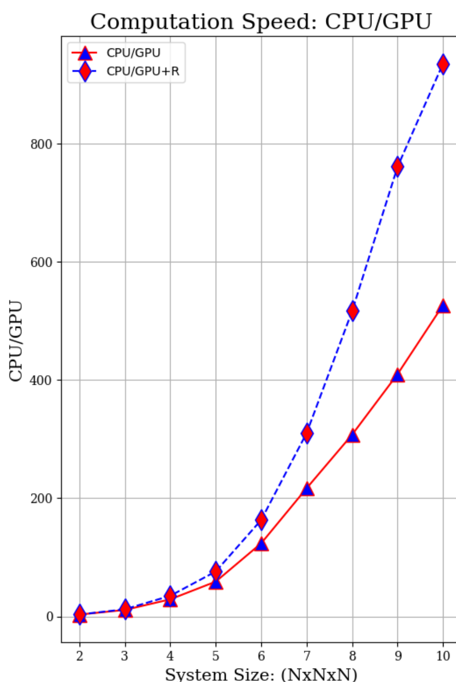

Рис. 5.63 – форма залежності відношення часу розрахунку на центральному процесорі (CPU) до часу розрахунку на графічному процесорі (GPU без радіусу обмеження – червона лінія, з радіусом обмеження – синя пунктирна лінія) від кількості частинок у системі

Судячи з отриманого результату (див. Рис. 5.63 – форма залежності відношення часу розрахунку на центральному процесорі (CPU) до часу розрахунку на графічному процесорі (GPU без радіусу обмеження – червона лінія, з радіусом обмеження – синя пунктирна лінія) від кількості частинок у системі) видно, що GPU-реалізація є порівняно швидшою. При розмірах системи у 1000 розрахунок на графічному процесорі без використання радіусу обмеження є більше ніж в 500 разів швидшим за реалізацію на центральному процесорі, а реалізація з використанням радіусу обмеження (що дорівнює 2.3 умовні одиниці) є ефективнішою за CPU версію понад 800 разів.

Результатами тесту швидкості при значенні радіусу в 28.4 (результат першого експерименту з попереднього розділу) є наступні графіки:

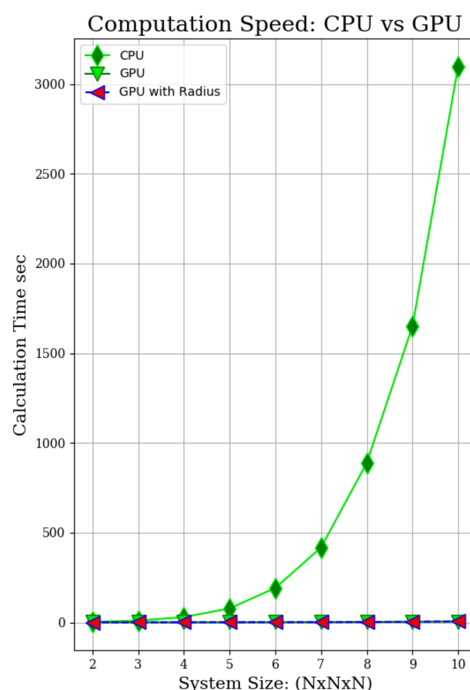

Рис. 5.64 – форма залежності часу розрахунку від кількості частинок: зелена лінія з зеленими ромбами залежності часу розрахунку на CPU, зелена лінія з зеленими трикутниками відповідає часу розрахунку на GPU без використання радіусу обмеження, синя лінія з червоними трикутниками відповідає часу розрахунку на GPU з використанням радіусу обмеження

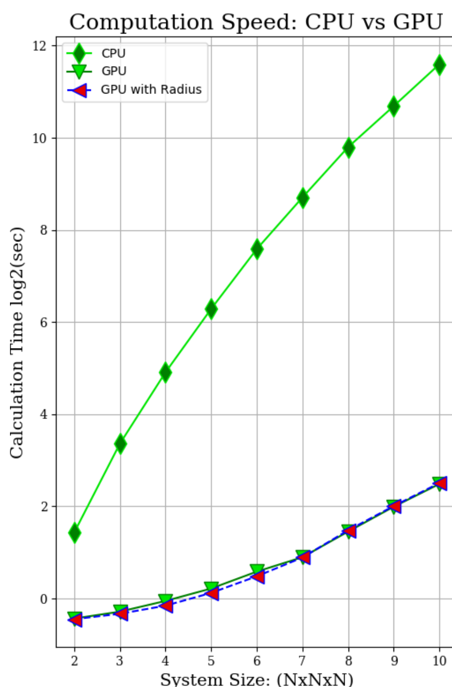

Рис. 5.65 – форма залежності часу розрахунку від кількості частинок у системі. Час логарифмовано за основою 2. Позначення відповідають попередньому графіку

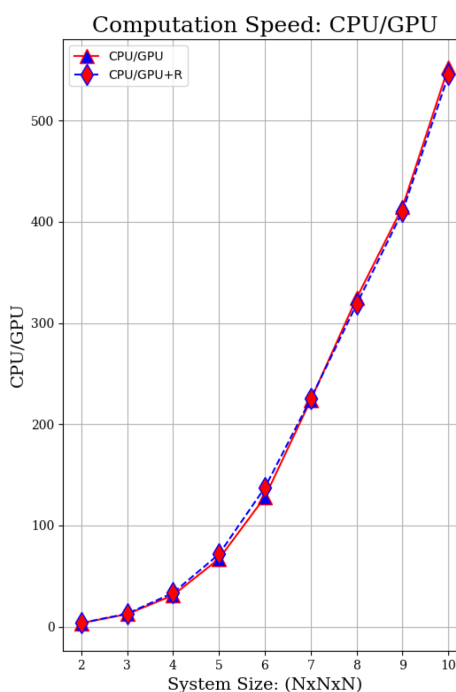

Рис. 5.66 – форма залежності відношення часу розрахунку на центральному процесорі (CPU) до часу розрахунку на графічному процесорі (GPU без радіусу обмеження – червона лінія, з радіусом обмеження – синя пунктирна лінія) від кількості частинок у системі

Результат тесту при значенні радіусу обмеження в 28.4 $(\approx 57\sigma)$ умовні одиниці говорить про те, що різниця між реалізаціями з використанням радіусу та без (з повним обрахуванням набору частинок) майже відсутня.

Загальний висновок з проведених тестів наступний: радіус обмеження дійсно є ефективним засобом прискорення розрахунків. Обирати значення слід (опираючись на тести, що були проведені) значення з діапазону $\left[4\sigma;57\sigma\right]$. Чіткої залежності між радіусом обмеження та розмірами системи виявити не вдалося, необхідні додаткові дослідження.

ВИСНОВКИ

Розроблено алгоритм визначення ефективного радіуса взаємодії у системі багатьох взаємодіючих частинок, що базується на аналізі повної енергії системи.

Виявлено, що радіус відсікання частинок, які не впливають суттєвим чином на енергію системи, несуттєво залежить від числа частинок (при N>1000). Це дозволяє провести розрахунки у два етапи. Перший етап – розрахунок радіуса відсікання для малої системи, другий розрахунок фізичних характеристик системи на основі отриманих на першому етапі даних.

Виявлено, що значення радіусу відсікання, яке пропонується в літературних джерелах (2σ, σ – рівноважна відстань) не є ефективним з точки зору збереження повної енергії. Розрахунки показують, що оптимальне значення варіюється у діапазоні від 4σ до 6σ у залежності від розмірів системи.

Запропонований підхід дозволяє скоротити час розрахунків приблизно на 50% при використанні пари CPU&GPU у порівнянні з розрахунками без уведення радіуса відсікання за нашим алгоритмом.

ПЕРЕЛІК ПОСИЛАНЬ

[1] А. М. Кривцов и Н. В. Кривцова, «Метод частиц и его использование в механике деформируемого твердого тела,» *Дальневосточный математический журнал,* т. 3, № 2, pp. 254-276, 2002.

[2] И. Ю. Зубко и Н. Д. Няшина, Математическое моделирование: дискретные подходы и численные методы, Пермь: Пермский национальный исследовательский политехнический университет, 2012.

[3] Х. Гулд и Я. Тобочник, Компьютерное моделирование в физике, т. 1.

[4] А. С. Боярченков и С. И. Поташников, «Использование графических процессоров и технологии CUDA для задач молекулярной динамики,» *Вычислительные методы и программирование,* т. 10, № 1, pp. 9-23, 2009.

[5] NVIDIA Corporation, «CUDA Toolkit Documentation,» 2007-2019. [Онлайновий]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[6] А. В. Уткин и М. С. Ожгибесов, «Применение технологий CUDA и MPI к решению задач молекулярной динамики,» pp. 127-129, 2014.

[7] В. Л. Малышев, Д. Ф. Марьин, Е. Ф. Моисеева, Н. А. Гумеров и И. Ш. Ахатов, «Ускорение молекулярно-динамического моделирования неполярных молекул при помощи GPU,» т. 1, № 3, pp. 126-133, 2014.

ДОДАТОК А

Лістинг програми побудови графіків залежності сили та потенціалу від відстані між частинками

```
# -*- coding: utf-8 -*-
from matplotlib import pyplot as plt
from numpy import array, zeros, float64, arange, apply_along_axis


def prepare_plot(autor_name, letter, xlabel, ylabel,
                 lylim, rylim, fwidth=9, fheight=6):
    fig, subplot = plt.subplots(nrows=1, ncols=1)

    fig.set_figwidth(fwidth)
    fig.set_figheight(fheight)
    # Plot title
    subplot.set_title(
        label=autor_name+" "+letter.upper()+"("+xlabel[0].lower()+")",
        fontdict={"fontsize": 18,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # X axis properties
    subplot.set_ylim(lylim, rylim)
    plt.setp(subplot.get_xticklabels(), "fontsize", "14")
    plt.setp(subplot.get_xticklabels(), "fontname", "Times New Roman")
    plt.setp(subplot.get_xticklabels(), "family", "serif")
    subplot.set_xlabel(
        xlabel=xlabel,
        fontdict={"fontsize": 16,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # Y axis properties
    plt.setp(subplot.get_yticklabels(), "fontsize", "14")
    plt.setp(subplot.get_yticklabels(), "fontname", "Times New Roman")
    plt.setp(subplot.get_yticklabels(), "family", "serif")
    subplot.set_ylabel(
        ylabel=ylabel,
        fontdict={"fontsize": 16,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    subplot.grid()
    return fig, subplot


def calculate_lennard_jones_potential(brange, epsilon, sigma):
    return 4*epsilon*((sigma/brange)**12 - (sigma/brange)**6)


def calculate_lennard_jones_force(brange, epsilon, sigma):
    return 24*(epsilon/sigma)*(2*(sigma/brange)**13 - (sigma/brange)**7)


def calculate_mie_potential(brange, epsilon, sigma, m, n):
    return (epsilon/(n - m))*(m*(sigma/brange)**n - n*(sigma/brange)**m)


def calculate_mie_force(brange, epsilon, sigma, m, n):
    return (n*m/(n-m))*(epsilon/sigma)*((sigma/brange)**(n + 1) - (sigma/brange)**(m + 1))


# Preparing constants
epsilon = float64(0.5)
sigma = float64(0.5)
m = float64(2)
n = float64(1)
first_range_value = 0.1
range_step = 0.01
last_range_value = 3*epsilon + range_step
# LENNARD-JONES POTENTIAL
# Preparing arrays
ranges = arange(first_range_value,
                last_range_value,
                range_step,
```

```
                dtype=float64)
lennard_jones_potentials = apply_along_axis(
    func1d=lambda r: calculate_lennard_jones_potential(
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
# Draw Plot
autor_name = "Lennard-Jones"
fig_lennard_jones_potential,\
    subplot_lennard_jones_potential = prepare_plot(
        autor_name=autor_name,
        letter="U",
        xlabel="Range",
        ylabel="Potential",
        lylim=-0.6,
        rylim=0.6)
subplot_lennard_jones_potential.plot(
    ranges,
    lennard_jones_potentials,
    marker="",
    color="blue")
# Saving Plot
fig_lennard_jones_potential.savefig(
    autor_name +
    subplot_lennard_jones_potential.get_ylabel() + ".png")
# LENNARD JONES FORCE
# Preparing arrays
ranges = arange(first_range_value,
                last_range_value,
                range_step,
                dtype=float64)
lennard_jones_forces = apply_along_axis(
    func1d=lambda r: calculate_lennard_jones_force(
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
# Draw Plot
autor_name = "Lennard-Jones"
fig_lennard_jones_force, subplot_lennard_jones_force = prepare_plot(
    autor_name=autor_name,
    letter="F",
    xlabel="Range",
    ylabel="Force",
    lylim=-3,
    rylim=3)
subplot_lennard_jones_force.plot(
    ranges,
    lennard_jones_forces,
    marker="",
    color="red")
# Saving Plot
fig_lennard_jones_force.savefig(
    autor_name + \
    subplot_lennard_jones_force.get_ylabel() + ".png")
# LENNARD-JONES FORCE&POTENTIAL
# Preparing arrays
ranges = arange(first_range_value,
                last_range_value,
                range_step,
                dtype=float64)
lennard_jones_forces = apply_along_axis(
    func1d=lambda r: calculate_lennard_jones_force(
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
lennard_jones_potentials = apply_along_axis(
    func1d=lambda r: calculate_lennard_jones_potential(
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
# Draw Plot
autor_name = "Lennard-Jones"
fig_lennard_jones, subplot_lennard_jones = prepare_plot(
    autor_name=autor_name,
    letter="U&F",
```

```
    xlabel="Range",
    ylabel="Values",
    lylim=-3,
    rylim=3)
subplot_lennard_jones.plot(
    ranges,
    lennard_jones_forces,
    marker="",
    color="red",
    label="Forse")
subplot_lennard_jones.plot(
    ranges,
    lennard_jones_potentials,
    marker="",
    color="blue",
    label="Potential")
subplot_lennard_jones.legend()
# Saving Plot
fig_lennard_jones.savefig(
    autor_name +
    subplot_lennard_jones.get_ylabel() + ".png")
# MIE POTENTIAL
# Preparing arrays
ranges = arange(first_range_value,
                last_range_value,
                range_step,
                dtype=float64)
mie_potentials = apply_along_axis(
    func1d=lambda r: calculate_mie_potential(
        brange=r,
        epsilon=epsilon,
        sigma=sigma,
        m=m,
        n=n),
    axis=0, arr=ranges)
# Draw Plot
autor_name = "Mie"
fig_mie_potential, subplot_mie_potential = prepare_plot(
    autor_name=autor_name,
    letter="U",
    xlabel="Range",
    ylabel="Potential",
    lylim=-0.6,
    rylim=0.6)
subplot_mie_potential.plot(
    ranges,
    mie_potentials,
    marker="",
    color="magenta")
# Saving Plot
fig_mie_potential.savefig(
    autor_name +
    subplot_mie_potential.get_ylabel() + ".png")
# MIE FORSE
# Preparing arrays
ranges = arange(first_range_value,
                last_range_value,
                range_step,
                dtype=float64)
mie_forces = apply_along_axis(
    func1d=lambda r: calculate_mie_force(
        brange=r,
        epsilon=epsilon,
        sigma=sigma,
        m=m,
        n=n),
    axis=0, arr=ranges)
# Draw Plot
autor_name = "Mie"
fig_mie_force, subplot_mie_force = prepare_plot(
    autor_name=autor_name,
    letter="F",
    xlabel="Range",
    ylabel="Force",
    lylim=-0.6,
    rylim=0.6)
subplot_mie_force.plot(
    ranges,
    mie_forces,
```

```
    marker="",
    color="green")
# Saving Plot
fig_mie_force.savefig(
    autor_name +
    subplot_mie_force.get_ylabel() + ".png")


# Модифікація потенціалу та сили
def calculate_modify_lennard_jones_potential(k, brange, epsilon, sigma):
    """k - modify potential; k < 1 - to stretch; k > 1 - to narrow"""
    return 4*epsilon*((sigma/(k*(brange - sigma) + sigma))**12 - (sigma/(k*(brange - sigma) + sigma))**6)


def calculate_modify_lennard_jones_force(k, brange, epsilon, sigma):
    """k - modify potential; k < 1 - to stretch; k > 1 - to narrow"""
    return k*24*(epsilon/sigma)*(2*(sigma/(k*(brange - sigma) + sigma))**13 - (sigma/(k*(brange - sigma) +
sigma))**7)


# MODIFY LENNARD-JONES POTENTIALS
# Preparing arrays
ranges = arange(first_range_value,
                last_range_value,
                range_step,
                dtype=float64)
lennard_jones_potentials = apply_along_axis(
    func1d=lambda r: calculate_lennard_jones_potential(
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
lennard_jones_potentials_narrow = apply_along_axis(
    func1d=lambda r: calculate_modify_lennard_jones_potential(
        k=float64(1.5),
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
lennard_jones_potentials_stretch = apply_along_axis(
    func1d=lambda r: calculate_modify_lennard_jones_potential(
        k=float64(0.5),
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
# Prepare Axes
autor_name = "Lennard-Jones Modify"
fig_lennard_jones_potential, subplot_lennard_jones_potential = prepare_plot(
    autor_name=autor_name,
    letter="U",
    xlabel="Range",
    ylabel="Potential",
    lylim=-0.6,
    rylim=0.2)
# Draw Normal Potential Plot
subplot_lennard_jones_potential.plot(
    ranges,
    lennard_jones_potentials,
    marker="",
    color="blue",
    label="Normal: k=1")
# Draw Modify Potential Plot NARROW
subplot_lennard_jones_potential.plot(
    ranges,
    lennard_jones_potentials_narrow,
    marker="",
    color="green",
    label="Narrow: k>1")
# Draw Modify Potential Plot STRETCH
subplot_lennard_jones_potential.plot(
    ranges,
    lennard_jones_potentials_stretch,
    marker="",
    color="red",
    label="Stretch: k<1")
# Show Legend
subplot_lennard_jones_potential.legend()
# Saving Plot
```

```
fig_lennard_jones_potential.savefig(
    autor_name +
    subplot_lennard_jones_potential.get_ylabel() + ".png")

# MODIFY LENNARD-JONES FORCE
# Preparing arrays
ranges = arange(0.55, 1.5, 0.01, dtype=float64)
lennard_jones_forces = apply_along_axis(
    func1d=lambda r: calculate_lennard_jones_force(
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
lennard_jones_forces_narrow = apply_along_axis(
    func1d=lambda r: calculate_modify_lennard_jones_force(
        k=float64(1.5),
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
lennard_jones_forces_stretch = apply_along_axis(
    func1d=lambda r: calculate_modify_lennard_jones_force(
        k=float64(0.5),
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
# Prepare Axes
autor_name = "Lennard-Jones Modify"
fig_lennard_jones_force, subplot_lennard_jones_force = prepare_plot(
    autor_name=autor_name,
    letter="k*F",
    xlabel="Range",
    ylabel="Force",
    lylim=-4.0,
    rylim=1)
# Draw Normal Force Plot
subplot_lennard_jones_force.plot(
    ranges,
    lennard_jones_forces,
    marker="",
    color="blue",
    label="Normal: k=1")
# Draw Modify Force Plot NARROW
subplot_lennard_jones_force.plot(
    ranges,
    lennard_jones_forces_narrow,
    marker="",
    color="green",
    label="Narrow: k>1")
# Draw Modify Force Plot STRETCH
subplot_lennard_jones_force.plot(
    ranges,
    lennard_jones_forces_stretch,
    marker="",
    color="red",
    label="Stretch: k<1")
# Show Legend
subplot_lennard_jones_force.legend()
# Saving Plot
fig_lennard_jones_force.savefig(
    autor_name +
    subplot_lennard_jones_force.get_ylabel() + ".png")


# MODIFY LENNARD-JONES FORCE
def calculate_modify_lennard_jones_force_02(k, brange, epsilon, sigma):
    """k - modify potential; k < 1 - to stretch; k > 1 - to narrow"""
    return 24*(epsilon/sigma)*(2*(sigma/(k*(brange - sigma) + sigma))**13 - (sigma/(k*(brange - sigma) +
sigma))**7)


# Preparing arrays
ranges = arange(0.55, 1.5, 0.01, dtype=float64)
lennard_jones_forces = apply_along_axis(
    func1d=lambda r: calculate_lennard_jones_force(
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
```

```
    axis=0, arr=ranges)
lennard_jones_forces_narrow = apply_along_axis(
    func1d=lambda r: calculate_modify_lennard_jones_force_02(
        k=float64(1.5),
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
lennard_jones_forces_stretch = apply_along_axis(
    func1d=lambda r: calculate_modify_lennard_jones_force_02(
        k=float64(0.5),
        brange=r,
        epsilon=epsilon,
        sigma=sigma),
    axis=0, arr=ranges)
# Prepare Axes
autor_name = "Lennard-Jones Modify"
fig_lennard_jones_force, subplot_lennard_jones_force = prepare_plot(
    autor_name=autor_name,
    letter="F",
    xlabel="Range",
    ylabel="Force",
    lylim=-2.5,
    rylim=1)
# Draw Normal Force Plot
subplot_lennard_jones_force.plot(
    ranges,
    lennard_jones_forces,
    marker="",
    color="blue",
    label="Normal: k=1")
# Draw Modify Force Plot NARROW
subplot_lennard_jones_force.plot(
    ranges,
    lennard_jones_forces_narrow,
    marker="",
    color="green",
    label="Narrow: k>1")
# Draw Modify Force Plot STRETCH
subplot_lennard_jones_force.plot(
    ranges,
    lennard_jones_forces_stretch,
    marker="",
    color="red",
    label="Stretch: k<1")
# Show Legend
subplot_lennard_jones_force.legend()
# Saving Plot
fig_lennard_jones_force.savefig(
    autor_name +
    subplot_lennard_jones_force.get_ylabel() + "_without_K.png")
```

ДОДАТОК Б

Лістинг програми моделювання системи частинок

```
# -*- coding: utf-8 -*-
from GUI.main_window import MolecularDynamic
from panda3d.core import load_prc_file_data


if __name__ == "__main__":
    load_prc_file_data("", "win-size 1024 768")
    application = MolecularDynamic()
    application.run()
```

ДОДАТОК В

Лістинг основного модуля *main_window* програми моделювання системи частинок

```
# -*- coding: utf-8 -*-
from direct.showbase.ShowBase import ShowBase
from direct.task import Task
from direct.gui.DirectGui import OnscreenText, DirectButton, DirectEntry
from direct.interval.IntervalGlobal import Sequence, Func

from panda3d.core import TextNode

from numpy import float32, int32, array, concatenate, ceil, zeros, random, log2, arange
from copy import deepcopy
from time import localtime
from timeit import default_timer as timer
from matplotlib import pyplot as plt
from time import time
import os
import shutil

from pycuda import autoinit
from pycuda import driver as cuda

from MSD_functionality.CPU_1Ker.calc_coordinates_cpu_1ker import calc_coordinates_cpu_1ker
from MSD_functionality.CPU_1Ker.calc_speeds_cpu_1ker import calc_speeds_cpu_1ker
from MSD_functionality.CPU_1Ker.calc_accelerate_cpu_1ker import calc_accelerate_v01
from MSD_functionality.CPU_1Ker.calc_system_energy import calc_kinetic_energy

from MSD_functionality.GPU.vectors.GPU_calc_coords import gpu_calc_coordinates_kernel
from MSD_functionality.GPU.vectors.GPU_calc_speeds import gpu_calc_speeds_kernel
from MSD_functionality.GPU.vectors.GPU_calc_accelerates import gpu_calc_accelerates_kernel
from MSD_functionality.GPU.vectors.GPU_calc_accelerates_with_rcut import
gpu_calc_accelerates_with_r_cut_kernel
from MSD_functionality.GPU.vectors.GPU_calc_kinetic_energy import gpu_calc_kinetic_energy
from MSD_functionality.GPU.vectors.GPU_calc_full_time_cycle_without_r import
gpu_calc_full_time_cycle_without_r_kernel

from MSD_functionality.GPU.matrix.GPU_calc_coordinates_mtx import gpu_calc_coordinates_mtx_kernel
from MSD_functionality.GPU.matrix.GPU_calc_speeds_mtx import gpu_calc_speeds_mtx_kernel
from MSD_functionality.GPU.matrix.GPU_calc_accelerates_mtx import gpu_calc_accelerates_mtx_kernel
from MSD_functionality.GPU.matrix.GPU_calc_kinetic_energy_mtx import gpu_calc_kinetic_energy_mtx_kernel

from Addition_Scripts.table_with_radiuses import create_radius_table


class MolecularDynamic(ShowBase):
    def __init__(self):
        ShowBase.__init__(self)

        # TASK CLEANER PARAMETER
        # --------------------------------------------------
        self.task_name_turn_off = None
        # --------------------------------------------------

        # CUDA PARAMETERS
        # --------------------------------------------------
        self.tpb = None
        self.bpg = None
        self.block = None
        self.grid = None
        # --------------------------------------------------

        # PLOT PARAMETERS
        # --------------------------------------------------
        self.fig_plot = None
        self.subplots = None
        # --------------------------------------------------

        # MOLECULAR DYNAMIC SYSTEM PARAMETERS
        # -------------------------
        self.dimensions = int32(3)
        self.iter_counter = 0
        self.iter_window = 100
        self.dt_accumulator = float32(0.0)
        self.dt = float32(0.001)  # 0.005
```

```
self.sigma = float32(0.5)
self.epsilon = float32(0.5)

self.box_width = None
self.box_height = None
self.box_depth = None
self.bodies_amount = None
self.bodies_start_speed_range = None
self.bodies = None

# BODIES OUTPUT PARAMETERS
# --------------------------------------------------
self.distance_between_bodies = float32(1.5)
self.distance_between_bodies_output =\
    float32(2*self.sigma + self.distance_between_bodies)
self.texture_for_body = self.loader.loadTexture("maps/circle.png")
self.path_2_object_for_body = "models/smiley"
# --------------------------------------------------

# SPEED TEST
# --------------------------------------------------
self.experiments_amount = None
self.cpu_computation_time = None
self.gpu_computation_time = None
self.gpu_r_cut_computation_time = None
# --------------------------------------------------

# CREATING TABLE with CUTTING RADIUS PARAMETERS
# --------------------------------------------------
self.global_bodies_coordinates = None
self.global_bodies_speeds = None

self.current_cutting_radius = None
self.cutting_radius_step = None
self.first_cutting_radius = None
self.last_cutting_radius = None
self.cutting_radius_list = None

self.experiment_average_energy_list = None
self.reference_average_energy = None
self.experiment_average_kinetic_energy_list = None
self.reference_average_kinetic_energy = None
self.experiment_average_potential_energy_list = None
self.reference_average_potential_energy = None

self.dt_list = None
self.system_energy_list = None
self.kinetic_energy_list = None
self.potential_energy_list = None

self.amount_bodies_in_system = [5_000, 6_000]
self.parameters_sets =\
    [(19, 18, 18) for _ in range(len(self.amount_bodies_in_system))]
self.box_parameters =\
    dict(tuple(map(
        lambda size, parameters: (size, parameters),
        self.amount_bodies_in_system,
        self.parameters_sets)))

self.box_width_index = 0
self.box_height_index = 1
self.box_depth_index = 2
self.positive_flag = "+"
self.negative_flag = "-"

self.experiment_exit_threshold = float32(0.01)
self.interrupted = False
self.steps_limit = 1_000  # trigger
# --------------------------------------------------

# CPU ARRAYS
self.host_bodies_coordinates = None
self.host_bodies_speeds = None
self.host_bodies_accelerates = None

self.bodies_in_row = None
self.bodies_in_col = None
self.bodies_in_dep = None
```

```
# GPU ARRAYS
self.gpu_coordinates = None
self.gpu_speeds = None
self.gpu_accelerates = None

self.host_coordinates_x = None
self.host_coordinates_y = None
self.host_coordinates_z = None
self.gpu_coordinates_x = None
self.gpu_coordinates_y = None
self.gpu_coordinates_z = None

self.host_speeds_x = None
self.host_speeds_y = None
self.host_speeds_z = None
self.gpu_speeds_x = None
self.gpu_speeds_y = None
self.gpu_speeds_z = None

self.host_accelerates_x = None
self.host_accelerates_y = None
self.host_accelerates_z = None
self.gpu_accelerates_x = None
self.gpu_accelerates_y = None
self.gpu_accelerates_z = None

self.host_kinetic_energy = None
self.gpu_kinetic_energy = None

self.host_potential_energy = None
self.gpu_potential_energy = None

# GPU functions
self.calc_coordinates_gpu = None
self.calc_speeds_gpu = None
self.calc_accelerates_gpu = None
self.calc_kinetic_energy_gpu = None
# ------------------------

# LABEL PARAMETERS
# -------------------------------------------------
self.label_scale = 0.1
self.label_x_coordinate = -1.3
self.label_y_coordinate = 0.5
self.label_y_step = 0.11
# -------------------------------------------------

# ENTRY PARAMETERS
# -------------------------------------------------
self.entry_scale = 0.065
self.entry_width = 5
self.entry_x_coordinate = self.label_x_coordinate + 1
self.entry_y_coordinate = 0.0
self.entry_z_coordinate = self.label_y_coordinate
self.entry_z_step = self.label_y_step
# -------------------------------------------------

# BUTTON PARAMETERS
# -------------------------------------------------
self.button_scale = 0.1
self.button_x_coordinate = self.entry_x_coordinate + 0.21
self.button_y_coordinate = self.entry_y_coordinate
self.button_z_coordinate = self.entry_z_coordinate - 0.6
# -------------------------------------------------

# TIME LABEL
# -------------------------------------------------
self.onscreen_timer = None
self.onscreen_timer_label = "Simulation Time"
self.start_time_label = "Start Time"
self.start_time_value = None
# -------------------------------------------------

# DATA FIELDS
# -------------------------------------------------
self.label_amount_bodies =\
    OnscreenText(text="Number of Bodies",
                 scale=self.label_scale,
                 pos=(self.label_x_coordinate,
```

```
                     self.label_y_coordinate),
                align=TextNode.ALeft)
self.label_y_coordinate -= self.label_y_step
self.entry_amount_bodies =\
    DirectEntry(scale=self.entry_scale,
               pos=(self.entry_x_coordinate,
                    self.entry_y_coordinate,
                    self.entry_z_coordinate),
               width=self.entry_width)
self.entry_z_coordinate -= self.entry_z_step

self.label_speed_range =\
    OnscreenText(text="Start Speeds Range",
                scale=self.label_scale,
                pos=(self.label_x_coordinate,
                     self.label_y_coordinate),
                align=TextNode.ALeft)
self.label_y_coordinate -= self.label_y_step
self.entry_speed_range =\
    DirectEntry(scale=self.entry_scale,
               pos=(self.entry_x_coordinate,
                    self.entry_y_coordinate,
                    self.entry_z_coordinate),
               width=self.entry_width)
self.entry_z_coordinate -= self.entry_z_step

self.label_bodies_in_row =\
    OnscreenText(text="Bodies in row",
                scale=self.label_scale,
                pos=(self.label_x_coordinate,
                     self.label_y_coordinate),
                align=TextNode.ALeft)
self.label_y_coordinate -= self.label_y_step
self.entry_bodies_in_row =\
    DirectEntry(scale=self.entry_scale,
               pos=(self.entry_x_coordinate,
                    self.entry_y_coordinate,
                    self.entry_z_coordinate),
               width=self.entry_width)
self.entry_z_coordinate -= self.entry_z_step

self.label_bodies_in_col =\
    OnscreenText(text="Bodies in col",
                scale=self.label_scale,
                pos=(self.label_x_coordinate,
                     self.label_y_coordinate),
                align=TextNode.ALeft)
self.label_y_coordinate -= self.label_y_step
self.entry_bodies_in_col =\
    DirectEntry(scale=self.entry_scale,
               pos=(self.entry_x_coordinate,
                    self.entry_y_coordinate,
                    self.entry_z_coordinate),
               width=self.entry_width)
self.entry_z_coordinate -= self.entry_z_step

self.label_bodies_in_dep =\
    OnscreenText(text="Bodies in dep",
                scale=self.label_scale,
                pos=(self.label_x_coordinate,
                     self.label_y_coordinate),
                align=TextNode.ALeft)
self.entry_bodies_in_dep =\
    DirectEntry(scale=self.entry_scale,
               pos=(self.entry_x_coordinate,
                    self.entry_y_coordinate,
                    self.entry_z_coordinate),
               width=self.entry_width)
self.entry_z_coordinate -= self.entry_z_step
# --------------------------------------------------

# BUTTONS
# --------------------------------------------------
# PREPARING MOLECULAR DYNAMIC SYSTEM WITH OUTPUT
self.button_read_data_with_output =\
    DirectButton(text=("Read with output",
                       "Read with output",
                       "Read with output",
                       ""),
```

```
                 scale=self.button_scale,
                 pos=(self.button_x_coordinate,
                      self.button_y_coordinate,
                      self.button_z_coordinate),
                 command=self.initialise_mds_with_output)
self.button_z_coordinate -= self.entry_z_step + 0.025

# PREPARING MOLECULAR DYNAMIC SYSTEM WITHOUT OUTPUT
self.button_read_data_without_output =\
    DirectButton(text=("Read without output",
                       "Read without output",
                       "Read without output"),
                 scale=self.button_scale,
                 pos=(self.button_x_coordinate,
                      self.button_y_coordinate,
                      self.button_z_coordinate),
                 command=self.initialise_mds_without_output)
self.button_z_coordinate -= self.entry_z_step + 0.025

# START EXPERIMENT CPU WITH OUTPUT
self.button_start_cpu_1_ker_experiment_with_output =\
    DirectButton(text=("CPU", "CPU", "CPU", ""),
                 scale=self.button_scale,
                 pos=(-0.95, 0.0, -0.95),
                 command=self.start_cpu_1_kernel_experiment_with_output)
self.button_start_cpu_1_ker_experiment_with_output.hide()

# START EXPERIMENT CPU WITHOUT OUTPUT
self.button_start_cpu_1_ker_experiment_without_output =\
    DirectButton(text=("CPU", "CPU", "CPU2", ""),
                 scale=self.button_scale,
                 pos=(-0.95, 0.0, -0.95),
                 command=self.start_cpu_1_kernel_experiment_without_output)
self.button_start_cpu_1_ker_experiment_without_output.hide()

# START EXPERIMENT GPU WITH OUTPUT
self.button_start_gpu_experiment_with_output =\
    DirectButton(text=("GPU", "GPU", "GPU", ""),
                 scale=self.button_scale,
                 pos=(0.95, 0.0, -0.95),
                 command=self.start_gpu_experiment_with_output)
self.button_start_gpu_experiment_with_output.hide()

# START EXPERIMENT GPU WITHOUT OUTPUT
self.button_start_gpu_experiment_without_output =\
    DirectButton(text=("GPU", "GPU", "GPU2", ""),
                 scale=self.button_scale,
                 pos=(0.95, 0.0, -0.95),
                 command=self.start_gpu_experiment_without_output)
self.button_start_gpu_experiment_without_output.hide()

# SAVE SYSTEM ENERGY PLOT
self.button_save_system_energy_plot =\
    DirectButton(text=("Save SysEnergy Plot",
                       "Save SysEnergy Plot",
                       "Save SysEnergy Plot",
                       "Save SysEnergy Plot"),
                 scale=self.button_scale,
                 pos=(0.85, 0.0, -0.95),
                 command=self.save_plot)
self.button_save_system_energy_plot.hide()

# COMPARE SPEED: GPU and CPU
self.button_compare_cpu_gpu = \
    DirectButton(text=("GPU vs CPU",
                       "GPU vs CPU",
                       "GPU vs CPU",
                       "GPU vs CPU"),
                 scale=self.button_scale,
                 pos=(self.button_x_coordinate,
                      self.button_y_coordinate,
                      self.button_z_coordinate),
                 command=self.compare_cpu_and_gpu)
self.button_z_coordinate -= self.entry_z_step

# CREATE TABLE with RADIUS
self.button_create_table_with_radius =\
    DirectButton(text=("Create table with Radius",
                       "Create table with Radius",
```

```
                            "Create table with Radius",
                            "Create table with Radius"),
                        scale=self.button_scale,
                        pos=(self.button_x_coordinate,
                             self.button_y_coordinate,
                             self.button_z_coordinate),
                        command=self.create_table_with_radius)
        self.button_z_coordinate -= self.entry_z_step
        # --------------------------------------------------

    # SET ONSCREEN TIMER
    # --------------------------------------------------
    def set_onscreen_timer(self,
                           start_time_label_pos=(-1.0, 0.9),
                           onscreen_timer_pos=(-1.0, 0.8)):
        start_time = localtime()
        self.start_time_value = time()
        self.start_time_label =\
            OnscreenText(text="{0}: {1}:{2}:{3}".format(
            self.start_time_label,
            start_time[3],
            start_time[4],
            start_time[5]),
            pos=start_time_label_pos,
            align=TextNode.ALeft)
        self.onscreen_timer = OnscreenText(text=self.onscreen_timer_label,
                                           pos=onscreen_timer_pos,
                                           align=TextNode.ALeft)
    # --------------------------------------------------

    # UPDATE ONSCREEN TIMER
    # --------------------------------------------------
    def update_onscreen_timer(self, task):
        simulation_time = time() - self.start_time_value
        self.onscreen_timer.setText("{0}: {1:2.0f}:{2:2.0f}:{3:2.3f}".format(
            self.onscreen_timer_label,
            (simulation_time//3600) % 24,
            (simulation_time//60) % 60,
            simulation_time % 60))
        return Task.cont
    # --------------------------------------------------

    # TASK CLEANER
    # --------------------------------------------------
    def task_cleaner(self, task):
        if self.steps_limit == self.iter_counter or \
                self.interrupted:
            self.taskMgr.remove(self.task_name_turn_off)
            self.taskMgr.remove("update_timer")
            print("Task finished.")

            return task.done
        return task.cont
    # --------------------------------------------------

    # READ SPEED RANGE
    # --------------------------------------------------
    def read_speed_range(self):
        return tuple(map(lambda el: int32(el), self.entry_speed_range.get().split(" ")))
    # --------------------------------------------------

    # METHOD FOR GENERATING BODIES COORDINATES
    # --------------------------------------------------
    def gen_bodies_coordinates(self):
        mods = [(1, 1, self.bodies_in_row),
                (self.bodies_in_row, self.bodies_in_col, self.bodies_in_dep),
                (1, self.bodies_in_row, self.bodies_in_col)]
        self.host_bodies_coordinates = array(
            [[(body_id//mods[dim_id][0]//mods[dim_id][1]) % mods[dim_id][2]
              for body_id in range(self.bodies_amount)]
             for dim_id in range(self.dimensions)],
            dtype=float32)

        self.host_bodies_coordinates[0] = array(
            list(map(lambda el: el*self.distance_between_bodies_output,
                     self.host_bodies_coordinates[0])),
            dtype=float32)
        self.host_bodies_coordinates[1] = array(
            list(map(lambda el: el*self.distance_between_bodies_output,
```

```
                   self.host_bodies_coordinates[1])),
            dtype=float32)
        self.host_bodies_coordinates[2] = array(
            list(map(lambda el: el*self.distance_between_bodies_output,
                   self.host_bodies_coordinates[2])),
            dtype=float32)
    # --------------------------------------------------

    # METHOD FOR GENERATING BODIES SPEEDS
    # --------------------------------------------------
    def gen_bodies_speeds(self):
        self.host_bodies_speeds = [None for _ in range(self.dimensions)]
        if self.bodies_start_speed_range is None:
            self.host_bodies_speeds = zeros(
                shape=(self.dimensions, self.bodies_amount),
                dtype=float32)
            return
        for dim_id in range(self.dimensions):
            half_v = ((self.bodies_start_speed_range[1] - self.bodies_start_speed_range[0]) *
                      random.uniform(size=self.bodies_amount//2) +
self.bodies_start_speed_range[0]).astype(float32)
            self.host_bodies_speeds[dim_id] = concatenate((half_v, -1*half_v), axis=0)
            random.shuffle(self.host_bodies_speeds[dim_id])
            if self.bodies_amount % 2 != 0:
                self.host_bodies_speeds[dim_id][0] = self.host_bodies_speeds[dim_id][0]/2
                self.host_bodies_speeds[dim_id] = (
                    concatenate((self.host_bodies_speeds[dim_id],
                                 [self.host_bodies_speeds[dim_id][0]]),
                                axis=0)).astype(float32)
        self.host_bodies_speeds = array(self.host_bodies_speeds, dtype=float32)
    # --------------------------------------------------

    # METHOD FOR FILLING THE BOX
    # --------------------------------------------------
    def filling_field(self):
        for index in range(self.bodies_amount):
            self.bodies[index] = self.loader.loadModel(self.path_2_object_for_body)
            self.bodies[index].set_pos(self.host_bodies_coordinates[0][index],
                                       self.host_bodies_coordinates[1][index],
                                       self.host_bodies_coordinates[2][index])
            self.bodies[index].reparentTo(self.render)
            self.bodies[index].setTexture(self.texture_for_body, 1)
    # --------------------------------------------------

    # METHOD FOR MOVING BODIES ON CPU
    # --------------------------------------------------
    def bodies_move(self):
        for index in range(self.bodies_amount):
            self.bodies[index].set_pos(self.host_bodies_coordinates[0][index],
                                       self.host_bodies_coordinates[1][index],
                                       self.host_bodies_coordinates[2][index])
    # --------------------------------------------------

    # METHOD FOR MOVING BODIES ON GPU
    # --------------------------------------------------
    def bodies_move_gpu(self):
        for index in range(self.bodies_amount):
            self.bodies[index].set_pos(self.host_coordinates_x[index],
                                       self.host_coordinates_y[index],
                                       self.host_coordinates_z[index])
    # --------------------------------------------------

    # HIDE ELEMENTS AFTER CLICK INITIALISE BUTTON
    # --------------------------------------------------
    def hide_after_read(self):
        self.button_read_data_with_output.hide()
        self.button_read_data_without_output.hide()
        self.button_compare_cpu_gpu.hide()
        self.button_create_table_with_radius.hide()

        self.label_amount_bodies.hide()
        self.label_bodies_in_row.hide()
        self.label_bodies_in_col.hide()
        self.label_bodies_in_dep.hide()
        self.label_speed_range.hide()

        self.entry_bodies_in_row.hide()
        self.entry_bodies_in_col.hide()
        self.entry_bodies_in_dep.hide()
```

```
        self.entry_speed_range.hide()
        self.entry_amount_bodies.hide()
    # --------------------------------------------------

    # SHOW ELEMENTS AFTER READ WITH OUTPUT
    # --------------------------------------------------
    def show_after_read_with_output(self):
        self.button_start_cpu_1_ker_experiment_with_output.show()
        self.button_start_gpu_experiment_with_output.show()
    # --------------------------------------------------

    # SHOW ELEMENTS AFTER READ WITHOUT OUTPUT
    # --------------------------------------------------
    def show_after_read_without_output(self):
        self.button_start_cpu_1_ker_experiment_without_output.show()
        self.button_start_gpu_experiment_without_output.show()
    # --------------------------------------------------

    # HIDE ELEMENTS AFTER START EXPERIMENT
    # --------------------------------------------------
    def hide_after_start(self):
        self.button_start_cpu_1_ker_experiment_with_output.hide()
        self.button_start_gpu_experiment_with_output.hide()
        self.button_save_system_energy_plot.show()
    # --------------------------------------------------

    # PREPARE PLOT FOR SYSTEM ENERGY
    # --------------------------------------------------
    def prepare_plot_properties(self):
        se = 0
        ke = 1
        pe = 2
        self.fig_plot, self.subplots = plt.subplots(3, 1)
        self.fig_plot.set_figheight(6)
        self.fig_plot.set_figwidth(9)

        # Window Properties
        window_manager = plt.get_current_fig_manager()
        window_manager.window.geometry("900x1000+0+0")

        # SYSTEM ENERGY
        self.subplots[se].grid()
        self.subplots[se].set_title(
            "Bodies = " + str(self.bodies_amount),
            fontdict={"fontsize": "18",
                      "fontname": "Times New Roman",
                      "family": "serif"})
        # X axis
        plt.setp(self.subplots[se].get_xticklabels(), "fontsize", "10")
        plt.setp(self.subplots[se].get_xticklabels(), "fontname", "Times New Roman")
        plt.setp(self.subplots[se].get_xticklabels(), "family", "serif")
        # Y axis
        self.subplots[se].set_ylim(-1, 1)
        plt.setp(self.subplots[se].get_yticklabels(), "fontsize", "10")
        plt.setp(self.subplots[se].get_yticklabels(), "fontname", "Times New Roman")
        plt.setp(self.subplots[se].get_yticklabels(), "family", "serif")

        # Y axis LABEL
        self.subplots[se].set_ylabel(
            "System Energy",
            fontdict={"fontsize": 14,
                      "fontname": "Times New Roman",
                      "family": "serif"})

        # KINETIC ENERGY
        self.subplots[ke].grid()
        # X axis
        plt.setp(self.subplots[ke].get_xticklabels(), "fontsize", "10")
        plt.setp(self.subplots[ke].get_xticklabels(), "fontname", "Times New Roman")
        plt.setp(self.subplots[ke].get_xticklabels(), "family", "serif")
        # Y axis
        plt.setp(self.subplots[ke].get_yticklabels(), "fontsize", "10")
        plt.setp(self.subplots[ke].get_yticklabels(), "fontname", "Times New Roman")
        plt.setp(self.subplots[ke].get_yticklabels(), "family", "serif")
        # Y axis LABEL
        self.subplots[ke].set_ylabel(
            "Kinetic Energy",
            fontdict={"fontsize": 14,
                      "fontname": "Times New Roman",
```

```
                "family": "serif"})

    # POTENTIAL ENERGY
    self.subplots[pe].grid()
    # X axis
    plt.setp(self.subplots[pe].get_xticklabels(), "fontsize", "10")
    plt.setp(self.subplots[pe].get_xticklabels(), "fontname", "Times New Roman")
    plt.setp(self.subplots[pe].get_xticklabels(), "family", "serif")
    # Y axis
    plt.setp(self.subplots[pe].get_yticklabels(), "fontsize", "10")
    plt.setp(self.subplots[pe].get_yticklabels(), "fontname", "Times New Roman")
    plt.setp(self.subplots[pe].get_yticklabels(), "family", "serif")
    # Y axis LABEL
    self.subplots[pe].set_ylabel(
        "Potential Energy",
        fontdict={"fontsize": 14,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    self.subplots[pe].set_xlabel(
        "Time (dt=" + str(self.dt) + ")",
        fontdict={"fontsize": 14,
                  "fontname": "Times New Roman",
                  "family": "serif"})
# --------------------------------------------------

# SAVE SYSTEM ENERGY PLOT
# --------------------------------------------------
@staticmethod
def save_plot():
    plt.savefig("Plot.png")
# --------------------------------------------------

# INITIALISE MOLECULAR DYNAMIC SYSTEM WITH OUTPUT
# --------------------------------------------------
def initialise_mds_with_output(self):  # mds - Molecular Dynamic System
    self.bodies_amount = int32(self.entry_amount_bodies.get())
    self.bodies_start_speed_range = self.read_speed_range()
    self.bodies = [None for _ in range(self.bodies_amount)]
    self.bodies_in_row = int32(self.entry_bodies_in_row.get())
    self.bodies_in_col = int32(self.entry_bodies_in_col.get())
    self.bodies_in_dep = int32(self.entry_bodies_in_dep.get())

    self.box_width = self.bodies_in_row*self.distance_between_bodies_output
    self.box_height = self.bodies_in_col*self.distance_between_bodies_output
    self.box_depth = self.bodies_in_dep*self.distance_between_bodies_output

    self.tpb = int(1024)
    self.bpg = int(ceil(self.bodies_amount/self.tpb))
    self.block = (self.tpb, 1, 1)
    self.grid = (self.bpg, 1, 1)

    print("Number of bodies: {0}.".format(self.bodies_amount))
    print("Start Speeds Range: {0}.".format(self.bodies_start_speed_range))
    print("Container parameters:")
    print("\tBox Width: {0}.".format(self.box_width))
    print("\tBox Height: {0}.".format(self.box_height))
    print("\tBox Depth: {0}.".format(self.box_depth))

    # CREATE BODIES (COORDINATES)
    if self.bodies_amount > (self.bodies_in_row*self.bodies_in_col*self.bodies_in_dep):
        print("Дуже багато тіл для даних параметрів контейнеру!\n{0} > {1}".format(
            self.bodies_amount,
            self.bodies_in_row *
            self.bodies_in_col *
            self.bodies_in_dep))
        return
    self.gen_bodies_coordinates()
    self.gen_bodies_speeds()

    # SCREEN MODIFICATIONS
    self.hide_after_read()
    self.show_after_read_with_output()
    camera_y_coordinate = self.box_width if self.box_width > self.box_height \
        else self.box_height
    self.cam.setPos(self.box_width // 2,
                    -camera_y_coordinate * 2.5,
                    self.box_height // 2)
    self.cam.lookAt(self.box_width // 2,
                    0.0,
```

```
                    self.box_height // 2)
    s = timer()
    self.filling_field()  # Draw Bodies
    e = timer()
    print("Filling field time {0} s.".format(e - s))
# --------------------------------------------------

# CALCULATE FIRST STEP CPU
# --------------------------------------------------
def calculate_first_step_cpu(self):
    # Calculation Start Acceleration
    self.host_bodies_accelerates, \
        self.host_potential_energy = calc_accelerate_v01(
            bodies_coordinates=self.host_bodies_coordinates,
            bodies_amount=self.bodies_amount,
            bodies_min_distance=self.distance_between_bodies,
            sigma=self.sigma,
            epsilon=self.epsilon,
            width=self.box_width,
            height=self.box_height,
            depth=self.box_depth)
    # Calculating Kinetic Energy
    self.host_kinetic_energy = sum(calc_kinetic_energy(velocities=self.host_bodies_speeds))

    # Draw System Energy Value on Plot
    self.subplots[0].plot(self.dt_accumulator,
                          self.host_kinetic_energy + self.host_potential_energy,
                          color="red", marker=".")
    self.subplots[1].plot(self.dt_accumulator, self.host_kinetic_energy,
                          color="blue", marker=".")
    self.subplots[2].plot(self.dt_accumulator, self.host_potential_energy,
                          color="green", marker=".")
    plt.draw()
    plt.pause(10**(-10))
    self.dt_accumulator += self.dt
    self.iter_counter += 1
# --------------------------------------------------

# START CPU EXPERIMENT WITH OUTPUT
# --------------------------------------------------
def start_cpu_1_kernel_experiment_with_output(self):
    self.hide_after_start()
    print("Start CPU experiment")
    # SET AND START TIMER
    self.set_onscreen_timer()
    self.taskMgr.add(self.update_onscreen_timer, "update_timer")
    # Prepare System Energy Plot
    self.prepare_plot_properties()
    plt.ion()
    # Calculate Accelerates, Potential and Kinetic Energies
    self.calculate_first_step_cpu()
    # Start experiment
    self.taskMgr.add(self.main_cpu_1_kernel_calculation_with_output, "Move_Bodies_CPU")
# --------------------------------------------------

# MAIN PART of CPU EXPERIMENT WITH OUTPUT
# --------------------------------------------------
def main_cpu_1_kernel_calculation_with_output(self, task):
    self.host_bodies_coordinates = calc_coordinates_cpu_1ker(
        coordinates=self.host_bodies_coordinates,
        speeds=self.host_bodies_speeds,
        accelerates=self.host_bodies_accelerates,
        width=self.box_width,
        height=self.box_height,
        depth=self.box_depth,
        dt=self.dt)
    self.host_bodies_speeds = calc_speeds_cpu_1ker(
        speeds=self.host_bodies_speeds,
        accelerates=self.host_bodies_accelerates,
        dt=self.dt)
    self.host_bodies_accelerates,\
        self.host_potential_energy = calc_accelerate_v01(
            bodies_coordinates=self.host_bodies_coordinates,
            bodies_amount=self.bodies_amount,
            bodies_min_distance=self.distance_between_bodies,
            sigma=self.sigma,
            epsilon=self.epsilon,
            width=self.box_width,
            height=self.box_height,
```

```
                depth=self.box_depth)
        self.host_bodies_speeds = calc_speeds_cpu_1ker(
            speeds=self.host_bodies_speeds,
            accelerates=self.host_bodies_accelerates,
            dt=self.dt)
        # Calculating Kinetic Energy
        self.host_kinetic_energy = sum(calc_kinetic_energy(velocities=self.host_bodies_speeds))
        # Move Bodies
        self.bodies_move()
        # Draw System Energy Value on Plot
        if self.iter_counter % self.iter_window == 0:
            self.subplots[0].plot(self.dt_accumulator,
                                  self.host_kinetic_energy + self.host_potential_energy,
                                  color="red", marker=".")
            self.subplots[1].plot(self.dt_accumulator,
                                  self.host_kinetic_energy,
                                  color="blue", marker=".")
            self.subplots[2].plot(self.dt_accumulator,
                                  self.host_potential_energy,
                                  color="green", marker=".")

            plt.draw()
            plt.pause(10**(-10))
        self.dt_accumulator += self.dt
        self.iter_counter += 1
        return Task.cont
    # --------------------------------------------------

    # START GPU EXPERIMENT WITH OUTPUT
    # --------------------------------------------------
    def start_gpu_experiment_with_output(self):
        self.hide_after_start()
        print("Start GPU experiment")
        # SET AND START TIMER
        self.set_onscreen_timer()
        self.taskMgr.add(self.update_onscreen_timer, "update_timer")
        # Preparing GPU functions
        self.calc_coordinates_gpu = gpu_calc_coordinates_kernel.get_function("calc_coordinates")
        self.calc_speeds_gpu = gpu_calc_speeds_kernel.get_function("calc_speeds")
        self.calc_accelerates_gpu = gpu_calc_accelerates_kernel.get_function("calc_accelerates")
        self.calc_kinetic_energy_gpu = gpu_calc_kinetic_energy.get_function("calc_kinetic_energy")
        # Preparing CPU arrays
        self.host_coordinates_x = deepcopy(self.host_bodies_coordinates[0])
        self.host_coordinates_y = deepcopy(self.host_bodies_coordinates[1])
        self.host_coordinates_z = deepcopy(self.host_bodies_coordinates[2])

        self.host_speeds_x = deepcopy(self.host_bodies_speeds[0])
        self.host_speeds_y = deepcopy(self.host_bodies_speeds[1])
        self.host_speeds_z = deepcopy(self.host_bodies_speeds[2])

        self.host_accelerates_x = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_accelerates_y = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_accelerates_z = zeros(shape=self.bodies_amount, dtype=float32)

        self.host_kinetic_energy = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_potential_energy = zeros(shape=self.bodies_amount, dtype=float32)
        # Preparing GPU arrays
        self.gpu_coordinates_x = cuda.mem_alloc(self.host_coordinates_x.nbytes)
        self.gpu_coordinates_y = cuda.mem_alloc(self.host_coordinates_y.nbytes)
        self.gpu_coordinates_z = cuda.mem_alloc(self.host_coordinates_z.nbytes)
        cuda.memcpy_htod(self.gpu_coordinates_x, self.host_coordinates_x)
        cuda.memcpy_htod(self.gpu_coordinates_y, self.host_coordinates_y)
        cuda.memcpy_htod(self.gpu_coordinates_z, self.host_coordinates_z)

        self.gpu_speeds_x = cuda.mem_alloc(self.host_speeds_x.nbytes)
        self.gpu_speeds_y = cuda.mem_alloc(self.host_speeds_y.nbytes)
        self.gpu_speeds_z = cuda.mem_alloc(self.host_speeds_z.nbytes)
        cuda.memcpy_htod(self.gpu_speeds_x, self.host_speeds_x)
        cuda.memcpy_htod(self.gpu_speeds_y, self.host_speeds_y)
        cuda.memcpy_htod(self.gpu_speeds_z, self.host_speeds_z)

        self.gpu_accelerates_x = cuda.mem_alloc(self.host_accelerates_x.nbytes)
        self.gpu_accelerates_y = cuda.mem_alloc(self.host_accelerates_y.nbytes)
        self.gpu_accelerates_z = cuda.mem_alloc(self.host_accelerates_z.nbytes)
        cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
        cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
        cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

        self.gpu_kinetic_energy = cuda.mem_alloc(self.host_kinetic_energy.nbytes)
```

```
    cuda.memcpy_htod(self.gpu_kinetic_energy, self.host_kinetic_energy)

    self.gpu_potential_energy = cuda.mem_alloc(self.host_potential_energy.nbytes)
    cuda.memcpy_htod(self.gpu_potential_energy, self.host_potential_energy)
    # PREPARING GPU FUNCTIONS CALL
    self.calc_coordinates_gpu.prepare(
        arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, int32, float32, float32))
    self.calc_speeds_gpu.prepare(
        arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
                   int32, float32))
    self.calc_accelerates_gpu.prepare(
        arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, int32, float32, float32,
                   float32, float32, float32, float32, float32))
    self.calc_kinetic_energy_gpu.prepare(
        arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, cuda.DeviceAllocation,
                   int32, float32))
    # CALCULATION ACCELERATES and KINETIC ENERGY
    autoinit.context.synchronize()
    self.calc_accelerates_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_accelerates_x, self.gpu_accelerates_y,
        self.gpu_accelerates_z, self.gpu_coordinates_x,
        self.gpu_coordinates_y, self.gpu_coordinates_z,
        self.gpu_potential_energy, self.bodies_amount,
        self.distance_between_bodies, self.sigma,
        self.epsilon, self.dt, self.box_width,
        self.box_height, self.box_depth)
    autoinit.context.synchronize()

    self.calc_kinetic_energy_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_kinetic_energy,
        self.gpu_speeds_x, self.gpu_speeds_y,
        self.gpu_speeds_z, self.bodies_amount, float32(1.0))
    autoinit.context.synchronize()
    # TRANSFERRING DATA BACK TO THE HOST
    cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
    autoinit.context.synchronize()
    cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
    autoinit.context.synchronize()
    # Prepare System Energy Plot
    self.prepare_plot_properties()
    plt.ion()
    # Draw System Energy Value on Plot
    self.subplots[0].plot(self.dt_accumulator,
                          (sum(self.host_kinetic_energy) + sum(self.host_potential_energy)),
                          color="red", marker=".")
    self.subplots[1].plot(self.dt_accumulator,
                          sum(self.host_kinetic_energy),
                          color="blue", marker=".")
    self.subplots[2].plot(self.dt_accumulator,
                          sum(self.host_potential_energy),
                          color="green", marker=".")
    plt.draw()
    plt.pause(10**(-10))
    self.dt_accumulator += self.dt
    self.iter_counter += 1
    # Start experiment
    self.taskMgr.add(self.main_gpu_calculation_with_output, "Move_Bodies_GPU", appendTask=True)
    return
# --------------------------------------------------

# MAIN PART of GPU EXPERIMENT WITH OUTPUT
# --------------------------------------------------
def main_gpu_calculation_with_output(self, task):
    # CALCULATING NEW COORDINATES
    autoinit.context.synchronize()
    self.calc_coordinates_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_coordinates_x, self.gpu_speeds_x,
        self.gpu_accelerates_x, self.bodies_amount,
        self.box_width, self.dt)
    autoinit.context.synchronize()
    self.calc_coordinates_gpu.prepared_call(
```

```
    self.grid, self.block,
    self.gpu_coordinates_y, self.gpu_speeds_y,
    self.gpu_accelerates_y, self.bodies_amount,
    self.box_depth, self.dt)
autoinit.context.synchronize()
self.calc_coordinates_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_coordinates_z,
    self.gpu_speeds_z, self.gpu_accelerates_z,
    self.bodies_amount, self.box_height, self.dt)
autoinit.context.synchronize()
# CALCULATING FIRST PART of SPEEDS
self.calc_speeds_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_speeds_x, self.gpu_accelerates_x,
    self.bodies_amount, self.dt)
autoinit.context.synchronize()
self.calc_speeds_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_speeds_y, self.gpu_accelerates_y,
    self.bodies_amount, self.dt)
autoinit.context.synchronize()
self.calc_speeds_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_speeds_z, self.gpu_accelerates_z,
    self.bodies_amount, self.dt)
autoinit.context.synchronize()
# CALCULATING NEW ACCELERATES
cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
autoinit.context.synchronize()
cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
autoinit.context.synchronize()
cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)
autoinit.context.synchronize()
cuda.memcpy_htod(self.gpu_potential_energy, zeros(shape=self.bodies_amount, dtype=float32))
autoinit.context.synchronize()
cuda.memcpy_htod(self.gpu_kinetic_energy, zeros(shape=self.bodies_amount, dtype=float32))
autoinit.context.synchronize()

self.calc_accelerates_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_accelerates_x, self.gpu_accelerates_y,
    self.gpu_accelerates_z, self.gpu_coordinates_x,
    self.gpu_coordinates_y, self.gpu_coordinates_z,
    self.gpu_potential_energy, self.bodies_amount,
    self.distance_between_bodies, self.sigma,
    self.epsilon, self.dt, self.box_width,
    self.box_height, self.box_depth)
autoinit.context.synchronize()
# CALCULATING SECOND PART of SPEEDS
self.calc_speeds_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_speeds_x, self.gpu_accelerates_x,
    self.bodies_amount, self.dt)
autoinit.context.synchronize()
self.calc_speeds_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_speeds_y, self.gpu_accelerates_y,
    self.bodies_amount, self.dt)
autoinit.context.synchronize()
self.calc_speeds_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_speeds_z, self.gpu_accelerates_z,
    self.bodies_amount, self.dt)
autoinit.context.synchronize()
# CALCULATING KINETIC ENERGY
self.calc_kinetic_energy_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_kinetic_energy,
    self.gpu_speeds_x, self.gpu_speeds_y,
    self.gpu_speeds_z, self.bodies_amount, float32(1.0))
autoinit.context.synchronize()

cuda.memcpy_dtoh(self.host_coordinates_x, self.gpu_coordinates_x)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_coordinates_y, self.gpu_coordinates_y)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_coordinates_z, self.gpu_coordinates_z)
autoinit.context.synchronize()
```

```
        # Move Bodies
        self.bodies_move_gpu()

        if self.iter_counter % self.iter_window == 0:
            # TRANSFERRING DATA BACK TO THE HOST
            cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
            autoinit.context.synchronize()
            cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
            autoinit.context.synchronize()
            # Draw System Energy Value on Plot
            self.subplots[0].plot(self.dt_accumulator,
                                  (sum(self.host_kinetic_energy) + sum(self.host_potential_energy)),
                                  color="red", marker=".")
            self.subplots[1].plot(self.dt_accumulator,
                                  sum(self.host_kinetic_energy),
                                  color="blue", marker=".")
            self.subplots[2].plot(self.dt_accumulator,
                                  sum(self.host_potential_energy),
                                  color="green", marker=".")
            plt.draw()
            plt.pause(10**(-15))
        self.dt_accumulator += self.dt
        self.iter_counter += 1
        return task.cont
    # --------------------------------------------------

    # INITIALISE MOLECULAR DYNAMIC SYSTEM WITHOUT OUTPUT
    # --------------------------------------------------
    def initialise_mds_without_output(self):  # mds - Molecular Dynamic System
        self.bodies_amount = int32(self.entry_amount_bodies.get())
        self.bodies_start_speed_range = self.read_speed_range()
        self.bodies = [None for _ in range(self.bodies_amount)]
        self.bodies_in_row = int32(self.entry_bodies_in_row.get())
        self.bodies_in_col = int32(self.entry_bodies_in_col.get())
        self.bodies_in_dep = int32(self.entry_bodies_in_dep.get())
        self.box_width = self.bodies_in_row * self.distance_between_bodies_output
        self.box_height = self.bodies_in_col * self.distance_between_bodies_output
        self.box_depth = self.bodies_in_dep * self.distance_between_bodies_output

        self.tpb = int(1024)
        self.bpg = int(ceil(self.bodies_amount/self.tpb))
        self.block = (self.tpb, 1, 1)
        self.grid = (self.bpg, 1)

        print("Number of bodies: {0}.".format(self.bodies_amount))
        print("Start Speeds Range: {0}.".format(self.bodies_start_speed_range))
        print("Container parameters:")
        print("\tBox Width: {0}.".format(self.box_width))
        print("\tBox Height: {0}.".format(self.box_height))
        print("\tBox Depth: {0}.".format(self.box_depth))

        if self.bodies_amount > (self.bodies_in_row * self.bodies_in_col * self.bodies_in_dep):
            print("Дуже багато тіл для даних параметрів контейнеру!\n{0} > {1}".format(
                self.bodies_amount,
                self.bodies_in_row *
                self.bodies_in_col *
                self.bodies_in_dep))
            return
        # CREATE BODIES (COORDINATES)
        self.gen_bodies_coordinates()
        self.gen_bodies_speeds()

        # SCREEN MODIFICATIONS
        self.show_after_read_without_output()
    # --------------------------------------------------

    # START CPU EXPERIMENT WITHOUT OUTPUT
    # --------------------------------------------------
    def start_cpu_1_kernel_experiment_without_output(self):
        self.hide_after_start()
        print("Start CPU experiment")
        # SET AND START TIMER
        self.set_onscreen_timer()
        self.taskMgr.add(self.update_onscreen_timer, "update_timer")
        # Prepare System Energy Plot
        self.prepare_plot_properties()
        # Calculation Start Acceleration
        self.host_bodies_accelerates, \
            self.host_potential_energy = calc_accelerate_v01(
```

```
            bodies_coordinates=self.host_bodies_coordinates,
            bodies_amount=self.bodies_amount,
            bodies_min_distance=self.distance_between_bodies,
            sigma=self.sigma,
            epsilon=self.epsilon,
            width=self.box_width,
            height=self.box_height,
            depth=self.box_depth)
    # Calculating Kinetic Energy
    self.host_kinetic_energy = sum(calc_kinetic_energy(velocities=self.host_bodies_speeds))
    # Draw System Energy Value on Plot
    self.subplots[0].plot(self.dt_accumulator,
                          self.host_kinetic_energy + self.host_potential_energy,
                          color="red", marker=".")
    self.subplots[1].plot(self.dt_accumulator,
                          self.host_kinetic_energy,
                          color="blue", marker=".")
    self.subplots[2].plot(self.dt_accumulator,
                          self.host_potential_energy,
                          color="green", marker=".")
    self.dt_accumulator += self.dt
    self.iter_counter += 1
    # Start experiment
    self.taskMgr.add(self.main_cpu_1_kernel_calculation_without_output, "Move_Bodies_CPU")
# --------------------------------------------------

# MAIN PART of CPU EXPERIMENT WITHOUT OUTPUT
# --------------------------------------------------
def main_cpu_1_kernel_calculation_without_output(self, task):
    self.host_bodies_coordinates = calc_coordinates_cpu_1ker(
        coordinates=self.host_bodies_coordinates,
        speeds=self.host_bodies_speeds,
        accelerates=self.host_bodies_accelerates,
        width=self.box_width,
        height=self.box_height,
        depth=self.box_depth,
        dt=self.dt)
    self.host_bodies_speeds = calc_speeds_cpu_1ker(
        speeds=self.host_bodies_speeds,
        accelerates=self.host_bodies_accelerates,
        dt=self.dt)
    self.host_bodies_accelerates,\
        self.host_potential_energy = calc_accelerate_v01(
            bodies_coordinates=self.host_bodies_coordinates,
            bodies_amount=self.bodies_amount,
            bodies_min_distance=self.distance_between_bodies,
            sigma=self.sigma,
            epsilon=self.epsilon,
            width=self.box_width,
            height=self.box_height,
            depth=self.box_depth)
    self.host_bodies_speeds = calc_speeds_cpu_1ker(
        speeds=self.host_bodies_speeds,
        accelerates=self.host_bodies_accelerates,
        dt=self.dt)
    # Calculating Kinetic Energy
    self.host_kinetic_energy = sum(calc_kinetic_energy(velocities=self.host_bodies_speeds))
    # Draw System Energy Value on Plot
    if self.iter_counter % self.iter_window == 0:
        self.subplots[0].plot(self.dt_accumulator,
                              self.host_kinetic_energy + self.host_potential_energy,
                              color="red", marker=".")
        self.subplots[1].plot(self.dt_accumulator,
                              self.host_kinetic_energy,
                              color="blue", marker=".")
        self.subplots[2].plot(self.dt_accumulator,
                              self.host_potential_energy,
                              color="green", marker=".")
        # plt.draw()
        # plt.pause(10 ** (-10))
    self.dt_accumulator += self.dt
    self.iter_counter += 1
    return task.cont
# --------------------------------------------------

# START GPU EXPERIMENT WITHOUT OUTPUT
# --------------------------------------------------
def start_gpu_experiment_without_output(self):
    self.hide_after_start()
```

```
print("Start GPU experiment")
# SET AND START TIMER
self.set_onscreen_timer()
self.taskMgr.add(self.update_onscreen_timer, "update_timer")

self.box_width = float32(self.box_width)
self.box_height = float32(self.box_height)
self.box_depth = float32(self.box_depth)
# Preparing GPU functions
self.calc_coordinates_gpu = gpu_calc_coordinates_kernel.get_function("calc_coordinates")
self.calc_speeds_gpu = gpu_calc_speeds_kernel.get_function("calc_speeds")
self.calc_accelerates_gpu = gpu_calc_accelerates_kernel.get_function("calc_accelerates")
self.calc_kinetic_energy_gpu = gpu_calc_kinetic_energy.get_function("calc_kinetic_energy")
# Preparing CPU arrays
self.host_coordinates_x = deepcopy(self.host_bodies_coordinates[0])
self.host_coordinates_y = deepcopy(self.host_bodies_coordinates[1])
self.host_coordinates_z = deepcopy(self.host_bodies_coordinates[2])

self.host_speeds_x = deepcopy(self.host_bodies_speeds[0])
self.host_speeds_y = deepcopy(self.host_bodies_speeds[1])
self.host_speeds_z = deepcopy(self.host_bodies_speeds[2])

self.host_accelerates_x = zeros(shape=self.bodies_amount, dtype=float32)
self.host_accelerates_y = zeros(shape=self.bodies_amount, dtype=float32)
self.host_accelerates_z = zeros(shape=self.bodies_amount, dtype=float32)

self.host_kinetic_energy = zeros(shape=self.bodies_amount, dtype=float32)
self.host_potential_energy = zeros(shape=self.bodies_amount, dtype=float32)
# Preparing GPU arrays
self.gpu_coordinates_x = cuda.mem_alloc(self.host_coordinates_x.nbytes)
self.gpu_coordinates_y = cuda.mem_alloc(self.host_coordinates_y.nbytes)
self.gpu_coordinates_z = cuda.mem_alloc(self.host_coordinates_z.nbytes)
cuda.memcpy_htod(self.gpu_coordinates_x, self.host_coordinates_x)
cuda.memcpy_htod(self.gpu_coordinates_y, self.host_coordinates_y)
cuda.memcpy_htod(self.gpu_coordinates_z, self.host_coordinates_z)

self.gpu_speeds_x = cuda.mem_alloc(self.host_speeds_x.nbytes)
self.gpu_speeds_y = cuda.mem_alloc(self.host_speeds_y.nbytes)
self.gpu_speeds_z = cuda.mem_alloc(self.host_speeds_z.nbytes)
cuda.memcpy_htod(self.gpu_speeds_x, self.host_speeds_x)
cuda.memcpy_htod(self.gpu_speeds_y, self.host_speeds_y)
cuda.memcpy_htod(self.gpu_speeds_z, self.host_speeds_z)

self.gpu_accelerates_x = cuda.mem_alloc(self.host_accelerates_x.nbytes)
self.gpu_accelerates_y = cuda.mem_alloc(self.host_accelerates_y.nbytes)
self.gpu_accelerates_z = cuda.mem_alloc(self.host_accelerates_z.nbytes)
cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

self.gpu_kinetic_energy = cuda.mem_alloc(self.host_kinetic_energy.nbytes)
cuda.memcpy_htod(self.gpu_kinetic_energy, self.host_kinetic_energy)

self.gpu_potential_energy = cuda.mem_alloc(self.host_potential_energy.nbytes)
cuda.memcpy_htod(self.gpu_potential_energy, self.host_potential_energy)
# PREPARING GPU FUNCTIONS CALL
self.calc_coordinates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32, float32))
self.calc_speeds_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
self.calc_accelerates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32,
               float32, float32, float32,
               float32, float32, float32))
self.calc_kinetic_energy_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
# CALCULATION ACCELERATES and KINETIC ENERGY
autoinit.context.synchronize()
self.calc_accelerates_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_accelerates_x, self.gpu_accelerates_y,
    self.gpu_accelerates_z, self.gpu_coordinates_x,
```

```
            self.gpu_coordinates_y, self.gpu_coordinates_z,
            self.gpu_potential_energy, self.bodies_amount,
            self.distance_between_bodies, self.sigma,
            self.epsilon, self.dt, self.box_width,
            self.box_height, self.box_depth)
        autoinit.context.synchronize()

        self.calc_kinetic_energy_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_kinetic_energy, self.gpu_speeds_x,
            self.gpu_speeds_y, self.gpu_speeds_z,
            self.bodies_amount, float32(1.0))
        autoinit.context.synchronize()
        # TRANSFERRING DATA BACK TO THE HOST
        cuda.memcpy_dtoh(self.host_accelerates_x, self.gpu_accelerates_x)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_accelerates_y, self.gpu_accelerates_y)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_accelerates_z, self.gpu_accelerates_z)
        autoinit.context.synchronize()

        cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
        # Prepare System Energy Plot
        self.prepare_plot_properties()
        # Draw System Energy Value on Plot
        self.subplots[0].plot(self.dt_accumulator,
                              sum(self.host_kinetic_energy) + sum(self.host_potential_energy),
                              color="red", marker=".")
        self.subplots[1].plot(self.dt_accumulator,
                              sum(self.host_kinetic_energy),
                              color="blue", marker=".")
        self.subplots[2].plot(self.dt_accumulator, sum(self.host_potential_energy),
                              color="green", marker=".")
        self.dt_accumulator += self.dt
        self.iter_counter += 1
        # Start experiment
        self.taskMgr.add(self.main_gpu_calculation_without_output, "Move_Bodies_GPU")
        return
    # ---------------------------------------------------

    # MAIN PART of GPU EXPERIMENT WITHOUT OUTPUT
    # ---------------------------------------------------
    def main_gpu_calculation_without_output(self, task):
        # CALCULATING NEW COORDINATES
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_x, self.gpu_speeds_x,
            self.gpu_accelerates_x, self.bodies_amount,
            self.box_width, self.dt)
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_y, self.gpu_speeds_y,
            self.gpu_accelerates_y, self.bodies_amount,
            self.box_depth, self.dt)
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_z, self.gpu_speeds_z,
            self.gpu_accelerates_z, self.bodies_amount,
            self.box_height, self.dt)
        autoinit.context.synchronize()
        # CALCULATING FIRST PART of SPEEDS
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_x, self.gpu_accelerates_x,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_y, self.gpu_accelerates_y,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_z, self.gpu_accelerates_z,
```

```
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        # CALCULATING NEW ACCELERATES
        cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_potential_energy, zeros(self.bodies_amount, dtype=float32))
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_kinetic_energy, zeros(self.bodies_amount, dtype=float32))
        autoinit.context.synchronize()

        self.calc_accelerates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_accelerates_x, self.gpu_accelerates_y,
            self.gpu_accelerates_z, self.gpu_coordinates_x,
            self.gpu_coordinates_y, self.gpu_coordinates_z,
            self.gpu_potential_energy, self.bodies_amount,
            self.distance_between_bodies, self.sigma,
            self.epsilon, self.dt, self.box_width,
            self.box_height, self.box_depth)
        autoinit.context.synchronize()

        # CALCULATING SECOND PART of SPEEDS
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_x, self.gpu_accelerates_x,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_y, self.gpu_accelerates_y,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_z, self.gpu_accelerates_z,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        # CALCULATING KINETIC ENERGY
        self.calc_kinetic_energy_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_kinetic_energy, self.gpu_speeds_x,
            self.gpu_speeds_y,
            self.gpu_speeds_z, self.bodies_amount,
            float32(1.0))
        autoinit.context.synchronize()

        if self.iter_counter % self.iter_window == 0:
            # TRANSFERRING DATA BACK TO THE HOST
            cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
            autoinit.context.synchronize()
            cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
            # Draw System Energy Value on Plot
            self.subplots[0].plot(self.dt_accumulator,
                                  sum(self.host_kinetic_energy) + sum(self.host_potential_energy),
                                  color="red", marker=".")
            self.subplots[1].plot(self.dt_accumulator, sum(self.host_kinetic_energy),
                                  color="blue", marker=".")
            self.subplots[2].plot(self.dt_accumulator, sum(self.host_potential_energy),
                                  color="green", marker=".")
        self.dt_accumulator += self.dt
        self.iter_counter += 1
        return task.cont
    # --------------------------------------------------

    # START GPU EXPERIMENT WITHOUT OUTPUT WITH RADIUS
    # --------------------------------------------------
    def start_gpu_experiment_without_output_with_radius(self):
        self.hide_after_start()
        print("Start GPU experiment")
        # SET AND START TIMER
        self.set_onscreen_timer()
        self.taskMgr.add(self.update_onscreen_timer, "update_timer")

        self.box_width = float32(self.box_width)
        self.box_height = float32(self.box_height)
```

```
self.box_depth = float32(self.box_depth)
# Preparing GPU functions
self.calc_coordinates_gpu =\
    gpu_calc_coordinates_kernel.get_function("calc_coordinates")
self.calc_speeds_gpu =\
    gpu_calc_speeds_kernel.get_function("calc_speeds")
self.calc_accelerates_gpu =\
    gpu_calc_accelerates_with_r_cut_kernel.get_function("calc_accelerates_with_r_cut")
self.calc_kinetic_energy_gpu =\
    gpu_calc_kinetic_energy.get_function("calc_kinetic_energy")
# Preparing CPU arrays
self.host_coordinates_x = deepcopy(self.host_bodies_coordinates[0])
self.host_coordinates_y = deepcopy(self.host_bodies_coordinates[1])
self.host_coordinates_z = deepcopy(self.host_bodies_coordinates[2])

self.host_speeds_x = deepcopy(self.host_bodies_speeds[0])
self.host_speeds_y = deepcopy(self.host_bodies_speeds[1])
self.host_speeds_z = deepcopy(self.host_bodies_speeds[2])

self.host_accelerates_x = zeros(shape=self.bodies_amount, dtype=float32)
self.host_accelerates_y = zeros(shape=self.bodies_amount, dtype=float32)
self.host_accelerates_z = zeros(shape=self.bodies_amount, dtype=float32)

self.host_kinetic_energy = zeros(shape=self.bodies_amount, dtype=float32)
self.host_potential_energy = zeros(shape=self.bodies_amount, dtype=float32)
# Preparing GPU arrays
self.gpu_coordinates_x = cuda.mem_alloc(self.host_coordinates_x.nbytes)
self.gpu_coordinates_y = cuda.mem_alloc(self.host_coordinates_y.nbytes)
self.gpu_coordinates_z = cuda.mem_alloc(self.host_coordinates_z.nbytes)
cuda.memcpy_htod(self.gpu_coordinates_x, self.host_coordinates_x)
cuda.memcpy_htod(self.gpu_coordinates_y, self.host_coordinates_y)
cuda.memcpy_htod(self.gpu_coordinates_z, self.host_coordinates_z)

self.gpu_speeds_x = cuda.mem_alloc(self.host_speeds_x.nbytes)
self.gpu_speeds_y = cuda.mem_alloc(self.host_speeds_y.nbytes)
self.gpu_speeds_z = cuda.mem_alloc(self.host_speeds_z.nbytes)
cuda.memcpy_htod(self.gpu_speeds_x, self.host_speeds_x)
cuda.memcpy_htod(self.gpu_speeds_y, self.host_speeds_y)
cuda.memcpy_htod(self.gpu_speeds_z, self.host_speeds_x)

self.gpu_accelerates_x = cuda.mem_alloc(self.host_accelerates_x.nbytes)
self.gpu_accelerates_y = cuda.mem_alloc(self.host_accelerates_y.nbytes)
self.gpu_accelerates_z = cuda.mem_alloc(self.host_accelerates_z.nbytes)
cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

self.gpu_kinetic_energy = cuda.mem_alloc(self.host_kinetic_energy.nbytes)
cuda.memcpy_htod(self.gpu_kinetic_energy, self.host_kinetic_energy)

self.gpu_potential_energy = cuda.mem_alloc(self.host_potential_energy.nbytes)
cuda.memcpy_htod(self.gpu_potential_energy, self.host_potential_energy)
# PREPARING GPU FUNCTIONS CALL
self.calc_coordinates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32, float32))
self.calc_speeds_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
self.calc_accelerates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32,
               float32, float32, float32,
               float32, float32, float32, float32))
self.calc_kinetic_energy_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
# CALCULATION ACCELERATES and KINETIC ENERGY
autoinit.context.synchronize()
self.calc_accelerates_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_accelerates_x, self.gpu_accelerates_y,
    self.gpu_accelerates_z, self.gpu_coordinates_x,
    self.gpu_coordinates_y, self.gpu_coordinates_z,
    self.gpu_potential_energy, self.bodies_amount,
    self.distance_between_bodies, self.sigma,
```

```
            self.epsilon, self.dt, self.box_width,
            self.box_height, self.box_depth,
            self.current_cutting_radius)
        autoinit.context.synchronize()

        self.calc_kinetic_energy_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_kinetic_energy, self.gpu_speeds_x,
            self.gpu_speeds_y, self.gpu_speeds_z,
            self.bodies_amount, float32(1.0))
        autoinit.context.synchronize()
        # TRANSFERRING DATA BACK TO THE HOST
        cuda.memcpy_dtoh(self.host_accelerates_x, self.gpu_accelerates_x)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_accelerates_y, self.gpu_accelerates_y)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_accelerates_z, self.gpu_accelerates_z)
        autoinit.context.synchronize()

        cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
        # Prepare System Energy Plot
        self.prepare_plot_properties()
        # Draw System Energy Value on Plot
        self.subplots[0].plot(self.dt_accumulator,
                              sum(self.host_kinetic_energy) + sum(self.host_potential_energy),
                              color="red", marker=".")
        self.subplots[1].plot(self.dt_accumulator,
                              sum(self.host_kinetic_energy),
                              color="blue", marker=".")
        self.subplots[2].plot(self.dt_accumulator, sum(self.host_potential_energy),
                              color="green", marker=".")
        self.dt_accumulator += self.dt
        self.iter_counter += 1
        # Start experiment
        self.taskMgr.add(
            self.main_gpu_calculation_without_output_with_radius,
            "Move_Bodies_GPU")
        return
    # --------------------------------------------------

    # MAIN PART of GPU EXPERIMENT WITHOUT OUTPUT WITH RADIUS
    # --------------------------------------------------
    def main_gpu_calculation_without_output_with_radius(self, task):
        # CALCULATING NEW COORDINATES
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_x, self.gpu_speeds_x,
            self.gpu_accelerates_x, self.bodies_amount,
            self.box_width, self.dt)
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_y, self.gpu_speeds_y,
            self.gpu_accelerates_y, self.bodies_amount,
            self.box_depth, self.dt)
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_z, self.gpu_speeds_z,
            self.gpu_accelerates_z, self.bodies_amount,
            self.box_height, self.dt)
        autoinit.context.synchronize()
        # CALCULATING FIRST PART of SPEEDS
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_x, self.gpu_accelerates_x,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_y, self.gpu_accelerates_y,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_z, self.gpu_accelerates_z,
```

```
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        # CALCULATING NEW ACCELERATES
        cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

        self.calc_accelerates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_accelerates_x, self.gpu_accelerates_y,
            self.gpu_accelerates_z, self.gpu_coordinates_x,
            self.gpu_coordinates_y, self.gpu_coordinates_z,
            self.gpu_potential_energy, self.bodies_amount,
            self.distance_between_bodies, self.sigma,
            self.epsilon, self.dt, self.box_width,
            self.box_height, self.box_depth,
            self.current_cutting_radius)
        autoinit.context.synchronize()
        # CALCULATING SECOND PART of SPEEDS
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_x, self.gpu_accelerates_x,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_y, self.gpu_accelerates_y,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_z, self.gpu_accelerates_z,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        # CALCULATING KINETIC ENERGY
        self.calc_kinetic_energy_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_kinetic_energy, self.gpu_speeds_x,
            self.gpu_speeds_y,
            self.gpu_speeds_z, self.bodies_amount,
            float32(1.0))
        autoinit.context.synchronize()

        if self.iter_counter % self.iter_window == 0:
            # TRANSFERRING DATA BACK TO THE HOST
            cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
            autoinit.context.synchronize()
            cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
            # Draw System Energy Value on Plot
            self.subplots[0].plot(self.dt_accumulator,
                                  sum(self.host_kinetic_energy) + sum(self.host_potential_energy),
                                  color="red", marker=".")
            self.subplots[1].plot(self.dt_accumulator, sum(self.host_kinetic_energy),
                                  color="blue", marker=".")
            self.subplots[2].plot(self.dt_accumulator, sum(self.host_potential_energy),
                                  color="green", marker=".")
        self.dt_accumulator += self.dt
        self.iter_counter += 1
        return task.cont
    # --------------------------------------------------

    # COMPARE SPEED GPU and CPU
    # --------------------------------------------------
    def compare_cpu_and_gpu(self):
        self.experiments_amount = 10
        my_seq = Sequence(Func(self.gpu_computation,
                               self.experiments_amount),
                          Func(self.gpu_computation_with_r_cut,
                               self.experiments_amount),
                          Func(self.cpu_computation,
                               self.experiments_amount),
                          Func(self.plot_speed_gpu_vs_cpu))
        my_seq.start()
        print("Sequence began.")
    # --------------------------------------------------

    # COMPARE SPEED: GPU vs CPU PLOT
```

```
# --------------------------------------------------
def plot_speed_gpu_vs_cpu(self):
    self.fig_plot, self.subplots = plt.subplots(1, 3)
    self.fig_plot.set_figheight(9)
    self.fig_plot.set_figwidth(20)
    # CPU_TIME&GPU_TIME
    self.subplots[0].set_title(
        "Computation Speed: CPU vs GPU",
        fontdict={"fontsize": "18",
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # X axis
    plt.setp(self.subplots[0].get_xticklabels(), "fontsize", "10")
    plt.setp(self.subplots[0].get_xticklabels(), "fontname", "Times New Roman")
    plt.setp(self.subplots[0].get_xticklabels(), "family", "serif")
    # X axis LABEL
    self.subplots[0].set_xlabel(
        "System Size: (NxNxN)",
        fontdict={"fontsize": 14,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # Y axis
    plt.setp(self.subplots[0].get_yticklabels(), "fontsize", "10")
    plt.setp(self.subplots[0].get_yticklabels(), "fontname", "Times New Roman")
    plt.setp(self.subplots[0].get_yticklabels(), "family", "serif")
    # Y axis LABEL
    self.subplots[0].set_ylabel(
        "Calculation Time sec",
        fontdict={"fontsize": 14,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # COMPARE GPU vs CPU (LOG2)
    self.subplots[1].set_title(
        "Computation Speed: CPU vs GPU",
        fontdict={"fontsize": "18",
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # X axis
    plt.setp(self.subplots[1].get_xticklabels(), "fontsize", "10")
    plt.setp(self.subplots[1].get_xticklabels(), "fontname", "Times New Roman")
    plt.setp(self.subplots[1].get_xticklabels(), "family", "serif")
    # X axis LABEL
    self.subplots[1].set_xlabel(
        "System Size: (NxNxN)",
        fontdict={"fontsize": 14,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # Y axis
    plt.setp(self.subplots[1].get_yticklabels(), "fontsize", "10")
    plt.setp(self.subplots[1].get_yticklabels(), "fontname", "Times New Roman")
    plt.setp(self.subplots[1].get_yticklabels(), "family", "serif")
    # Y axis LABEL
    self.subplots[1].set_ylabel(
        "Calculation Time log2(sec)",
        fontdict={"fontsize": 14,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # CPU_TIME/GPU_TIME
    self.subplots[2].set_title(
        "Computation Speed: CPU/GPU",
        fontdict={"fontsize": "18",
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # X axis
    plt.setp(self.subplots[2].get_xticklabels(), "fontsize", "10")
    plt.setp(self.subplots[2].get_xticklabels(), "fontname", "Times New Roman")
    plt.setp(self.subplots[2].get_xticklabels(), "family", "serif")
    # X axis LABEL
    self.subplots[2].set_xlabel(
        "System Size: (NxNxN)",
        fontdict={"fontsize": 14,
                  "fontname": "Times New Roman",
                  "family": "serif"})
    # Y axis
    plt.setp(self.subplots[2].get_yticklabels(), "fontsize", "10")
    plt.setp(self.subplots[2].get_yticklabels(), "fontname", "Times New Roman")
    plt.setp(self.subplots[2].get_yticklabels(), "family", "serif")
    # Y axis LABEL
    self.subplots[2].set_ylabel(
```

```
    "CPU/GPU",
    fontdict={"fontsize": 14,
              "fontname": "Times New Roman",
              "family": "serif"})

cpu_computation_time_log = array(
    list(map(lambda el: log2(el),
             self.cpu_computation_time)),
    dtype=float32)
gpu_computation_time_log = array(
    list(map(lambda el: log2(el),
             self.gpu_computation_time)),
    dtype=float32)
gpu_r_cut_computation_time_log = array(
    list(map(lambda el: log2(el),
             self.gpu_r_cut_computation_time)),
    dtype=float32)
frac_cpu_gpu_computation_time = array(
    self.cpu_computation_time)/array(self.gpu_computation_time)
frac_cpu_gpu_r_cut_computation_time = array(
    self.cpu_computation_time/array(self.gpu_r_cut_computation_time))
# PLOT GPU&CPU
# CPU
self.subplots[0].plot(
    list(range(2, self.experiments_amount + 1)),
    self.cpu_computation_time,
    color=(0, 0.9, 0), marker="d",
    markersize=12, markerfacecolor=(0, 0.5, 0),
    label="CPU")
# GPU
self.subplots[0].plot(
    list(range(2, self.experiments_amount + 1)),
    self.gpu_computation_time,
    color=(0, 0.5, 0), marker="v",
    markersize=12, markerfacecolor=(0, 0.9, 0),
    label="GPU")
# GPU with CUTTING RADIUS
self.subplots[0].plot(
    list(range(2, self.experiments_amount + 1)),
    self.gpu_r_cut_computation_time,
    linestyle="--",
    color=(0.0, 0.0, 1.0), marker="<",
    markersize=12, markerfacecolor=(0.9, 0.0, 0.0),
    label="GPU with Radius")
# LOG2 PLOT GPU&CPU
# CPU LOG2
self.subplots[1].plot(
    list(range(2, self.experiments_amount + 1)),
    cpu_computation_time_log,
    color=(0, 0.9, 0), marker="d",
    markersize=12, markerfacecolor=(0, 0.5, 0),
    label="CPU")
# GPU LOG2
self.subplots[1].plot(
    list(range(2, self.experiments_amount + 1)),
    gpu_computation_time_log,
    color=(0, 0.5, 0), marker="v",
    markersize=12, markerfacecolor=(0, 0.9, 0),
    label="GPU")
# GPU with CUTTING RADIUS LOG2
self.subplots[1].plot(
    list(range(2, self.experiments_amount + 1)),
    gpu_r_cut_computation_time_log,
    linestyle="--",
    color=(0.0, 0.0, 1.0), marker="<",
    markersize=12, markerfacecolor=(0.9, 0.0, 0.0),
    label="GPU with Radius")
# FRACTIONS PLOT
# CPU/GPU
self.subplots[2].plot(
    list(range(2, self.experiments_amount + 1)),
    frac_cpu_gpu_computation_time,
    color=(1, 0, 0), marker="^",
    markersize=12, markerfacecolor=(0, 0, 1),
    label="CPU/GPU")
# CPU/GPU with CUTTING RADIUS
self.subplots[2].plot(
    list(range(2, self.experiments_amount + 1)),
    frac_cpu_gpu_r_cut_computation_time,
```

```
            linestyle="--",
            color=(0, 0, 1), marker="d",
            markersize=12, markerfacecolor=(1, 0, 0),
            label="CPU/GPU+R")
        self.subplots[0].grid()
        self.subplots[1].grid()
        self.subplots[2].grid()
        self.subplots[0].legend()
        self.subplots[1].legend()
        self.subplots[2].legend()
        plt.savefig("CompareSpeed_GPUvsCPU.png")
        plt.show()
    # --------------------------------------------------

    # CPU-s COMPUTATION
    # --------------------------------------------------
    def cpu_computation(self, amount, bottom_lim=2):
        top_lim = amount + 1
        amount_bodies_list = list(range(bottom_lim, top_lim))
        self.cpu_computation_time = [None for _ in range(len(amount_bodies_list))]
        for ind in range(len(self.cpu_computation_time)):
            s = timer()
            bodies_in_1_dimension = amount_bodies_list[ind]
            self.bodies_amount = int32(bodies_in_1_dimension**3)
            self.bodies_start_speed_range = (-1, 1)
            self.bodies = [None for _ in range(self.bodies_amount)]
            self.bodies_in_row = int32(bodies_in_1_dimension)
            self.bodies_in_col = int32(bodies_in_1_dimension)
            self.bodies_in_dep = int32(bodies_in_1_dimension)
            self.box_width = self.bodies_in_row*self.distance_between_bodies_output
            self.box_height = self.bodies_in_col*self.distance_between_bodies_output
            self.box_depth = self.bodies_in_dep*self.distance_between_bodies_output

            self.tpb = int(1024)
            self.bpg = int(ceil(self.bodies_amount/self.tpb))
            self.block = (self.tpb, 1, 1)
            self.grid = (self.bpg, 1)
            # Prepare Coordinates & Speeds
            self.gen_bodies_coordinates()
            self.gen_bodies_speeds()

            self.iter_counter = int32(0)
            self.dt_accumulator = float32(0.0)

            print("Start CPU experiment")
            # Calculation Start Acceleration
            self.host_bodies_accelerates, \
                self.host_potential_energy = calc_accelerate_v01(
                    bodies_coordinates=self.host_bodies_coordinates,
                    bodies_amount=self.bodies_amount,
                    bodies_min_distance=self.distance_between_bodies,
                    sigma=self.sigma,
                    epsilon=self.epsilon,
                    width=self.box_width,
                    height=self.box_height,
                    depth=self.box_depth)
            # Calculating Kinetic Energy
            self.host_kinetic_energy =\
                sum(calc_kinetic_energy(velocities=self.host_bodies_speeds))
            # Draw System Energy Value on Plot
            self.dt_accumulator += self.dt
            self.iter_counter += 1

            while self.iter_counter != 1000:
                self.host_bodies_coordinates = calc_coordinates_cpu_1ker(
                    coordinates=self.host_bodies_coordinates,
                    speeds=self.host_bodies_speeds,
                    accelerates=self.host_bodies_accelerates,
                    width=self.box_width,
                    height=self.box_height,
                    depth=self.box_depth,
                    dt=self.dt)
                self.host_bodies_speeds = calc_speeds_cpu_1ker(
                    speeds=self.host_bodies_speeds,
                    accelerates=self.host_bodies_accelerates,
                    dt=self.dt)
                self.host_bodies_accelerates, \
                    self.host_potential_energy = calc_accelerate_v01(
                        bodies_coordinates=self.host_bodies_coordinates,
```

```
                    bodies_amount=self.bodies_amount,
                    bodies_min_distance=self.distance_between_bodies,
                    sigma=self.sigma,
                    epsilon=self.epsilon,
                    width=self.box_width,
                    height=self.box_height,
                    depth=self.box_depth)
            self.host_bodies_speeds = calc_speeds_cpu_1ker(
                speeds=self.host_bodies_speeds,
                accelerates=self.host_bodies_accelerates,
                dt=self.dt)
            # Calculating Kinetic Energy
            self.host_kinetic_energy =\
                sum(calc_kinetic_energy(velocities=self.host_bodies_speeds))
            self.dt_accumulator += self.dt
            self.iter_counter += 1
        # REPORT about CPU
        e = timer()
        self.cpu_computation_time[ind] = e - s
        print("CPU. Step: {0}. Time: {1} s.".format(bodies_in_1_dimension, e - s))
    return
# --------------------------------------------------

# GPU-s COMPUTATION
# --------------------------------------------------
def gpu_computation(self, amount, bottom_lim=2):
    top_lim = amount + 1
    amount_bodies_list = list(range(bottom_lim, top_lim))
    self.gpu_computation_time = [None for _ in range(len(amount_bodies_list))]
    for ind in range(len(amount_bodies_list)):
        s = timer()
        bodies_in_1_dimension = amount_bodies_list[ind]
        self.bodies_amount = int32(bodies_in_1_dimension**3)
        self.bodies_start_speed_range = (-1, 1)
        self.bodies = [None for _ in range(self.bodies_amount)]
        self.bodies_in_row = int32(bodies_in_1_dimension)
        self.bodies_in_col = int32(bodies_in_1_dimension)
        self.bodies_in_dep = int32(bodies_in_1_dimension)
        self.box_width = self.bodies_in_row*self.distance_between_bodies_output
        self.box_height = self.bodies_in_col*self.distance_between_bodies_output
        self.box_depth = self.bodies_in_dep*self.distance_between_bodies_output

        self.tpb = int(1024)
        self.bpg = int(ceil(self.bodies_amount/self.tpb))
        self.block = (self.tpb, 1, 1)
        self.grid = (self.bpg, 1)
        # Prepare Coordinates & Speeds
        self.gen_bodies_coordinates()
        self.gen_bodies_speeds()

        self.iter_counter = int32(0)
        self.dt_accumulator = float32(0.0)

        print("Start GPU experiment")

        self.box_width = float32(self.box_width)
        self.box_height = float32(self.box_height)
        self.box_depth = float32(self.box_depth)
        # Preparing GPU functions
        self.calc_coordinates_gpu = gpu_calc_coordinates_kernel.get_function("calc_coordinates")
        self.calc_speeds_gpu = gpu_calc_speeds_kernel.get_function("calc_speeds")
        self.calc_accelerates_gpu = gpu_calc_accelerates_kernel.get_function("calc_accelerates")
        self.calc_kinetic_energy_gpu = gpu_calc_kinetic_energy.get_function("calc_kinetic_energy")
        # Preparing CPU arrays
        self.host_coordinates_x = deepcopy(self.host_bodies_coordinates[0])
        self.host_coordinates_y = deepcopy(self.host_bodies_coordinates[1])
        self.host_coordinates_z = deepcopy(self.host_bodies_coordinates[2])

        self.host_speeds_x = deepcopy(self.host_bodies_speeds[0])
        self.host_speeds_y = deepcopy(self.host_bodies_speeds[1])
        self.host_speeds_z = deepcopy(self.host_bodies_speeds[2])

        self.host_accelerates_x = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_accelerates_y = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_accelerates_z = zeros(shape=self.bodies_amount, dtype=float32)

        self.host_kinetic_energy = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_potential_energy = zeros(shape=self.bodies_amount, dtype=float32)
        # Preparing GPU arrays
```

```
self.gpu_coordinates_x = cuda.mem_alloc(self.host_coordinates_x.nbytes)
self.gpu_coordinates_y = cuda.mem_alloc(self.host_coordinates_y.nbytes)
self.gpu_coordinates_z = cuda.mem_alloc(self.host_coordinates_z.nbytes)
cuda.memcpy_htod(self.gpu_coordinates_x, self.host_coordinates_x)
cuda.memcpy_htod(self.gpu_coordinates_y, self.host_coordinates_y)
cuda.memcpy_htod(self.gpu_coordinates_z, self.host_coordinates_z)

self.gpu_speeds_x = cuda.mem_alloc(self.host_speeds_x.nbytes)
self.gpu_speeds_y = cuda.mem_alloc(self.host_speeds_y.nbytes)
self.gpu_speeds_z = cuda.mem_alloc(self.host_speeds_z.nbytes)
cuda.memcpy_htod(self.gpu_speeds_x, self.host_speeds_x)
cuda.memcpy_htod(self.gpu_speeds_y, self.host_speeds_y)
cuda.memcpy_htod(self.gpu_speeds_z, self.host_speeds_x)

self.gpu_accelerates_x = cuda.mem_alloc(self.host_accelerates_x.nbytes)
self.gpu_accelerates_y = cuda.mem_alloc(self.host_accelerates_y.nbytes)
self.gpu_accelerates_z = cuda.mem_alloc(self.host_accelerates_z.nbytes)
cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

self.gpu_kinetic_energy = cuda.mem_alloc(self.host_kinetic_energy.nbytes)
cuda.memcpy_htod(self.gpu_kinetic_energy, self.host_kinetic_energy)

self.gpu_potential_energy = cuda.mem_alloc(self.host_potential_energy.nbytes)
cuda.memcpy_htod(self.gpu_potential_energy, self.host_potential_energy)
# PREPARING GPU FUNCTIONS CALL
self.calc_coordinates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32, float32))
self.calc_speeds_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
self.calc_accelerates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32,
               float32, float32, float32,
               float32, float32, float32))
self.calc_kinetic_energy_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
# CALCULATION ACCELERATES and KINETIC ENERGY
autoinit.context.synchronize()
self.calc_accelerates_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_accelerates_x, self.gpu_accelerates_y,
    self.gpu_accelerates_z, self.gpu_coordinates_x,
    self.gpu_coordinates_y, self.gpu_coordinates_z,
    self.gpu_potential_energy, self.bodies_amount,
    self.distance_between_bodies, self.sigma,
    self.epsilon, self.dt, self.box_width,
    self.box_height, self.box_depth)
autoinit.context.synchronize()

self.calc_kinetic_energy_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_kinetic_energy, self.gpu_speeds_x,
    self.gpu_speeds_y, self.gpu_speeds_z,
    self.bodies_amount, float32(1.0))
autoinit.context.synchronize()
# TRANSFERRING DATA BACK TO THE HOST
cuda.memcpy_dtoh(self.host_accelerates_x, self.gpu_accelerates_x)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_accelerates_y, self.gpu_accelerates_y)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_accelerates_z, self.gpu_accelerates_z)
autoinit.context.synchronize()

cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
self.dt_accumulator += self.dt
self.iter_counter += 1

while self.iter_counter != 1000:
    # CALCULATING NEW COORDINATES
```

```
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_x, self.gpu_speeds_x,
            self.gpu_accelerates_x, self.bodies_amount,
            self.box_width, self.dt)
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_y, self.gpu_speeds_y,
            self.gpu_accelerates_y, self.bodies_amount,
            self.box_depth, self.dt)
        autoinit.context.synchronize()
        self.calc_coordinates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_coordinates_z, self.gpu_speeds_z,
            self.gpu_accelerates_z, self.bodies_amount,
            self.box_height, self.dt)
        autoinit.context.synchronize()
        # CALCULATING FIRST PART of SPEEDS
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_x, self.gpu_accelerates_x,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_y, self.gpu_accelerates_y,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_z, self.gpu_accelerates_z,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        # CALCULATING NEW ACCELERATES
        cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

        self.calc_accelerates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_accelerates_x, self.gpu_accelerates_y,
            self.gpu_accelerates_z, self.gpu_coordinates_x,
            self.gpu_coordinates_y, self.gpu_coordinates_z,
            self.gpu_potential_energy, self.bodies_amount,
            self.distance_between_bodies, self.sigma,
            self.epsilon, self.dt, self.box_width,
            self.box_height, self.box_depth)
        autoinit.context.synchronize()
        # CALCULATING SECOND PART of SPEEDS
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_x, self.gpu_accelerates_x,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_y, self.gpu_accelerates_y,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_z, self.gpu_accelerates_z,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        # CALCULATING KINETIC ENERGY
        self.calc_kinetic_energy_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_kinetic_energy, self.gpu_speeds_x,
            self.gpu_speeds_y, self.gpu_speeds_z,
            self.bodies_amount, float32(1.0))
        autoinit.context.synchronize()

        self.dt_accumulator += self.dt
        self.iter_counter += 1
    e = timer()
```

```
        self.gpu_computation_time[ind] = e - s
        print("GPU. Step: {0}. Time: {1} s.".format(bodies_in_1_dimension, e - s))
    return
# --------------------------------------------------

# GPU-s COMPUTATION with CUTTING RADIUS
# --------------------------------------------------
def gpu_computation_with_r_cut(self, amount, bottom_lim=2, cutting_radius=float32(28.4)):
    top_lim = amount + 1
    amount_bodies_list = list(range(bottom_lim, top_lim))
    self.gpu_r_cut_computation_time = [None for _ in range(len(amount_bodies_list))]
    for ind in range(len(amount_bodies_list)):
        s = timer()
        bodies_in_1_dimension = amount_bodies_list[ind]
        self.bodies_amount = int32(bodies_in_1_dimension ** 3)
        self.bodies_start_speed_range = (-1, 1)
        self.bodies = [None for _ in range(self.bodies_amount)]
        self.bodies_in_row = int32(bodies_in_1_dimension)
        self.bodies_in_col = int32(bodies_in_1_dimension)
        self.bodies_in_dep = int32(bodies_in_1_dimension)
        self.box_width = self.bodies_in_row * self.distance_between_bodies_output
        self.box_height = self.bodies_in_col * self.distance_between_bodies_output
        self.box_depth = self.bodies_in_dep * self.distance_between_bodies_output

        self.tpb = int(1024)
        self.bpg = int(ceil(self.bodies_amount / self.tpb))
        self.block = (self.tpb, 1, 1)
        self.grid = (self.bpg, 1)
        # Prepare Coordinates & Speeds
        self.gen_bodies_coordinates()
        self.gen_bodies_speeds()

        self.iter_counter = int32(0)
        self.dt_accumulator = float32(0.0)

        print("Start GPU experiment")

        self.box_width = float32(self.box_width)
        self.box_height = float32(self.box_height)
        self.box_depth = float32(self.box_depth)
        # Preparing GPU functions
        self.calc_coordinates_gpu = gpu_calc_coordinates_kernel.get_function("calc_coordinates")
        self.calc_speeds_gpu = gpu_calc_speeds_kernel.get_function("calc_speeds")
        self.calc_accelerates_gpu =\
            gpu_calc_accelerates_with_r_cut_kernel.get_function("calc_accelerates_with_r_cut")
        self.calc_kinetic_energy_gpu = gpu_calc_kinetic_energy.get_function("calc_kinetic_energy")
        # Preparing CPU arrays
        self.host_coordinates_x = deepcopy(self.host_bodies_coordinates[0])
        self.host_coordinates_y = deepcopy(self.host_bodies_coordinates[1])
        self.host_coordinates_z = deepcopy(self.host_bodies_coordinates[2])

        self.host_speeds_x = deepcopy(self.host_bodies_speeds[0])
        self.host_speeds_y = deepcopy(self.host_bodies_speeds[1])
        self.host_speeds_z = deepcopy(self.host_bodies_speeds[2])

        self.host_accelerates_x = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_accelerates_y = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_accelerates_z = zeros(shape=self.bodies_amount, dtype=float32)

        self.host_kinetic_energy = zeros(shape=self.bodies_amount, dtype=float32)
        self.host_potential_energy = zeros(shape=self.bodies_amount, dtype=float32)
        # Preparing GPU arrays
        self.gpu_coordinates_x = cuda.mem_alloc(self.host_coordinates_x.nbytes)
        self.gpu_coordinates_y = cuda.mem_alloc(self.host_coordinates_y.nbytes)
        self.gpu_coordinates_z = cuda.mem_alloc(self.host_coordinates_z.nbytes)
        cuda.memcpy_htod(self.gpu_coordinates_x, self.host_coordinates_x)
        cuda.memcpy_htod(self.gpu_coordinates_y, self.host_coordinates_y)
        cuda.memcpy_htod(self.gpu_coordinates_z, self.host_coordinates_z)

        self.gpu_speeds_x = cuda.mem_alloc(self.host_speeds_x.nbytes)
        self.gpu_speeds_y = cuda.mem_alloc(self.host_speeds_y.nbytes)
        self.gpu_speeds_z = cuda.mem_alloc(self.host_speeds_z.nbytes)
        cuda.memcpy_htod(self.gpu_speeds_x, self.host_speeds_x)
        cuda.memcpy_htod(self.gpu_speeds_y, self.host_speeds_y)
        cuda.memcpy_htod(self.gpu_speeds_z, self.host_speeds_x)

        self.gpu_accelerates_x = cuda.mem_alloc(self.host_accelerates_x.nbytes)
        self.gpu_accelerates_y = cuda.mem_alloc(self.host_accelerates_y.nbytes)
        self.gpu_accelerates_z = cuda.mem_alloc(self.host_accelerates_z.nbytes)
```

```
cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

self.gpu_kinetic_energy = cuda.mem_alloc(self.host_kinetic_energy.nbytes)
cuda.memcpy_htod(self.gpu_kinetic_energy, self.host_kinetic_energy)

self.gpu_potential_energy = cuda.mem_alloc(self.host_potential_energy.nbytes)
cuda.memcpy_htod(self.gpu_potential_energy, self.host_potential_energy)
# PREPARING GPU FUNCTIONS CALL
self.calc_coordinates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32, float32))
self.calc_speeds_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
self.calc_accelerates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32,
               float32, float32, float32,
               float32, float32, float32, float32))
self.calc_kinetic_energy_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
# CALCULATION ACCELERATES and KINETIC ENERGY
autoinit.context.synchronize()
self.calc_accelerates_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_accelerates_x, self.gpu_accelerates_y,
    self.gpu_accelerates_z, self.gpu_coordinates_x,
    self.gpu_coordinates_y, self.gpu_coordinates_z,
    self.gpu_potential_energy, self.bodies_amount,
    self.distance_between_bodies, self.sigma,
    self.epsilon, self.dt, self.box_width,
    self.box_height, self.box_depth, cutting_radius)
autoinit.context.synchronize()

self.calc_kinetic_energy_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_kinetic_energy, self.gpu_speeds_x,
    self.gpu_speeds_y, self.gpu_speeds_z,
    self.bodies_amount, float32(1.0))
autoinit.context.synchronize()
# TRANSFERRING DATA BACK TO THE HOST
cuda.memcpy_dtoh(self.host_accelerates_x, self.gpu_accelerates_x)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_accelerates_y, self.gpu_accelerates_y)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_accelerates_z, self.gpu_accelerates_z)
autoinit.context.synchronize()

cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)

self.dt_accumulator += self.dt
self.iter_counter += 1

while self.iter_counter != 1000:
    # CALCULATING NEW COORDINATES
    autoinit.context.synchronize()
    self.calc_coordinates_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_coordinates_x, self.gpu_speeds_x,
        self.gpu_accelerates_x, self.bodies_amount,
        self.box_width, self.dt)
    autoinit.context.synchronize()
    self.calc_coordinates_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_coordinates_y, self.gpu_speeds_y,
        self.gpu_accelerates_y, self.bodies_amount,
        self.box_depth, self.dt)
    autoinit.context.synchronize()
    self.calc_coordinates_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_coordinates_z, self.gpu_speeds_z,
```

```
                self.gpu_accelerates_z, self.bodies_amount,
                self.box_height, self.dt)
            autoinit.context.synchronize()
            # CALCULATING FIRST PART of SPEEDS
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_x, self.gpu_accelerates_x,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_y, self.gpu_accelerates_y,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_z, self.gpu_accelerates_z,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            # CALCULATING NEW ACCELERATES
            cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
            autoinit.context.synchronize()
            cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
            autoinit.context.synchronize()
            cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

            self.calc_accelerates_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_accelerates_x, self.gpu_accelerates_y,
                self.gpu_accelerates_z, self.gpu_coordinates_x,
                self.gpu_coordinates_y, self.gpu_coordinates_z,
                self.gpu_potential_energy, self.bodies_amount,
                self.distance_between_bodies, self.sigma,
                self.epsilon, self.dt, self.box_width,
                self.box_height, self.box_depth, cutting_radius)
            autoinit.context.synchronize()
            # CALCULATING SECOND PART of SPEEDS
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_x, self.gpu_accelerates_x,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_y, self.gpu_accelerates_y,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_z, self.gpu_accelerates_z,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            # CALCULATING KINETIC ENERGY
            self.calc_kinetic_energy_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_kinetic_energy, self.gpu_speeds_x,
                self.gpu_speeds_y, self.gpu_speeds_z,
                self.bodies_amount, float32(1.0))
            autoinit.context.synchronize()

            self.dt_accumulator += self.dt
            self.iter_counter += 1
        e = timer()
        self.gpu_r_cut_computation_time[ind] = e - s
        print("GPU. Step: {0}. Time: {1} s.".format(bodies_in_1_dimension, e - s))
    return
# --------------------------------------------------

# CREATING TABLE WITH CUTTING RADIUS
# --------------------------------------------------
def create_table_with_radius(self):
    for system_size in self.amount_bodies_in_system:
        print("Table building started.")
        self.iter_window = 1
        self.initialise_experiment_with_cutting_radius(
            system_size=system_size)
        self.start_gpu_experiment_without_radius()
        self.update_data_gpu_experiment_with_cutting_radius(
            system_size=system_size,
```

```
                first_try=True)
            self.start_gpu_experiment_with_radius(system_size=system_size)
            create_radius_table(
                system_size,
                self.cutting_radius_list,
                self.experiment_average_energy_list,
                self.reference_average_energy,
                self.experiment_average_kinetic_energy_list,
                self.reference_average_kinetic_energy,
                self.experiment_average_potential_energy_list,
                self.reference_average_potential_energy,
                directory_name="Tables\\{0}\\".format(system_size))
            print("Table Created.")
        return
    # --------------------------------------------------

    # INITIALISE EXPERIMENT with CUTTING RADIUS
    # --------------------------------------------------
    def initialise_experiment_with_cutting_radius(self, system_size):
        # DIRECTORY CREATION
        path2directory = "Tables\\{0}".format(system_size)
        if os.path.isdir("Tables\\{0}\\".format(system_size)):
            shutil.rmtree(path2directory)
        os.mkdir(os.getcwd() + "\\" + path2directory)
        # PREPARING VARIABLES
        # PLOT ARRAYS
        self.dt_list = array([i*self.dt for i in range(self.steps_limit)])
        self.system_energy_list = zeros(shape=self.steps_limit, dtype=float32)
        self.kinetic_energy_list = zeros(shape=self.steps_limit, dtype=float32)
        self.potential_energy_list = zeros(shape=self.steps_limit, dtype=float32)

        self.bodies_amount = int32(system_size)
        self.bodies_start_speed_range = (float32(-1), float32(1))
        self.bodies = [None for _ in range(self.bodies_amount)]

        self.bodies_in_row = \
            int32(self.box_parameters[system_size][self.box_width_index])
        self.bodies_in_col = \
            int32(self.box_parameters[system_size][self.box_height_index])
        self.bodies_in_dep = \
            int32(self.box_parameters[system_size][self.box_depth_index])

        self.box_width = self.bodies_in_row * self.distance_between_bodies_output
        self.box_height = self.bodies_in_col * self.distance_between_bodies_output
        self.box_depth = self.bodies_in_dep * self.distance_between_bodies_output

        self.tpb = int(1024)
        self.bpg = int(ceil(self.bodies_amount / self.tpb))
        self.block = (self.tpb, 1, 1)
        self.grid = (self.bpg, 1)

        print("Number of bodies: {0}.".format(self.bodies_amount))
        print("Start Speeds Range: {0}.".format(self.bodies_start_speed_range))
        print("Container parameters:")
        print("\tBox Width: {0}.".format(self.box_width))
        print("\tBox Height: {0}.".format(self.box_height))
        print("\tBox Depth: {0}.".format(self.box_depth))

        if self.bodies_amount > (self.bodies_in_row * self.bodies_in_col * self.bodies_in_dep):
            print("Дуже багато тіл для даних параметрів контейнеру!\n{0} > {1}".format(
                self.bodies_amount,
                self.bodies_in_row *
                self.bodies_in_col *
                self.bodies_in_dep))
            return
        # CREATE BODIES (COORDINATES)
        self.gen_bodies_coordinates()
        self.gen_bodies_speeds()
        # SAVING COORDINATES&SPEEDS in GLOBAL STORE
        self.global_bodies_coordinates = deepcopy(self.host_bodies_coordinates)
        self.global_bodies_speeds = deepcopy(self.host_bodies_speeds)
        # CUTTING RADIUS PARAMETERS
        self.current_cutting_radius = None
        self.cutting_radius_step = 0.1
        self.first_cutting_radius = \
            2*self.sigma - 2*self.cutting_radius_step if self.cutting_radius_step < 1\
            else 2*self.sigma
        self.last_cutting_radius = \
            ceil(pow((0.5*self.box_width)**2 +
```

```
                (0.5*self.box_height)**2 +
                (0.5*self.box_depth)**2, 1/2))
    self.cutting_radius_list = \
        arange(self.first_cutting_radius,
               self.last_cutting_radius + self.cutting_radius_step,
               self.cutting_radius_step)
    print(self.cutting_radius_list)
    print("Кількість: {0}".format(len(self.cutting_radius_list)))
# --------------------------------------------------

# START GPU EXPERIMENT WITHOUT RADIUS
# --------------------------------------------------
def start_gpu_experiment_without_radius(self):
    # Start Experiment Timer
    global_time_start = timer()
    # Preparing GPU functions
    self.calc_coordinates_gpu = gpu_calc_coordinates_kernel.get_function("calc_coordinates")
    self.calc_speeds_gpu = gpu_calc_speeds_kernel.get_function("calc_speeds")
    self.calc_accelerates_gpu = gpu_calc_accelerates_kernel.get_function("calc_accelerates")
    self.calc_kinetic_energy_gpu = gpu_calc_kinetic_energy.get_function("calc_kinetic_energy")
    # Preparing CPU arrays
    self.host_coordinates_x = deepcopy(self.host_bodies_coordinates[0])
    self.host_coordinates_y = deepcopy(self.host_bodies_coordinates[1])
    self.host_coordinates_z = deepcopy(self.host_bodies_coordinates[2])

    self.host_speeds_x = deepcopy(self.host_bodies_speeds[0])
    self.host_speeds_y = deepcopy(self.host_bodies_speeds[1])
    self.host_speeds_z = deepcopy(self.host_bodies_speeds[2])

    self.host_accelerates_x = zeros(shape=self.bodies_amount, dtype=float32)
    self.host_accelerates_y = zeros(shape=self.bodies_amount, dtype=float32)
    self.host_accelerates_z = zeros(shape=self.bodies_amount, dtype=float32)

    self.host_kinetic_energy = zeros(shape=self.bodies_amount, dtype=float32)
    self.host_potential_energy = zeros(shape=self.bodies_amount, dtype=float32)
    # Preparing GPU arrays
    self.gpu_coordinates_x = cuda.mem_alloc(self.host_coordinates_x.nbytes)
    self.gpu_coordinates_y = cuda.mem_alloc(self.host_coordinates_y.nbytes)
    self.gpu_coordinates_z = cuda.mem_alloc(self.host_coordinates_z.nbytes)
    cuda.memcpy_htod(self.gpu_coordinates_x, self.host_coordinates_x)
    cuda.memcpy_htod(self.gpu_coordinates_y, self.host_coordinates_y)
    cuda.memcpy_htod(self.gpu_coordinates_z, self.host_coordinates_z)

    self.gpu_speeds_x = cuda.mem_alloc(self.host_speeds_x.nbytes)
    self.gpu_speeds_y = cuda.mem_alloc(self.host_speeds_y.nbytes)
    self.gpu_speeds_z = cuda.mem_alloc(self.host_speeds_z.nbytes)
    cuda.memcpy_htod(self.gpu_speeds_x, self.host_speeds_x)
    cuda.memcpy_htod(self.gpu_speeds_y, self.host_speeds_y)
    cuda.memcpy_htod(self.gpu_speeds_z, self.host_speeds_z)

    self.gpu_accelerates_x = cuda.mem_alloc(self.host_accelerates_x.nbytes)
    self.gpu_accelerates_y = cuda.mem_alloc(self.host_accelerates_y.nbytes)
    self.gpu_accelerates_z = cuda.mem_alloc(self.host_accelerates_z.nbytes)
    cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
    cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
    cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

    self.gpu_kinetic_energy = cuda.mem_alloc(self.host_kinetic_energy.nbytes)
    cuda.memcpy_htod(self.gpu_kinetic_energy, self.host_kinetic_energy)

    self.gpu_potential_energy = cuda.mem_alloc(self.host_potential_energy.nbytes)
    cuda.memcpy_htod(self.gpu_potential_energy, self.host_potential_energy)
    # PREPARING GPU FUNCTIONS CALL
    self.calc_coordinates_gpu.prepare(
        arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, int32, float32, float32))
    self.calc_speeds_gpu.prepare(
        arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
                   int32, float32))
    self.calc_accelerates_gpu.prepare(
        arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, int32, float32, float32,
                   float32, float32, float32, float32, float32))
    self.calc_kinetic_energy_gpu.prepare(
        arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
                   cuda.DeviceAllocation, cuda.DeviceAllocation,
                   int32, float32))
```

```
# CALCULATION ACCELERATES and KINETIC ENERGY
autoinit.context.synchronize()
self.calc_accelerates_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_accelerates_x, self.gpu_accelerates_y,
    self.gpu_accelerates_z, self.gpu_coordinates_x,
    self.gpu_coordinates_y, self.gpu_coordinates_z,
    self.gpu_potential_energy, self.bodies_amount,
    self.distance_between_bodies, self.sigma,
    self.epsilon, self.dt, self.box_width,
    self.box_height, self.box_depth)
autoinit.context.synchronize()

self.calc_kinetic_energy_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_kinetic_energy, self.gpu_speeds_x,
    self.gpu_speeds_y, self.gpu_speeds_z,
    self.bodies_amount, float32(1.0))
autoinit.context.synchronize()
# TRANSFERRING DATA BACK TO THE HOST
cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
autoinit.context.synchronize()
cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
autoinit.context.synchronize()
# Prepare System Energy Plot
self.prepare_plot_properties()

self.system_energy_list[self.iter_counter] =\
    (sum(self.host_kinetic_energy) + sum(self.host_potential_energy))
self.kinetic_energy_list[self.iter_counter] =\
    sum(self.host_kinetic_energy)
self.potential_energy_list[self.iter_counter] =\
    sum(self.host_potential_energy)
self.iter_counter += 1

# LOCAL TIMER
local_time_start = timer()
while self.iter_counter != self.steps_limit:
    # CALCULATING NEW COORDINATES
    autoinit.context.synchronize()
    self.calc_coordinates_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_coordinates_x, self.gpu_speeds_x,
        self.gpu_accelerates_x, self.bodies_amount,
        self.box_width, self.dt)
    autoinit.context.synchronize()
    self.calc_coordinates_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_coordinates_y, self.gpu_speeds_y,
        self.gpu_accelerates_y, self.bodies_amount,
        self.box_depth, self.dt)
    autoinit.context.synchronize()
    self.calc_coordinates_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_coordinates_z, self.gpu_speeds_z,
        self.gpu_accelerates_z, self.bodies_amount,
        self.box_height, self.dt)
    autoinit.context.synchronize()
    # CALCULATING FIRST PART of SPEEDS
    self.calc_speeds_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_speeds_x, self.gpu_accelerates_x,
        self.bodies_amount, self.dt)
    autoinit.context.synchronize()
    self.calc_speeds_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_speeds_y, self.gpu_accelerates_y,
        self.bodies_amount, self.dt)
    autoinit.context.synchronize()
    self.calc_speeds_gpu.prepared_call(
        self.grid, self.block,
        self.gpu_speeds_z, self.gpu_accelerates_z,
        self.bodies_amount, self.dt)
    autoinit.context.synchronize()
    # CALCULATING NEW ACCELERATES
    cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
    autoinit.context.synchronize()
    cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
    autoinit.context.synchronize()
```

```
        cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_potential_energy, zeros(shape=self.bodies_amount, dtype=float32))
        autoinit.context.synchronize()
        cuda.memcpy_htod(self.gpu_kinetic_energy, zeros(shape=self.bodies_amount, dtype=float32))
        autoinit.context.synchronize()

        self.calc_accelerates_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_accelerates_x, self.gpu_accelerates_y,
            self.gpu_accelerates_z, self.gpu_coordinates_x,
            self.gpu_coordinates_y, self.gpu_coordinates_z,
            self.gpu_potential_energy, self.bodies_amount,
            self.distance_between_bodies, self.sigma,
            self.epsilon, self.dt, self.box_width,
            self.box_height, self.box_depth)
        autoinit.context.synchronize()
        # CALCULATING SECOND PART of SPEEDS
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_x, self.gpu_accelerates_x,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_y, self.gpu_accelerates_y,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        self.calc_speeds_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_speeds_z, self.gpu_accelerates_z,
            self.bodies_amount, self.dt)
        autoinit.context.synchronize()
        # CALCULATING KINETIC ENERGY
        self.calc_kinetic_energy_gpu.prepared_call(
            self.grid, self.block,
            self.gpu_kinetic_energy, self.gpu_speeds_x,
            self.gpu_speeds_y, self.gpu_speeds_z,
            self.bodies_amount, float32(1.0))
        autoinit.context.synchronize()
        # TRANSFERRING DATA BACK TO THE HOST
        cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
        autoinit.context.synchronize()
        # WRITING DATA to the ARRAYS
        self.system_energy_list[self.iter_counter] =\
            (sum(self.host_kinetic_energy) + sum(self.host_potential_energy))
        self.kinetic_energy_list[self.iter_counter] =\
            sum(self.host_kinetic_energy)
        self.potential_energy_list[self.iter_counter] =\
            sum(self.host_potential_energy)
        self.iter_counter += 1
    global_time_end = timer()
    print("Reference Experiment Time:")
    print("\tLocal: {0} s.".format(
        global_time_end - local_time_start))
    print("\tGlobal: {0} s.".format(
        global_time_end - global_time_start))
    return
# --------------------------------------------------

# START GPU EXPERIMENT WITH RADIUS
# --------------------------------------------------
def start_gpu_experiment_with_radius(self, system_size):
    for index in range(len(self.cutting_radius_list)):
        # Start Experiment Timer
        global_time_start = timer()
        # Save New Cutting Radius
        self.current_cutting_radius = \
            float32(self.cutting_radius_list[index])
        # Preparing GPU functions
        self.calc_coordinates_gpu = \
            gpu_calc_coordinates_kernel.get_function("calc_coordinates")
        self.calc_speeds_gpu = \
            gpu_calc_speeds_kernel.get_function("calc_speeds")
        self.calc_accelerates_gpu = \
            gpu_calc_accelerates_with_r_cut_kernel.get_function(
                "calc_accelerates_with_r_cut")
```

```
self.calc_kinetic_energy_gpu = \
    gpu_calc_kinetic_energy.get_function("calc_kinetic_energy")
# Preparing CPU arrays
self.host_coordinates_x = deepcopy(self.host_bodies_coordinates[0])
self.host_coordinates_y = deepcopy(self.host_bodies_coordinates[1])
self.host_coordinates_z = deepcopy(self.host_bodies_coordinates[2])

self.host_speeds_x = deepcopy(self.host_bodies_speeds[0])
self.host_speeds_y = deepcopy(self.host_bodies_speeds[1])
self.host_speeds_z = deepcopy(self.host_bodies_speeds[2])

self.host_accelerates_x = zeros(shape=self.bodies_amount, dtype=float32)
self.host_accelerates_y = zeros(shape=self.bodies_amount, dtype=float32)
self.host_accelerates_z = zeros(shape=self.bodies_amount, dtype=float32)

self.host_kinetic_energy = zeros(shape=self.bodies_amount, dtype=float32)
self.host_potential_energy = zeros(shape=self.bodies_amount, dtype=float32)
# Preparing GPU arrays
self.gpu_coordinates_x = cuda.mem_alloc(self.host_coordinates_x.nbytes)
self.gpu_coordinates_y = cuda.mem_alloc(self.host_coordinates_y.nbytes)
self.gpu_coordinates_z = cuda.mem_alloc(self.host_coordinates_z.nbytes)
cuda.memcpy_htod(self.gpu_coordinates_x, self.host_coordinates_x)
cuda.memcpy_htod(self.gpu_coordinates_y, self.host_coordinates_y)
cuda.memcpy_htod(self.gpu_coordinates_z, self.host_coordinates_z)

self.gpu_speeds_x = cuda.mem_alloc(self.host_speeds_x.nbytes)
self.gpu_speeds_y = cuda.mem_alloc(self.host_speeds_y.nbytes)
self.gpu_speeds_z = cuda.mem_alloc(self.host_speeds_z.nbytes)
cuda.memcpy_htod(self.gpu_speeds_x, self.host_speeds_x)
cuda.memcpy_htod(self.gpu_speeds_y, self.host_speeds_y)
cuda.memcpy_htod(self.gpu_speeds_z, self.host_speeds_z)

self.gpu_accelerates_x = cuda.mem_alloc(self.host_accelerates_x.nbytes)
self.gpu_accelerates_y = cuda.mem_alloc(self.host_accelerates_y.nbytes)
self.gpu_accelerates_z = cuda.mem_alloc(self.host_accelerates_z.nbytes)
cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)

self.gpu_kinetic_energy = cuda.mem_alloc(self.host_kinetic_energy.nbytes)
cuda.memcpy_htod(self.gpu_kinetic_energy, self.host_kinetic_energy)

self.gpu_potential_energy = cuda.mem_alloc(self.host_potential_energy.nbytes)
cuda.memcpy_htod(self.gpu_potential_energy, self.host_potential_energy)
# PREPARING GPU FUNCTIONS CALL
self.calc_coordinates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32, float32))
self.calc_speeds_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
self.calc_accelerates_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, int32, float32,
               float32, float32, float32,
               float32, float32, float32, float32))
self.calc_kinetic_energy_gpu.prepare(
    arg_types=(cuda.DeviceAllocation, cuda.DeviceAllocation,
               cuda.DeviceAllocation, cuda.DeviceAllocation,
               int32, float32))
# CALCULATION ACCELERATES and KINETIC ENERGY
autoinit.context.synchronize()
self.calc_accelerates_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_accelerates_x, self.gpu_accelerates_y,
    self.gpu_accelerates_z, self.gpu_coordinates_x,
    self.gpu_coordinates_y, self.gpu_coordinates_z,
    self.gpu_potential_energy, self.bodies_amount,
    self.distance_between_bodies, self.sigma,
    self.epsilon, self.dt, self.box_width,
    self.box_height, self.box_depth,
    self.current_cutting_radius)
autoinit.context.synchronize()

self.calc_kinetic_energy_gpu.prepared_call(
    self.grid, self.block,
    self.gpu_kinetic_energy, self.gpu_speeds_x,
```

```
            self.gpu_speeds_y, self.gpu_speeds_z,
            self.bodies_amount, float32(1.0))
        autoinit.context.synchronize()
        # TRANSFERRING DATA BACK TO THE HOST
        cuda.memcpy_dtoh(self.host_accelerates_x, self.gpu_accelerates_x)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_accelerates_y, self.gpu_accelerates_y)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_accelerates_z, self.gpu_accelerates_z)
        autoinit.context.synchronize()

        cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
        autoinit.context.synchronize()
        cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
        autoinit.context.synchronize()
        # Prepare System Energy Plot
        self.prepare_plot_properties()

        self.system_energy_list[self.iter_counter] =\
            (sum(self.host_kinetic_energy) + sum(self.host_potential_energy))
        self.kinetic_energy_list[self.iter_counter] =\
            sum(self.host_kinetic_energy)
        self.potential_energy_list[self.iter_counter] =\
            sum(self.host_potential_energy)

        self.iter_counter += 1

        # LOCAL TIMER
        local_time_start = timer()
        while self.iter_counter != self.steps_limit:
            # CALCULATING NEW COORDINATES
            autoinit.context.synchronize()
            self.calc_coordinates_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_coordinates_x, self.gpu_speeds_x,
                self.gpu_accelerates_x, self.bodies_amount,
                self.box_width, self.dt)
            autoinit.context.synchronize()
            self.calc_coordinates_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_coordinates_y, self.gpu_speeds_y,
                self.gpu_accelerates_y, self.bodies_amount,
                self.box_depth, self.dt)
            autoinit.context.synchronize()
            self.calc_coordinates_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_coordinates_z, self.gpu_speeds_z,
                self.gpu_accelerates_z, self.bodies_amount,
                self.box_height, self.dt)
            autoinit.context.synchronize()
            # CALCULATING FIRST PART of SPEEDS
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_x, self.gpu_accelerates_x,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_y, self.gpu_accelerates_y,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_z, self.gpu_accelerates_z,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            # CALCULATING NEW ACCELERATES
            cuda.memcpy_htod(self.gpu_accelerates_x, self.host_accelerates_x)
            autoinit.context.synchronize()
            cuda.memcpy_htod(self.gpu_accelerates_y, self.host_accelerates_y)
            autoinit.context.synchronize()
            cuda.memcpy_htod(self.gpu_accelerates_z, self.host_accelerates_z)
            autoinit.context.synchronize()
            cuda.memcpy_htod(self.gpu_potential_energy, zeros(shape=self.bodies_amount,
dtype=float32))
            autoinit.context.synchronize()
            cuda.memcpy_htod(self.gpu_kinetic_energy, zeros(shape=self.bodies_amount, dtype=float32))
            autoinit.context.synchronize()
```

```
            self.calc_accelerates_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_accelerates_x, self.gpu_accelerates_y,
                self.gpu_accelerates_z, self.gpu_coordinates_x,
                self.gpu_coordinates_y, self.gpu_coordinates_z,
                self.gpu_potential_energy, self.bodies_amount,
                self.distance_between_bodies, self.sigma,
                self.epsilon, self.dt, self.box_width,
                self.box_height, self.box_depth,
                self.current_cutting_radius)
            autoinit.context.synchronize()
            # CALCULATING SECOND PART of SPEEDS
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_x, self.gpu_accelerates_x,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_y, self.gpu_accelerates_y,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            self.calc_speeds_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_speeds_z, self.gpu_accelerates_z,
                self.bodies_amount, self.dt)
            autoinit.context.synchronize()
            # CALCULATING KINETIC ENERGY
            self.calc_kinetic_energy_gpu.prepared_call(
                self.grid, self.block,
                self.gpu_kinetic_energy, self.gpu_speeds_x,
                self.gpu_speeds_y, self.gpu_speeds_z,
                self.bodies_amount, float32(1.0))
            autoinit.context.synchronize()
            # TRANSFERRING DATA BACK TO THE HOST
            cuda.memcpy_dtoh(self.host_kinetic_energy, self.gpu_kinetic_energy)
            autoinit.context.synchronize()
            cuda.memcpy_dtoh(self.host_potential_energy, self.gpu_potential_energy)
            autoinit.context.synchronize()
            # WRITING VALUES to the ARRAYS
            self.system_energy_list[self.iter_counter] =\
                (sum(self.host_kinetic_energy) + sum(self.host_potential_energy))
            self.kinetic_energy_list[self.iter_counter] =\
                sum(self.host_kinetic_energy)
            self.potential_energy_list[self.iter_counter] =\
                sum(self.host_potential_energy)
            self.iter_counter += 1
        global_time_end = timer()
        print("Experiment Time:")
        print("\tLocal: {0} s.".format(
            global_time_end - local_time_start))
        print("\tGlobal: {0} s.".format(
            global_time_end - global_time_start))
        # Updating data
        self.update_data_gpu_experiment_with_cutting_radius(
            system_size=system_size)
    return
# --------------------------------------------------

# UPDATE DATA GPU EXPERIMENT with CUTTING RADIUS
# --------------------------------------------------
def update_data_gpu_experiment_with_cutting_radius(self, system_size, first_try=False):
    # CONSTANTS
    se = 0  # System Energy Plot index
    ke = 1  # Kinetic Energy Plot index
    pe = 2  # Potential Energy Plot index

    self.iter_counter = 0
    # UPDATING BODIES
    # COORDINATES & SPEEDS
    self.host_bodies_coordinates =\
        array(deepcopy(self.global_bodies_coordinates), dtype=float32)
    self.host_bodies_speeds =\
        array(deepcopy(self.global_bodies_speeds), dtype=float32)
    # SYSTEM ENERGY
    average_energy_per_body =\
        sum(self.system_energy_list)/float(self.system_energy_list.shape[0])
    # KINETIC ENERGY
    average_kinetic_energy_per_body =\
```

```
            sum(self.kinetic_energy_list)/float(self.kinetic_energy_list.shape[0])
        # POTENTIAL ENERGY
        average_potential_energy_per_body =\
            sum(self.potential_energy_list)/float(self.potential_energy_list.shape[0])
        # SAVING RESULTS
        if first_try:
            # PLOTS
            self.subplots[0].plot(self.dt_list,
                                  self.system_energy_list,
                                  color="red", marker="."
                                  )
            self.subplots[1].plot(self.dt_list,
                                  self.kinetic_energy_list,
                                  color="blue", marker="."
                                  )
            self.subplots[2].plot(self.dt_list,
                                  self.potential_energy_list,
                                  color="green", marker="."
                                  )

            self.fig_plot.savefig("Tables\\{0}\\Reference_plot_{0}.png".format(system_size))
            # VALUES
            self.reference_average_energy =\
                average_energy_per_body
            self.reference_average_kinetic_energy =\
                average_kinetic_energy_per_body
            self.reference_average_potential_energy =\
                average_potential_energy_per_body

            self.experiment_average_energy_list = list()
            self.experiment_average_kinetic_energy_list = list()
            self.experiment_average_potential_energy_list = list()
        else:
            # PLOTS
            self.subplots[0].plot(self.dt_list,
                                  self.system_energy_list,
                                  color="red", marker=".")
            self.subplots[1].plot(self.dt_list,
                                  self.kinetic_energy_list,
                                  color="blue", marker=".")
            self.subplots[2].plot(self.dt_list,
                                  self.potential_energy_list,
                                  color="green", marker=".")

            self.fig_plot.savefig(
                "Tables\\{0}\\Experiment_{0}_plot_R_{1}.png".format(
                    system_size,
                    self.current_cutting_radius))
            # VALUES
            self.experiment_average_energy_list.append(
                average_energy_per_body)
            self.experiment_average_kinetic_energy_list.append(
                average_kinetic_energy_per_body)
            self.experiment_average_potential_energy_list.append(
                average_potential_energy_per_body)
        # CLEANING PLOTS
        plt.close(self.fig_plot)
    # --------------------------------------------------
```

ДОДАТОК Г

Лістинг модуля *calc_coordinates_cpu_1ker* для розрахунку координат для CPU

```
# -*- coding: UTF-8 -*-
from numpy import vectorize, apply_along_axis


def correct_position(coordinates, width, height, depth):
    function_x = vectorize(lambda x: width + x if x < 0 else x - width if x > width else x)
    function_y = vectorize(lambda y: depth + y if y < 0 else y - depth if y > depth else y)
    function_z = vectorize(lambda z: height + z if z < 0 else z - height if z > height else z)
    # Step through left&Right borders
    coordinates[0] = apply_along_axis(func1d=function_x, axis=0, arr=coordinates[0])
    # Step through Front&Back borders
    coordinates[1] = apply_along_axis(func1d=function_y, axis=0, arr=coordinates[1])
    # Step through Top&Bottom borders
    coordinates[2] = apply_along_axis(func1d=function_z, axis=0, arr=coordinates[2])
    return coordinates


def calc_coordinates_cpu_1ker(coordinates, speeds, accelerates, width, height, depth, dt):
    coordinates += speeds*dt + 0.5*dt*dt*accelerates
    coordinates = correct_position(coordinates=coordinates, width=width, height=height, depth=depth)
    return coordinates
```

ДОДАТОК Ґ

Лістинг модуля *calc_speeds_cpu_1ker* для розрахунку швидкостей на CPU

```
# -*- coding: UTF-8 -*-


def calc_speeds_cpu_1ker(speeds, accelerates, dt):
    return speeds + 0.5*accelerates*dt
```

ДОДАТОК Д

Лістинг модуля *calc_accelerate_cpu_1ker* для розрахунку прискорень на CPU

```
# -*- coding: UTF-8 -*-
from numpy import float32, zeros_like
from MSD_functionality.CPU_1Ker.calc_distance import calc_distance_components_mtx, calc_distances_mtx
from MSD_functionality.CPU_1Ker.calc_force import calc_forces, calc_forces_components
from MSD_functionality.CPU_1Ker.calc_system_energy import calc_potential_energy


def calc_accelerate_v01(bodies_coordinates, bodies_amount, bodies_min_distance,
                        sigma, epsilon, width, height, depth
                        ):
    accelerates = zeros_like(bodies_coordinates)
    potential_energy_accumulator = float32(0.0)
    for i in range(bodies_amount):
        distances_components = calc_distance_components_mtx(coordinates=bodies_coordinates,
                                                            column_index=i,
                                                            width=width,
                                                            height=height,
                                                            depth=depth
                                                            )
        distances = calc_distances_mtx(distances_components=distances_components)
        distances_between_faces = abs(distances - bodies_min_distance)
        distances[i] = float32(1)
        distances_between_faces[i] = float32(1)
        forces = calc_forces(distances=distances_between_faces, epsilon=epsilon, sigma=sigma)
        forces_components = calc_forces_components(forces=forces,
                                                   distances=distances,
                                                   distances_components=distances_components
                                                   )
        accelerates[0][i] += sum(forces_components[0])
        accelerates[1][i] += sum(forces_components[1])
        accelerates[2][i] += sum(forces_components[2])
        accelerates -= forces_components

        # Calculating Potential Energy
        potential_energy_components = calc_potential_energy(epsilon=epsilon,
                                                            sigma=sigma,
                                                            distances=distances_between_faces
                                                            )
        potential_energy_components[i] = float32(0)
        potential_energy_accumulator += sum(potential_energy_components)
    return accelerates, potential_energy_accumulator
```

ДОДАТОК Е

Лістинг модуля *calc_force* для розрахунку сили на CPU

```
# -*- coding: UTF-8 -*-


def calc_forces(distances, epsilon, sigma):
    return 24*(epsilon/sigma)*(2*pow(sigma/distances, 13) - pow(sigma/distances, 7))


def calc_forces_components(forces, distances, distances_components):
    return forces*distances_components/distances
```

ДОДАТОК Є

Лістинг модуля *calc_distance* для розрахунку відстаней на CPU

```
# -*- coding: UTF-8 -*-


from numpy import sign, sqrt, arange, delete, transpose, \
    apply_along_axis, vectorize, ones, float32, dot, matmul


def calc_range_components(x1, y1, x2, y2, z1, z2):
    return x1 - x2, y1 - y2, z1 - z2


def correct_range_components(rx, ry, rz, width, depth, height):
    return rx - sign(rx)*width if abs(rx) > width*0.5 else rx, \
           ry - sign(ry)*depth if abs(ry) > depth*0.5 else ry, \
           rz - sign(rz)*height if abs(rz) > height*0.5 else rz


def calc_distance(rx, ry, rz):
    return sqrt(rx**2 + ry**2 + rz**2)


def calc_distance_components_mtx(coordinates, column_index, width, height, depth):

    correct_x = vectorize(lambda rx: rx - sign(rx)*width if abs(rx) > width*0.5 else rx)
    correct_y = vectorize(lambda ry: ry - sign(ry)*depth if abs(ry) > depth*0.5 else ry)
    correct_z = vectorize(lambda rz: rz - sign(rz)*height if abs(rz) > height*0.5 else rz)

    range_components = transpose((coordinates[:, column_index]) - transpose(coordinates))

    range_components[0] = apply_along_axis(func1d=correct_x,
                                           axis=0,
                                           arr=range_components[0]
                                           )
    range_components[1] = apply_along_axis(func1d=correct_y,
                                           axis=0,
                                           arr=range_components[1]
                                           )
    range_components[2] = apply_along_axis(func1d=correct_z,
                                           axis=0,
                                           arr=range_components[2]
                                           )
    return range_components


def calc_distances_mtx(distances_components):
    distances = sqrt(matmul(ones(shape=3, dtype=float32), distances_components*distances_components))
    return distances
```

ДОДАТОК Ж

Лістинг модуля *calc_system_energy* для розрахунку енергій на CPU

```
# -*- coding: UTF-8 -*-
from numpy import float32, ones, matmul


def calc_kinetic_energy(velocities, m=1):
    return (m*(matmul(ones(shape=3, dtype=float32), velocities*velocities)))*0.5/velocities.shape[1]


def calc_potential_energy(epsilon, sigma, distances):
    return 4*epsilon*(pow(sigma/distances, 12) - pow(sigma/distances, 6))/distances.shape[0]
```

ДОДАТОК З

Лістинг модуля *GPU_calc_coords* для розрахунку координат на GPU

```
# -*- coding: UTF-8 -*-
from pycuda import autoinit
from pycuda.compiler import SourceModule


gpu_calc_coordinates_kernel = SourceModule("""
    # include <pycuda-helpers.hpp>
    # include <math.h>
    __device__ float correct_coordinate(float coordinate, float box_parameter) {
        return fmod(coordinate + box_parameter, box_parameter);
    }

    __global__ void calc_coordinates(float *coordinates, float *speeds, float *accelerates, \
                                     int amount, float box_parameter, float dt) {
        for (int tid = threadIdx.x + blockIdx.x*blockDim.x; tid < amount; tid += blockDim.x*gridDim.x) {
            coordinates[tid] += speeds[tid]*dt + 0.5*accelerates[tid]*dt*dt;
            __syncthreads();
            coordinates[tid] = correct_coordinate(coordinates[tid], box_parameter);
            __syncthreads();
        }
    }
""")
```

ДОДАТОК И

Лістинг модуля *GPU_calc_speeds* для розрахунку швидкостей на GPU

```
# -*- coding: UTF-8 -*-
from pycuda import autoinit
from pycuda.compiler import SourceModule


gpu_calc_speeds_kernel = SourceModule("""
    __global__ void calc_speeds(float *speeds, float *accelerates, int amount, float dt) {
        for (int tid = threadIdx.x + blockIdx.x*blockDim.x; tid < amount; tid += blockDim.x*gridDim.x) {
            speeds[tid] += 0.5*accelerates[tid]*dt;
            __syncthreads();
        }
    }
""")
```

ДОДАТОК І

Лістинг модуля *GPU_calc_accelerates* для розрахунку прискорень на GPU

```
# -*- coding: UTF-8 -*-
from pycuda import autoinit
from pycuda.compiler import SourceModule

gpu_calc_accelerates_kernel = SourceModule("""
    # include <pycuda-helpers.hpp>
    # include <math.h>
    __device__ float calc_dist_component(float coordinate1, float coordinate2, float box_parameter) {
        float dist_component = coordinate1 - coordinate2;
            if (abs(dist_component) >= 0.5*box_parameter) {
                dist_component -= copysignf(1.0, dist_component)*box_parameter;
            }
        return dist_component;
    }

    __device__ float calc_potential_energy(float epsilon, float sigma, float body_distance) {
        return 4*epsilon*(pow(sigma/body_distance, 12) - pow(sigma/body_distance, 6));
    }

    __global__ void calc_accelerates(float *accelerates_x, float *accelerates_y, float *accelerates_z,
                                     float *coordinates_x, float *coordinates_y, float *coordinates_z,
                                     float *potential_energy, int amount, float b_min_distance,
                                     float sigma, float epsilon, float dt, float width, float height,
float depth) {
        float dist_x = 0.0;
        float dist_y = 0.0;
        float dist_z = 0.0;
        float center_distance = 0.0;
        float faces_distance = 0.0;
        float force = 0.0;
        for (int tid_out = threadIdx.x + blockIdx.x*blockDim.x; tid_out < amount - 1; tid_out +=
blockDim.x*gridDim.x) {
            for (int tid_in = tid_out + 1; tid_in < amount; tid_in += 1){
                dist_x = calc_dist_component(coordinates_x[tid_out], coordinates_x[tid_in], width);
                __syncthreads();
                dist_y = calc_dist_component(coordinates_y[tid_out], coordinates_y[tid_in], depth);
                __syncthreads();
                dist_z = calc_dist_component(coordinates_z[tid_out], coordinates_z[tid_in], height);
                __syncthreads();
                center_distance = sqrt(pow(dist_x, 2) + pow(dist_y, 2) + pow(dist_z, 2));
                __syncthreads();
                faces_distance = abs(center_distance - b_min_distance);
                __syncthreads();
                force = 24*(sigma/epsilon)*(2*(pow(sigma/faces_distance, 13)) - pow(sigma/faces_distance,
7));
                __syncthreads();
                accelerates_x[tid_out] += force*(dist_x/center_distance);
                __syncthreads();
                accelerates_y[tid_out] += force*(dist_y/center_distance);
                __syncthreads();
                accelerates_z[tid_out] += force*(dist_z/center_distance);
                __syncthreads();

                accelerates_x[tid_in] -= force*(dist_x/center_distance);
                __syncthreads();
                accelerates_y[tid_in] -= force*(dist_y/center_distance);
                __syncthreads();
                accelerates_z[tid_in] -= force*(dist_z/center_distance);
                __syncthreads();
                // Calculate potential energy between couple of bodies
                float potential_energy_value =\
                    calc_potential_energy(epsilon, sigma, faces_distance)/amount;
                potential_energy[tid_out] += potential_energy_value;
                __syncthreads();
            }
        }
    }
""")
```

ДОДАТОК Ї

Лістинг модуля *GPU_calc_kinetic_energy* для розрахунку кінетичної енергії на GPU

```
# -*- coding: UTF-8 -*-
from pycuda import autoinit
from pycuda.compiler import SourceModule

gpu_calc_kinetic_energy = SourceModule("""
    __global__ void calc_kinetic_energy(float *kinetic_energy, float *speeds_x, float *speeds_y, float
*speeds_z,
                                        int amount, float m) {
        for(int tid = threadIdx.x + blockIdx.x*blockDim.x; tid < amount; tid += blockDim.x*gridDim.x) {
            kinetic_energy[tid] = m*0.5*(pow(speeds_x[tid], 2) + pow(speeds_y[tid], 2) +
pow(speeds_z[tid], 2))/amount;
            __syncthreads();
        }
    }
""")
```

ДОДАТОК Й

Лістинг модуля *GPU_calc_accelerates_with_rcut.py* для розрахунку прискорень на GPU з використанням радіусу обмеження

```
# -*- coding: UTF-8 -*-
from pycuda import autoinit
from pycuda.compiler import SourceModule

gpu_calc_accelerates_with_r_cut_kernel = SourceModule("""
    # include <pycuda-helpers.hpp>
    # include <math.h>
    __device__ float calc_dist_component(float coordinate1, float coordinate2, float box_parameter) {
        float dist_component = coordinate1 - coordinate2;
            if (abs(dist_component) >= 0.5*box_parameter) {
                dist_component -= copysignf(1.0, dist_component)*box_parameter;
            }
        return dist_component;
    }
    __device__ float calc_potential_energy(float epsilon, float sigma, float body_distance) {
        return 4*epsilon*(pow(sigma/body_distance, 12) - pow(sigma/body_distance, 6));
    }
    __global__ void calc_accelerates_with_r_cut(float *accelerates_x, float *accelerates_y, float
*accelerates_z,
                                                float *coordinates_x, float *coordinates_y, float
*coordinates_z,
                                                float *potential_energy, int amount, float b_min_distance,
                                                float sigma, float epsilon, float dt,
                                                float width, float height, float depth, float r_cut) {
        float dist_x = 0.0;
        float dist_y = 0.0;
        float dist_z = 0.0;
        float center_distance = 0.0;
        float faces_distance = 0.0;
        float force = 0.0;
        for (int tid_out = threadIdx.x + blockIdx.x*blockDim.x; tid_out < amount - 1; tid_out +=
blockDim.x*gridDim.x) {
            //potential_energy[tid_out] = 0.0;
            //__syncthreads();
            for (int tid_in = tid_out + 1; tid_in < amount; tid_in += 1){
                dist_x = calc_dist_component(coordinates_x[tid_out], coordinates_x[tid_in], width);
                __syncthreads();
                dist_y = calc_dist_component(coordinates_y[tid_out], coordinates_y[tid_in], depth);
                __syncthreads();
                dist_z = calc_dist_component(coordinates_z[tid_out], coordinates_z[tid_in], height);
                __syncthreads();
                center_distance = sqrt(pow(dist_x, 2) + pow(dist_y, 2) + pow(dist_z, 2));
                __syncthreads();
                faces_distance = abs(center_distance - b_min_distance);
                __syncthreads();
                if (faces_distance < r_cut) {
                    force = 24*(sigma/epsilon)*(2*(pow(sigma/faces_distance, 13)) -
pow(sigma/faces_distance, 7));
                    __syncthreads();
                    accelerates_x[tid_out] += force*(dist_x/center_distance);
                    __syncthreads();
                    accelerates_y[tid_out] += force*(dist_y/center_distance);
                    __syncthreads();
                    accelerates_z[tid_out] += force*(dist_z/center_distance);
                    __syncthreads();
                    accelerates_x[tid_in] -= force*(dist_x/center_distance);
                    __syncthreads();
                    accelerates_y[tid_in] -= force*(dist_y/center_distance);
                    __syncthreads();
                    accelerates_z[tid_in] -= force*(dist_z/center_distance);
                    __syncthreads();
                    // Calculate potential energy between couple of bodies
                    potential_energy[tid_out] +=  calc_potential_energy(epsilon, sigma,
faces_distance)/amount;
                    __syncthreads();
                }
            }
        }
    }
""")
```

ДОДАТОК К

Лістинг програми для побудови графіків залежності усередненої енергії від радіусу обмеження

```python
# -*- coding: utf-8 -*-
# IMPORTING LIBRARIES
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# LOADING CSV-FILES with DATA
# Constants
system_sizes = [1_000, 10_000]

table_name = "System {0} with Cutting Radius.csv"
table_shop = [None for _ in range(len(system_sizes))]
counter = 0

for index in range(len(table_shop)):
    table_shop[index] = pd.read_csv(
        table_name.format(system_sizes[index]),
        sep=";",
        header=[0, 1])
# SHOW TABLE COLUMNS
for index in range(len(table_shop[0].columns)):
    print("{0}: {1}".format(index, table_shop[0].columns[index]))
# COLUMN INDEXES for TABLES
column_indexes = [(2, 3, 4),
                  (2, 5, 6),
                  (2, 7, 8)]
# CREATING PLOTS
# Constants
plots_amount = 3
plot_storage = [None for _ in range(plots_amount)]
plot_titles = ["Average System Energy per Body(Radius)",
               "Average Kinetic Energy per Body(Radius)",
               "Average Potential Energy per Body(Radius)"]
y_labels = ["ASEpB",
            "APEpB",
            "AKEpB"]
fig_width = 20  # 200 - for long plot; 20 - for short plot
fig_height = 15
fig_dpi = 150
f_idx = 0   # figure index
ax_idx = 1  # axes index
title_font_dict = {
    "fontname": "Times New Roman",
    "fontsize": 12,
    "family": "serif"}
axes_font_dict = {
    "fontname": "Times New Roman",
    "fontsize": 11,
    "family": "serif"}  # 5 - for long plot 11 - for short plot
colors = [(1.0, 0.0, 0.0),
          (0.0, 0.3, 0.0),
          (0.0, 0.0, 0.9)]
for shelf_index in range(len(system_sizes)):
    shelf = plt.subplots(plots_amount, 1)
    for plot_index in range(plots_amount):
        # FIGURE PROPERTIES
        shelf[f_idx].set_figwidth(fig_width)
        shelf[f_idx].set_figheight(fig_height)
        shelf[f_idx].set_dpi(fig_dpi)
        # AXES PROPERTIES
        # Title
        shelf[ax_idx][plot_index].set_title(
            plot_titles[plot_index],
            fontdict=title_font_dict)
        # X axes preparing
        # values
        step = 0.1
        x_axes_values =\
            list(map(lambda el: round(el, 3),
```

```
                 np.arange(0,
                           max(table_shop[shelf_index][
                                   table_shop[
                                       shelf_index].columns[
                                       column_indexes[plot_index][0]]].values) + step,
                           step)))
    shelf[ax_idx][plot_index].set_xlabel(
        "Radius",
        fontdict=axes_font_dict)
    # shelf[ax_idx][plot_index].set_xticks(x_axes_values)  # for long plot
    # shelf[ax_idx][plot_index].set_xticklabels(x_axes_values)  # for long plot
    plt.setp(
        shelf[ax_idx][plot_index].get_xticklabels(),
        "fontname",
        axes_font_dict["fontname"])
    plt.setp(
        shelf[ax_idx][plot_index].get_xticklabels(),
        "fontsize",
        axes_font_dict["fontsize"])
    plt.setp(
        shelf[ax_idx][plot_index].get_xticklabels(),
        "family",
        axes_font_dict["family"])
    # Y axes preparing
    shelf[ax_idx][plot_index].set_ylabel(
        y_labels[plot_index],
        fontdict=axes_font_dict)
    plt.setp(
        shelf[ax_idx][plot_index].get_yticklabels(),
        "fontname",
        axes_font_dict["fontname"])
    plt.setp(
        shelf[ax_idx][plot_index].get_yticklabels(),
        "fontsize",
        axes_font_dict["fontsize"])
    plt.setp(
        shelf[ax_idx][plot_index].get_yticklabels(),
        "family",
        axes_font_dict["family"])
    # PLOTS
    # Average System Energy per Body
    if plot_index < plots_amount:
        shelf[ax_idx][plot_index].plot(
            table_shop[shelf_index][table_shop[shelf_index].columns[column_indexes[plot_index][0]]],
            table_shop[shelf_index][table_shop[shelf_index].columns[column_indexes[plot_index][1]]],
            color=colors[plot_index])
        # Reference values
        shelf[ax_idx][plot_index].plot(
            table_shop[shelf_index][table_shop[shelf_index].columns[column_indexes[plot_index][0]]],
            table_shop[shelf_index][table_shop[shelf_index].columns[column_indexes[plot_index][2]]],
            linestyle="--", color=(0.4, 0.3, 0.1))
    # SHOW GRID
    shelf[ax_idx][plot_index].grid()
    # SAVING RESULTS on SHELF
    plot_storage[plot_index] = shelf
# SAVING PLOTS
shelf[f_idx].savefig("Система_{0}_Контейнер_.png".format(
    system_sizes[shelf_index]))
```

ДОДАТОК Л

Лістинг функції, що будує таблиці для побудови графіків залежності усередненої енергії від радіусу обмеження

```
# coding=utf-8
from numpy import float64, array
from pandas import DataFrame, ExcelWriter


def create_radius_table(bodies_amount,
                        radius_list,
                        experiment_average_energy_list,
                        reference_average_energy,
                        experiment_average_kinetic_energy_list,
                        reference_average_kinetic_energy,
                        experiment_average_potential_energy_list,
                        reference_average_potential_energy,
                        directory_name=""):

    # DATA PREPARING
    data_for_table =\
        [[bodies_amount,
          radius_list[index],
          experiment_average_energy_list[index],
          reference_average_energy,
          experiment_average_kinetic_energy_list[index],
          reference_average_kinetic_energy,
          experiment_average_potential_energy_list[index],
          reference_average_potential_energy] for index in range(len(radius_list))]

    # TABLE CREATION
    table = DataFrame(
        columns=[["",
                  "",
                  "З РАДІУСОМ",
                  "БЕЗ РАДІУСУ",
                  "З РАДІУСОМ",
                  "БЕЗ РАДІУСУ",
                  "З РАДІУСОМ",
                  "БЕЗ РАДІУСУ"],
                 ["Кількість частинок",
                  "Радіус",
                  "Середня енергія на частинку",
                  "Середня енергія на частинку",
                  "Середня кінетична енергія на частинку",
                  "Середня кінетична енергія на частинку",
                  "Середня потенціальна енергія на частинку",
                  "Середня потенціальна енергія на частинку"]],
        data=array(array(data_for_table)),
        dtype=float64)

    # SAVING TABLES
    table_name = "System {0} with Cutting Radius".format(bodies_amount)
    table.to_csv(
        path_or_buf=directory_name + table_name + ".csv", sep=";")
    table.to_excel(directory_name + table_name + ".xlsx")
    return
```