Міністерство освіти і науки України
Кам’янець-Подільський національний університет імені Івана Огієнка
Фізико-математичний факультет
Кафедра комп’ютерних наук

Дипломна робота
бакалавра
з теми: **“КОМП’ЮТЕРНА ВІЗУАЛІЗАЦІЯ ДИНАМІКИ МЕХАНІЧНОЇ СИСТЕМИ”**

Виконав: студент 4 курсу, KN1-B18 групи
спеціальності 122 Комп’ютерні науки
Задворний Антон Олексійович

Керівник: Федорчук Володимир Аналолійович
доктор технічних наук, професор

Кам’янець-Подільський, 2022

ЗМІСТ

ВСТУП

Нас оточує різноманітний і дивовижний світ механіки, який є досконало вивченим. Нам залишається лише візуалізувати взаємодію частинок, врахувавши при цьому встановлені закономірності. Серед багатьох механічних систем можливість візуалізації руху викликали кульки, з’єднані пружинами.

Системи, засновані на вивченні руху набору кульок, з'єднаних пружинами (так звані мас-пружинні системи), дуже активно використовуються в різних галузях сучасного математичного моделювання, постійно ставлячи перед їх дослідниками нові завдання. Важливою перевагою цих моделей є можливість продемонструвати основні принципи математичного моделювання: заміна складного явища або процесу набором простих схем, які потім уточняються, розвиваються і, врешті-решт, дозволяють досягти потрібного рівня адекватності.

Прикладом такого підходу є розгляд ієрархічного ланцюжка моделей системи кулька-пружина у [23]. Ці моделі виходять одна з іншої при послідовній відмові від припущень, які ідеалізують об'єкт, що вивчається: вивчаються різні варіанти діючих на систему зовнішніх сил, змінюються точки кріплення пружини і властивості закріплення, приймається до уваги сили тертя різної природи а також нелінійність властивостей пружин. В одних випадках такі ускладнення не вносять нічого нового у поведінку системи, в інших − її властивості змінюються істотно. Шлях «від простого до складного» дає можливість поетапно вивчати все більш реалістичні моделі та порівнювати їх властивості.

Процес побудови математичної моделі дає змогу досконаліше проаналізувати та зрозуміти характеристики досліджуваного об’єкту для створення комп’ютерної візуалізації його поведінки.

Об’єктом дослідження є пружинна модель твердого тіла.

Предметом дослідження є бібліотека VPython для створення найпростіших 3D-моделей.

Мета: дослідити методи моделювання динаміки механічних систем для відображення їх динаміки у вигляді комп’ютерної анімації.

Завдання дослідження полягає у розробці комп'ютерної візуалізації механічної системи.

Робота складається з чотирьох розділів. Теоретичні аспекти дослідження розкриті у розділах 1 та 2. У розділі 1 зроблено огляд сфер застосування мас-пружинних систем. Розділ 2 містить три підрозділи, які розкривають зміст поставленої задачі, методи її математичного вирішення та реалізацію розробленої математичної моделі за допомогою бібліотеки scipy середовища VPython, що є предметом дослідження в даній роботі. Практична реалізація завдання детально описана у розділах 3 та 4. Розділ 3 присвячений візуалізації пружинної моделі твердого тіла, містить три підрозділи, в яких формулюється завдання, описуються класи та бібліотека VPython. Розділ 4 поділений на два підрозділи в яких описується робота створених програм ShapeEditor та SpringModel. У додатках А-Д наведено код створених класів та програм.

РОЗДІЛ 1

ОСОБЛИВОСТІ МАС-ПРУЖИННИХ СИСТЕМ ТА СФЕРИ ЇХ ЗАСТОСУВАННЯ

Використання мас-пружинних моделей деформованих твердих та газоподібних тіл має давню історію. Цей підхід дозволяє розраховувати характеристики стержнів, пластин та навіть багатошарових тіл, використовуючи ланцюжки з N зв'язаних коливальних ланок. Аналогії між такими ланцюжками та поведінкою електричних коливальних контурів призвели до того, що ці системи використовуються під час моделювання хвильових явищ в радіоелектроніці, акустиці, оптиці з одного боку та застосування методики розрахунку електронних схем для задач, пов'язаних із мас-пружинними моделями [25].

У 1989 році Р. Блікхан [1] запропонував просту бігову модель та модель стрибків, заснованих на поведінці невагомої пружини та приєднаної до неї ваги. Центр ваги людини представлено як єдина вага, прикріплена до пружини, і яка контактує із землею при кожному поштовху. Біг, таким чином, складається із фази «політ», тобто руху ваги в полі сили тяжіння, і фази зіткнення із землею, коли пружина пружно деформується і починає наповнюватися енергією, а далі розпрямляється і знову запускає вагу у політ. Не дивлячись на простоту, ця модель змогла вдало описати якісну залежність механічних параметрів, що характеризують біг та стрибки людей, від швидкості. Викликаючи багато суперечок, вона, тим не менше, активно використовується в сучасній біотехнології бігу.

Ще одним прикладом використання мас-пружинних систем в біотехніці є метод моделювання альтернативний методу кінцевих елементів. Заміна біологічних тканин набором ваги з'єднаних між собою пружин (рис.1.1), отримала поширення при моделюванні результатів хірургічних операції, в тих випадках, коли швидкість розрахунків є вирішальною. Наприклад, такі моделі використовуються для опису властивостей шкіри та жирової тканини [26].

Запропонований підхід можна використовувати для моделювання розтягнутих тканин під час введення під них імплантів (для покращення зовнішнього вигляду пацієнта) або експандерів (для реконструктивних цілей, тобто вирощування донорських тканин). Ціль моделювання – допомогти хірургам зробити правильний вибір імплантат або розрахунок необхідної кількості донорських клітин.

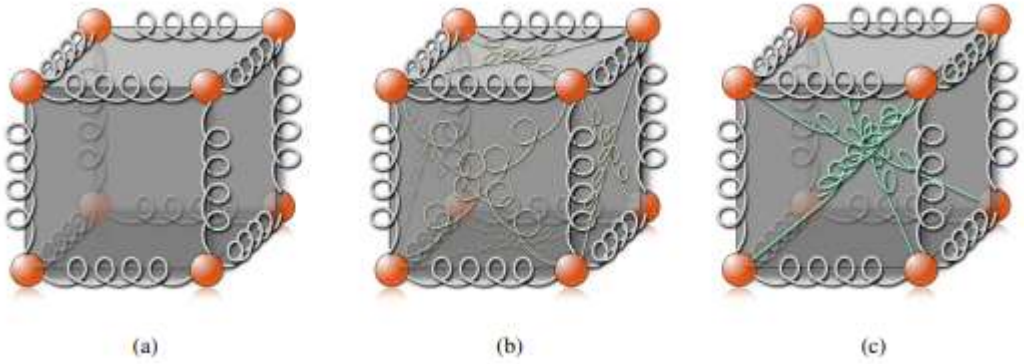

Рис.1.1 Приклад мас-пружинних моделей (малюнок з [9] з різною топологією пружинної мережі: структурні пружини (a), структурні та поверхневі пружини (b), структурні та діагональні пружини (c). Для моделювання геометрії і механіки деформуючого об’єкту такі елементи об’єднуються в трьохвимірну мережу.

Цей підхід використовувався також для моделювання механіки серцевої діяльності [17]. Суцільне середовище замінюється просторовою сіткою тетраедрів, в вузлах якої знаходяться зосереджені маси. Властивості матеріалу при цьому визначаються типом сітки і функцією потенціальної енергії (властивостями пружини). Авторам вдалося розробити високопродуктивний програмний комплекс для аналізу електромеханічної поведінки лівого шлуночка серця людини.

У системах віртуальної реальності часто потрібно моделювати динаміку систем тіл, частина з яких пов'язана між собою пружинами або підпружинними з'єднаннями. Прикладами можуть бути ресори підвіски колісного робота, які пом'якшують удар та поштовхи під час переміщення по нерівній поверхні, щеплення промислового робота з пружиною, яке дозволяє брати крихкі предмети, двері з пружиною, які закриваються автоматично.

Таким чином, розглянута задача розробки швидких і абсолютно стійких методів моделювання динаміки систем тіл за наявності пружинних та підпружинних з'єднань між деякими з них є дуже актуальною.

Цікаве застосування мас-пружинні моделі знайшли в задачах про квазірозгортки поверхонь, тобто такі відображення поверхні на площину, при яких довжина ліній, або кути між ними можуть спотворюватися. Наприклад, в швейній промисловості такі задачі виникають в процесі автоматизованого проектування лекал одягу та інших тканинних виробів. Алгоритм розв’язку [29] полягає в заміні тканини чи поверхні набором з’єднувальних пружинних мас, а процес створення розгортки полягає в моделюванні падіння цих мас на площину. Властивості пружини, або те саме, енергія системи, визначається на основі інформації про властивості реальної тканини.

На завершення, як вказано у [9] можна перерахувати слабкі місця цих моделей, які обмежують їх використання в фізичному моделюванні:

- У порівнянні з моделями, які засновані на теорії пружності, такими як методи кінцевих елементів чи кінцевих різниць, більшість мас-пружинних систем не може гарантувати задану точність. В більшості випадків зменшення кроку сітки не призводить до збіжності до точного вирішення початкової задачі.
- Поведінка цих систем надто залежить від топології та розширення сітки. При зміні сітки результат моделювання може кардинально відрізнятися від початкового.
- Вибір функцій, які описують властивості пружини, і визначення параметрів цих функції – дуже важкий процес, який залежить від області застосування моделі.
- Для низки випадків важко підібрати маси так, щоб властивості були однорідними.
- При використані звичайних мас-пружинних систем важко створити та проконтролювати такі властивості матеріалу, як ізотропність або анізотропія потрібного типу.
- Багато біологічних тканин в процесі деформування не змінюють свій об’єм.

Звичайні мас-пружинні системи не можуть забезпечити умови сталості об’єму для модельованого об’єкта.

Отже, у даному розділі розкрито актуальність застосування мас-пружинних систем та актуальність комп’ютерної візуалізації такої механічної системи.

РОЗДІЛ 2

ЧИСЕЛЬНІ МЕТОДИ РОЗВ'ЯЗУВАННЯ ЗАДАЧІ КОШІ

Звичайні диференціальні рівняння та їх системи зустрічаються у різних задачах математичного моделювання. Вони використовуються для опису процесів коливань, росту популяцій, радіоактивного розпаду, кінетики хімічних реакцій, динаміки взаємодії матеріальних тіл та багатьох інших. Задачі динаміки – один з найважливіших класів фізичної теорії; з точки зору математики вони зводяться до задач Коші, які ще називають початковими задачами (initial value problems, IVP): за відомими в даний момент значеннями функції (та, в залежності від порядку рівняння, кількох її похідних) потрібно побудувати розв'язок у певному проміжку часу, починаючи із даного моменту. У більшості випадків подібні задачі не мають аналітичного розв'язку і для їх дослідження активно використовуються чисельні методи.

Чисельні методи дозволяють отримати розв'язок аналітичних моделей, для яких застосування аналітичних методів неможливо або недоцільно. Розв'язання чисельними методами здійснюється для конкретних даних і має додаткову похибку.

2.1. Постановка задачі

Постановка задачі виглядає таким чином: дано диференціальне рівняння виду

$$y^{(n)}(t) = F\left(t, y(t), y'(t), \dots, y^{(n-1)}(t)\right). \tag{2.1}$$

Потрібно знайти його частковий розв'язок на відрізку $[t_0, T]$, що задовольняє умови

$$y(t_0) = y_0, y'(t_0) = y_1, \dots, y^{(n-1)}(t_0) = y_{n-1}. \tag{2.2}$$

В (2.1), (2.2) позначення $y^{(k)}(t)$ використано для k-ї похідної функції y(t):

$$y^{(k)}(t) \equiv \frac{d^k y}{dt^k}$$

Не будемо обговорювати питання існування такого розв'язку, будемо вважати, що розв'язок існує та має потрібний ступінь гладкості.

Іншим варіантом формулювання задачі (2.1), (2.2) є її запис у термінах системи рівнянь першого порядку. Якщо ввести позначення

$$y(t) = y_0(t), y'(t) = y_1(t), \dots, y^{(n-1)}(t) = y_{n-1}(t),$$

замість (2.1) та (2.2) отримаємо відповідно

$$\begin{cases} y_0' = y_1(t), \\ y_1' = y_2(t), \\ \dots \\ y_{n-2}' = y_{n-1}(t), \\ y_{n-1}'(t) = F(t, y_0(t), y_1(t), \dots, y_{n-1}(t)) \end{cases} \tag{2.3}$$

та

$$y_0(t_0) = y_0, y_1(t_0) = y_1, \dots, y_{n-1}(t_0) = y_{n-1} \tag{2.4}$$

Нарешті, обидві форми задачі Коші часто записуються у векторному форматі. Так, функцію $y(t)$ в (2.1) можна вважати вектор-функцією. При цьому (2.1) стає не рівнянням, а системою рівнянь, але постановка задачі находження її розв'язку за заданими початковими даними зберігається. Наприклад, для опису руху матеріальної частинки масою m під дією сили $\vec{F}$ необхідно розв'язати диференціальне рівняння

$$m\frac{d^2\vec{r}}{dt^2} = \vec{F},$$

де $\vec{r}(t) = \big(x(t), y(t), z(t)\big)$ – радіус-вектор частинки, а у початковий момент відомо значення цього вектора $\overrightarrow{r(t_0)} = \vec{r}_0$ та його похідної за часом (швидкості частинки) $\vec{r}' = \vec{v}_0$.

2.2. Метод Ейлера та його модифікації

Оскільки нас будуть цікавити задачі Коші, пов'язані з динамікою матеріальних точок, при описі методів обмежимося випадком одновимірного руху матеріальної точки, положення якої визначається її ординатою *y*. Рівняння другого закону Ньютона

$$m\frac{d^2y}{dt^2} = F$$

перепишемо у вигляді системи двох диференціальних рівнянь першого порядку:

$$\frac{dv}{dt} = a\big(t, y(t), v(t)\big), \frac{dy}{dt} = v(t), \tag{2.5}$$

де $a = F/m$. В початковий момент часу t_0 вважаються відомими значення координати точки та її швидкості

$$y(t_0) = y_0, v(t_0) = v_0. \tag{2.6}$$

Чисельно розв'язати задачу Коші (2.5), (2.6) на відрізку $[t_0, T]$ – означає знайти наближені значення функцій $y_k = y(t_k)$ та в деякі задані моменти часу $t_k \in [t_0, T]$, k = 1,2,3, …, n. В якості таких моментів звичайно обирають значення $t_0 + \Delta t, t_0 + 2\Delta t, \ldots., t_0 + n\Delta t = T$, тобто $t_k = t_0 + k\Delta t$, де Δt – крок інтегрування, який задається достатньо малим, щоб забезпечити стійкість проведення розрахунків. У випадку консервативних систем ще однією вимогою до величини Δt може бути збереження повної енергії системи.

Таким чином, основна задача чисельного інтегрування полягає у визначенні величин y_{k+1} та v_{k+1} за відомими значеннями y_k, v_k та Δt. Сутність багатьох обчислювальних алгоритмів може бути поясненням шляхом розкладання величин $v_{k+1} = v(t_k + \Delta t)$ та $y_{k+1} = y(t_k + \Delta t)$ у ряд Тейлора:

$$v_{k+1} = v_k + a_k\Delta t + O(\Delta t^2), y_{k+1} = y_k + v_k\Delta t + \frac{1}{2}a_k\Delta t^2 + O(\Delta t^3). \tag{2.7}$$

Метод Ейлера чисельного інтегрування полягає у відкиданні членів другого та більш високих порядків у співвідношеннях (2.7):

$$\begin{aligned} v_{k+1} &= v_k + a_k\Delta t, \\ y_{k+1} &= y_k + v_k\Delta t. \end{aligned} \tag{2.8}$$

Порядок відкинутих членів визначає локальну помилку урізання метода, тобто помилку на одному кроці інтегрування; таким чином, локальна помилка метода Ейлера має порядок Δt^2. Глобальна помилка цього метода, тобто помилка, що накопичується при інтегруванні по всьому часовому проміжку, буде пропорційна Δt. Грубо цю оцінку можна отримати, просто сумуючи всі локальні помилки на кожному кроці інтегрування, кількість яких пропорційна $(\Delta t)^{-1}$.

Порядок величини Δt в оцінці глобальної помилки визначає порядок метода: тому метод Ейлера вважається методом першого порядку. До його характеристик слід також віднести те, що він є *явним* та *самостартуючим*, тобто, знаючи $y(t_0)$ та $v(t_0)$ за формулами (2.8) без додаткових розрахунків можна послідовно обчислити $y(t_1),\ y(t_2)$, і т. д.

В чистому вигляді метод Ейлера на практиці використовують не дуже часто. Це пов'язано не тільки з невисоким порядком точності, але й з проблемами обчислювальної нестійкості, а також нефізичністю його результатів у ряді задач математичного моделювання, наприклад, з відсутністю збереження енергії коливальної системи.

Саме для розв'язання останньої згаданої проблеми використовується модифікація метода (2.8), яку будемо називати методом Ейлера – Кромера, за ім'ям автора роботи [2], де цю модифікацію достатньо докладно описано. Там для класичного варіанта метода Ейлера запропонована альтернативна назва FPA (first point approximation, апроксимація за першою точкою), оскільки для обчислення координати точки y_{k+1} використовується значення швидкості v_k у початковій точці інтервалу інтегрування. Як альтернативний варіант А. Кромер запропонував використовувати для обчислення y_{k+1} значення швидкості у кінцевій точці інтервалу:

$$\begin{aligned} v_{k+1} &= v_k + a_k \Delta t, \\ y_{k+1} &= y_k + v_{k+1} \Delta t. \end{aligned} \tag{2.9}$$

Як і метод Ейлера, даний метод має перший порядок точності, але він потрапляє в клас симплектичних інтеграторів або геометричних методів [22, розділ 4.6], які враховують властивості диференціальних рівнянь, точніше, їх розв'язків. Наприклад, для гамільтонових систем (а рівняння руху тіла, осцилюючого на пружинці, або системи тіл, що переміщуються під дією гравітаційних сил, відносяться до цього класу) чисельні рішення з відповідною порядку метода точністю зберігають енергію системи.

Ще однією модифікацією метода Ейлера є метод апроксимації за серединною точкою. Для обчислення координати тіла він використовує не початкове або кінцеве значення швидкості на відрізку, а їх середнє значення:

$$\begin{aligned} v_{k+1} &= v_k + a_k \Delta t, \\ y_{k+1} &= y_k + \frac{1}{2}(v_k + v_{k+1})\Delta t. \end{aligned} \tag{2.10}$$

Якщо підставити у другу рівність у (2.10) вираз для v_{k+1} з першої, то отримаємо відповідну схему:

$$\begin{aligned} v_{k+1} &= v_k + a_k \Delta t, \\ y_{k+1} &= y_k + v_k \Delta t + a_k \Delta t^2. \end{aligned} \tag{2.11}$$

З (2.11) видно, що метод апроксимації за середньою точкою забезпечує другий порядок точності для переміщення та перший порядок для швидкості. Але суттєвих практичних переваг у порівнянні з базовим методом Ейлера цей метод не має [2].

2.3. Методи другого порядку

Ідея обчислення швидкості в середині інтервалу використовується ще в одному достатньо розповсюдженому методі, який будемо називати методом Ейлера-Ричардсона, інша назва цього метода – метод середньої точки (Midpoint method). Метод Ейлера-Ричардсона зазвичай використовують, коли сили у рівнянні руху залежать від швидкості, в інших випадках його ефективність аналогічна попередньо описаним методам. Алгоритм метода складається з двох кроків. Спочатку методом Ейлера обчислюється положення y_{mid} та швидкість v_{mid} в середній точці часового проміжку. Потім обчислюється сила $F = (y_{mid}, v_{mid}, t_{mid})$, а за нею знаходяться прискорення a_{mid} у момент часу t_{mid}. Нове положення y_{k+1} та швидкість v_{k+1} у момент часу t_{k+1} знаходяться з використанням v_{mid} та a_{mid}. Таким чином, схему Ейлера-Ричардсона можна записати у вигляді наступних кроків:

$$
\begin{aligned}
a_k &= \frac{1}{m} F(y_k, v_k, t_k), \\
v_{mid} &= v_k + \frac{1}{2} a_k \Delta t, \\
y_{mid} &= y_k + \frac{1}{2} v_k \Delta t, \\
a_{mid} &= \frac{1}{m} F\left(y_{mid}, v_{mid}, t_k + \frac{1}{2}\Delta t\right), \\
v_{k+1} &= v_k + a_{mid} \Delta t, \\
y_{k+1} &= y_k + v_{mid} \Delta t.
\end{aligned}
\tag{2.12}
$$

На кожному кроці методу Ейлера-Ричардсона виконується у два рази більше обчислень, ніж на кожному кроці попередніх варіантів метода Ейлера. Тим не менш, багато дослідників відмічають його більш високу швидкодію, оскільки, будучи методом другого порядку, для великого кола завдань він дозволяє досягнути потрібної точності при великих значеннях кроку за часом.

Метод Ейлера-Ричардсона відноситься до одного з варіантів сімейства методів Рунге-Кутти другого порядку. Іншим варіантом з цього сімейства, що часто використовується, є метод Хойна (Heun's method), ідея якого у чомусь схожа: для підвищення точності апроксимації замість похідної на початку відрізку використовується деяке проміжне значення, у цей раз – середнє арифметичне двох величин: похідна на початку відрізка та її наближеного значення у кінці відрізка інтегрування, отриманого методом Ейлера. Алгоритмічна обчислювальна схема цього метода при застосуванні до системи (2.5) приймає вигляд:

$$
\begin{aligned}
a_k &= \frac{1}{m} F(y_k, v_k, t_k) \\
y^*_{k+1} &= y_k + v_k \Delta t, \\
v^*_{k+1} &= v_k + a_k \Delta t, \\
a^*_{k+1} &= \frac{1}{m} F(y^*_{k+1}, v^*_{k+1}, t_{k+1}) \\
y_{k+1} &= y_k + \frac{v_k + v^*_{k+1}}{2} \Delta t, \\
v_{k+1} &= v_k + \frac{a_k + a^*_{k+1}}{2} \Delta t.
\end{aligned}
\tag{2.13}
$$

Серед методів другого порядку дуже розповсюдженим, особливо при чисельному розв'язанні рівнянь молекулярної динаміки, є метод Верлє (інша назва – явний метод центральних різниць), що використовувався у роботі [19] для розрахунку руху системи з 864 частинок, взаємодія яких визначалося потенціалом

Леннарда – Джонса. Щоб визначити його розрахункові формули, подамо вираз y_{k-1} у вигляді, аналогічному другій формулі у (2.7):

$$y_{k-1} = y_k - v_k\Delta t + \frac{1}{2}a_k\Delta t^2 + O(\Delta t^3).$$

Додаючи дане подання для y_{k-1} з виразом для y_{k+1} з (2.7) та виражаючи з отриманої рівності y_{k+1}, знаходимо

$$y_{k+1} = 2y_k - y_{k-1} + a_k\Delta t^2 + O(\Delta t^4).$$

Якщо ж відняти ці вирази, то отримаємо співвідношення для швидкості:

$$v_k = \frac{y_{k+1} - y_{k-1}}{\Delta t} + O(\Delta t^3).$$

Таким чином, прийдемо до *оригінального* алгоритму Верле:

$$y_{k+1} = 2y_k - y_{k-1} + a_k\Delta t^2,$$
$$v_k = \frac{y_{k+1} - y_{k-1}}{\Delta t}.$$

Саме цей алгоритм був реалізований у движку *fysix*, який був використаний при розробці достатньо відомої у свій час комп’ютерної гри «Hitman: Codename».

Локальна помилка визначення координат методом Верле має четвертий порядок, а швидкостей – другий порядок. Але глобальна помилка у обох випадках однакова та має другий порядок. Зауважимо, що розрахункова схема цього методу взагалі не використовує значення швидкостей для обчислення координат.

Алгоритм Верле не є самостартуючим, тому для знаходження перших значень необхідно використовувати додаткові міркування або обчислювальні схеми. Найбільш розповсюдженим є обчислення першого наближення за формулою, похибка якої має третій порядок:

$$y_1 = y_0 + v_0\Delta t + \frac{1}{2}a_0\Delta t^2,$$

де $a_0 = \frac{F(y_0, v_0, t_0)}{m}$.

Ще одна проблема реалізації цього методу пов’язана з тим, що для обчислення нового значення швидкості потрібно віднімати величини, близькі за значеннями, а така операція, як відомо, може призводити до значних помилок округлення.

Якщо прискорення у рівняннях (2.5) руху частинки не залежить від швидкості, то звичайно використовують іншу форму алгоритму Верле, тобто алгоритм Верле в швидкостях (*velocity Verlet*). Його обчислювальна схема, математично еквівалентна основному варіанту, записується наступним чином:

$$\begin{aligned} y_{k+1} &= y_k + v_k \Delta t + \frac{1}{2} a_k \Delta t^2, \\ a_{k+1} &= a(t_k + \Delta t, y_{k+1}), \\ v_{k+1} &= v_k + \frac{1}{2}(a_k + a_{k+1})\Delta t. \end{aligned} \qquad (2.14)$$

Перевагою цього варіанту, який отримується з попереднього з використанням простих алгебраїчних перетворень, є те, що він самостартуючий, та формула для обчислення швидкостей не містить різниць близьких за значеннями величин.

Ще одним варіантом інтегрування рівнянь, права частина яких не залежить від швидкості, є **метод з перекрокуванням** (*leapfrog method*). Цей метод було застосовано у Фейнмановських лекціях з фізики для чисельного визначення положення вантажу, що підвішений на пружинці. Ідея методу пов’язана із заміною похідної на одному з кінців відрізка (як в методах Ейлера та Ейлера-Кромера) її значенням у середині відрізка. Перший крок інтегрування рівнянь (2.5) із заданими початковими умовами (2.6) має вигляд

$$y_1 = y_0 + v_{1/2}\Delta t,$$

де використано позначення

$$v_{1/2} = v\left(t_0 + \frac{\Delta t}{2}\right).$$

Вважатимемо далі, що величина $v_{1/2}$ відома, значення швидкості *y* момент часу $t_0 + 3\Delta t/2$ знайдемо, використовуючи значення похідної в середній точці відрізка інтегрування:

$$v_{3/2} = v_{1/2} + a(t + \Delta t, y_1)\Delta t.$$

Продовжуючи процес, приходимо до схеми з перекрокуванням, коли координата та швидкість обчислюються у різні моменти часу.

Ітераційна схема методу задається співвідношеннями:

$$y_{k+1} = y_k + v_{k+1/2}\Delta t,$$
$$v_{k+3/2} = v_{k+1/2} + a(t_k + \Delta t, y_{k+1})\Delta t. \quad (2.15)$$

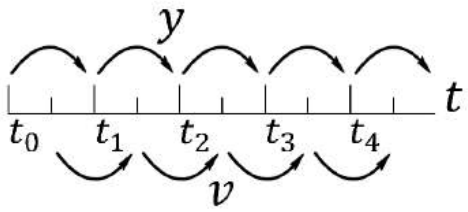

Рис.2.1 Схема інтегрування з перекрокуванням

Як і метод Верле, метод з перекрокуванням має другий порядок точності. Він не є самостартуючим, для запуску потрібно знання величини $v_{1/2}$. Найпростішим способом її обчислення є один крок метода Ейлера (точніше, половина кроку):

$$v_{1/2} = v_0 + a(t_0, a_0)\frac{\Delta t}{2}.$$

Похибка цієї формули має другий порядок, але оскільки вона застосовується лише один раз, то загальний другий порядок обчислювальної схеми не погіршується.

Строго кажучи, методи Верле та перекрокування можна застосовувати і в тих випадках, коли праві частини залежать від швидкості, але якщо ця залежність нелінійна, тоді на кожному кроці інтегрування кожне наступне наближення для швидкості доведеться знаходити, чисельно розв'язуючи нелінійне рівняння.

Потрібно зауважити, що обидва останніх методи є симплектичними. При застосуванні до фізичних задач це означає, що їм надається перевага, якщо при чисельному інтегруванні важливо зберегти деякі характеристики, наприклад, енергію.

Слід зауважити, що назви методів не є загальноприйнятими. Коректування переміщення за методом Ейлера у англомовній літературі часто позначається словом drift (дрейф, повільне переміщення), а коректування швидкостей – словом kick (удар, поштовх). Тому співвідношення (2.16) та (2.17) часто називають обчислювальними алгоритмами DKD (drift-kick-drift) та KDK (kick-drift-kick) відповідно.

2.4. Розв'язування задачі Коші з використанням бібліотеки scipy

Пакет `scipy` містить кілька функцій, призначених для розв'язування початкових задач. Найбільш вживаною з них є універсальна функція `solve_ivp` з підмодуля `integrate`. Ця функція призначена для розв'язування задачі Коші для системи рівнянь першого порядку наступного вигляду:

$$\frac{d\vec{y}(t)}{dt} = \vec{f}\left(t, \vec{f}(t)\right), \vec{y}(t_0) = \vec{y}_0, \quad (2.18)$$

де t – незалежна змінна («час»), $\vec{y}(t)$ – n-вимірна вектор-функція, що підлягає визначенню, а n-вимірна вектор-функція $\vec{f}(t, \vec{y})$ визначає диференціальні рівняння, точніше їх праві частини; $\overrightarrow{y_0}$ – заданий вектор початкових значень шуканої функції.

Для чисельного знаходження наближеного розв'язку сформульованої задачі на відрізку $[t_0, T]$ призначений виклик:

```
solution = solve_ivp(fun, [t0, T], y0, method='RK45',
                     t_eval = None, dense_output=False, **options)
```

Коротко перелічимо вхідні параметри функції:

- `fun` – функція з заголовком `fun(t, y)`, що повертає вектор правих частин системи. Параметр t є скалярним, а y, як результат функції fun, представляє собою масив (`ndarray`) розмірності n.
- `y0` – масив початкових значень розмірності n.
- `method` – рядок, що задає чисельний метод інтегрування. Окрім встановленого за замовчуванням явного метода Рунге – Кутти порядка точності 4/5, можливий вибір наступних значень: 'RK23', 'DOP853' (явні схеми Рунге-Кути другого та восьмого порядку, відповідно), 'Radau' (неявна схема Рунге-Кути 5-го порядка), 'BDF' (неявний багатокроковий метод змінного порядку), 'LSODA' (вибір метода інтегрування відбувається автоматично на основі аналізу жорсткості диференціального рівняння).
- `t_eval` – впорядкований список або масив значень параметра t, в яких потрібно визначити значення невідомих функцій. Всі числа з цього списку повинні знаходитися у межах відрізку `[t0, T]`. Якщо цей параметр

встановлений у None (значення за замовчуванням), то функції обчислюються в точках, що обираються розв’язувачем автоматично.

- `dense_output` – визначає, чи достатньо побудувати розв’язок у точках списку, заданого попереднім параметром (цей варіант встановлений за замовчуванням), або в подальшому може знадобитися знаходження розв’язку у всіх точках відрізка `[t0, T]`. У другому випадку результат роботи функції - об’єкт `solution` – буде містити додатковий метод, що дозволяє здійснити такі обчислення на основі сплайн-інтерполяції, порядок якої узгоджується з порядком метода інтегрування.
- Набір додаткових ключових параметрів `options` пов'язаний з передаванням налаштувань тому чи іншому методу. Як приклад можна навести параметри `rtol` та `atol`, які є дійсними числами або масивами розмірності n та задають, відповідно, відносну та абсолютну похибку обчислювальних схем. Розв’язувач намагається гарантувати оцінку локальної похибки величиною `atol + rtol + abs(y)`. Наближено можна вважати, що `rtol` керує кількістю вірних значущих цифр, якщо тільки саме значення у не стає менше величини `atol`. Якщо компоненти вектора `y` мають різний масштаб, то може бути корисно встановлювати значення `atol` окремо для кожної компоненти, використовуючи масив розмірності n. За замовчуванням `rtol=1e-3, atoll=1e-6`.

Значення, яке повертає функція `solve_ivp(solution)` являє собою об’єкт достатньо складної структури. Перелічимо кілька полів цього об’єкта:

- `solution.t` – одновимірний масив типу ndarray довжини n_points, що містить список точок відрізка `[t0, T]`, в яких було обчислено розв’язок. Співпадає з `t_eval`, якщо останній заданий, в протилежному випадку кількість точок та крок (відстань між точками) визначаються автоматично обраним методом інтегрування;
- `solution.y` – одновимірний масив типу `ndarray` довжини `n_points`, що містить значення шуканої функції;

- `solution.sol` – об'єкт типу `OdeSolution` (метод, що дозволяє обчислити розв'язок в довільній точці відрізку `[t0, T]` на основі інформації про розв'язок в точках масиву `solution.t` шляхом сплайн-інтерполяції) або **`None`** (якщо при виклику функції параметр dense_output дорівнює **`False`**);
- `solution.message` – рядок, опис причини завершення (успішного або ні) роботи, наприклад, '`The solver successfully reached the end of the integration interval.`';
- `solution.success` – логічна змінна, що має значення **`True`**, якщо досягнутий кінець інтервалу інтегрування.

Дуже важливою є тема жорсткості систем диференціальних рівнянь. Як наведено у [11]: жорсткими прийнято називати системи звичайних диференційних рівнянь чисельне дослідження яких призводить до таких неприємних результатів як обчислювальні нестійкості, надлишкова чуттєвість до початкових даних, необхідність практично недосяжного зменшення кроку інтегрування для досягнення потрібної точності і т. п., навіть для тих випадків, коли самий розв'язок є гладкою та функцією, що слабко змінюється.

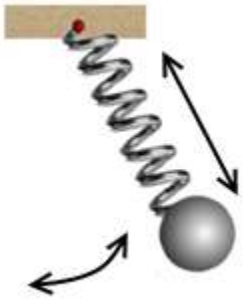

Рис. 2.2 Система з двома типами коливань

При розв'язуванні прикладних задач часто потрібно моделювати фізичні явища з дуже різними часовими або просторовими масштабами. Такі задачі звичайно призводять до систем звичайних диференціальних рівнянь, розв'язок яких включає кілька членів, величина яких змінюється з суттєво різною швидкістю. Наприклад, як показано на малюнку, кулька на пружинці може коливатися зліва

направо, а також коливатися вгору та донизу вздовж напрямної пружини. Таким чином, в задачі є дві різні шкали часу: період маятникових рухів системи та період коливального руху (стиснення-розтягування) пружини. Якщо пружина дуже жорстка, то частота коливань пружини буде значно вище частоти коливань маятника. Для чисельного моделювання системи доведеться обирати дуже маленький крок за часом, щоб отримати задовільний опис цих коливань.

Залежно від властивостей системи, що вивчається, доводиться обирати метод її розв’язання. Прямі методи, такі як `'RK45'` або `'RK23'` підходять для нежорстких систем, а методи `'Radau'` або `'BDF'` – для жорстких. Якщо властивості системи заздалегідь невідомі, то документація з пакету `scipy` пропонує розпочати з використання `'RK45'`. Якщо ж розв’язок потребує надлишково велику кількість ітерацій, або чисельна схема не сходиться або завершується аварійно, то рекомендується (пропонується) спробувати `'Radau'` або `'BDF'`. Метод `'LSODA'` також може бути вдалим універсальним розв’язкам, оскільки в ньому реалізовано автоматичне переключення з явного метода до неявного при визначенні проблем з інтегруванням.

РОЗДІЛ 3

ВІЗУАЛІЗАЦІЯ ПРУЖИННОЇ МОДЕЛІ ТВЕРДОГО ТІЛА

Завдання дипломної роботи полягає у розробці комп'ютерної візуалізації механічної системи. Як механічну систему розглянемо пружинну модель твердого тіла, її математичну та алгоритмічну реалізації та візуалізацію засобами бібліотеки `vpython`.

3.1. Формулювання завдання

Розробимо програму, що візуалізує падіння тіла у формі простого геометричного тіла, всі кульки якого мають однакову масу, а пружинки – однакову жорсткість. Інтегрування виконаємо кількома методами (метод можна буде вибрати під час запуску програми), інтервал інтегрування, крок інтегрування можна буде задати власноруч.

Як модель використаємо прості геометричні тіла – тетраедр, куб, призма, а також обєкти, які можна з них отримати шляхом видалення окремих ребер та граней. Для визначення координат вершин використаємо візуальний редактор, що дозволяє моделювати вказані прості тіла з кульок та пружин для зв'язку. Координати вершин та пружин для зв'язку потрібно завантажити у програму для візуалізації.

Зробимо кілька зауважень відносно параметрів моделювання. Розглянемо величину жорсткості пружини. Спробуємо оцінити, як можуть бути пов'язані між собою маси кульок, відстань між ними та жорсткість пружини. Формула для коефіцієнта жорсткості крученої циліндричної пружини, яка намотана з дроту круглого поперечного перерізу, має вигляд

$$k = \frac{\mu r^4}{4nR^3}, \tag{3.1}$$

де μ - модуль зсуву матеріалу пружини, r – радіус дроту, R – радіус намотки, n – кількість витків пружини.

Спираючись на формулу (3.1), визначимо жорсткість пружини, що з'єднує дві сталевих кульки, маса кожної по 100 г. Із врахуванням густини сталі 7.7 – 7.9

г/см3, радіус кульки повинен бути в діапазоні 1.45 – 1.46 см.(1.5 см). Саме це значення було використано для побудови наведеного зображення. Для значення радіуса дроту при цьому була обрана величина 0.2 см, а для радіуса намотки – 0.5 см. Довжина пружини – 10 см, а загальна кількість витків – 25. Після переведення перелічених одиниць довжини в метри, підставимо їх та величину модуля зсуву для сталі, 80 ГПа, у формулу, отримаємо, що жорсткість намальованої пружини приблизно дорівнює 100 Н/м.

Таким чином, можна вважати, що всі розмірні вхідні параметри задачі задані в системі СІ, тобто час t вимірюється у секундах.

3.2. Бібліотека VPython 7

Для моделювання використаємо бібліотеку VPython. Бібліотека VPython призначена для суттєвого полегшення розробки нескладних 3D-моделей та анімаційних роликів. Бібліотека не демонструє надвисоку ефективність та потужність графічного рушія, але дозволяє швидко та порівняно просто візуалізувати механічні, фізичні та хімічні явища та процеси, забезпечуючи при цьому певний рівень інтерактивності. Використання мови програмування Python як основи дозволяє достатньо просто засвоїти та використовувати основні функції навіть не професійним програмістам.

Проект VPython було започатковано Давидом Шерером (David Scherer) у 2000 році. У 2011 році разом з Брюсом Шервудом (Bruce Sherwood) почалася розробка середовища GlowScript – програмної оболонки для VPython, що працювала у вікні браузера. З 2014 року у середовищі GlowScript з'явилася можливість використання мови програмування RapidScript, дуже наближеної до стандартного Python. В той самий час, групою програмістів під керівництвом Джона Коуди (John Coady) розпочався та продовжується розвиток пакету VPython 7, що є класичною бібліотекою для стандартного Python, що дає можливість використовувати будь які інші бібліотеки Python.

VPython 7 спочатку був орієнтований на застосування у відомому середовищі Jupyter notebook. З 2017 року програми, що використовують цей пакет можуть

запускатися зі стандартного IDLE або з середовища Spyder, при цьому вікно з 3D-сценою відображається у вкладці системного браузера. У даний час розробники підтримують обидві версії продукту: як GlowScript VPython, так і VPython 7. Обидві реалізації використовують бібліотеку трьохвимірної графіки WebGL, яка дозволяє розв'язувати досить складні задачі побудови зображень з використанням сучасного графічного обладнання у середовищі браузера. До основних переваг браузерної графіки відносять відсутність коду, специфічного для операційної системи, та відсутність необхідності у інсталяторах. Крім цього, бібліотеки орієнтовані на браузери, краще підтримуються розробниками. Для роботи була обрана бібліотека VPython 7.

У сучасній літературі з механіки та фізики VPython використовується досить широко; наприклад, університетські курси [14], лабораторні практикуми [18], література для професійних науковців [6].

Бібліотека досить об'ємна, тому обсяг роботи не дозволяє включити її детальний опис.

Пропонуємо звернутися до офіційної документації з пакету [7]. Розробники пакету створили та ведуть YouTube-канал «Let's code|Physics» [30].

Зауважимо, що VPython стандартно не встановлюється при інсталяції Python. Встановити VPython можна з використанням термінала таким чином:

```
запрошення_системи> pip3 install vpython
```

3.3. Опис класів

В даній програмі використані такі класи:

- `class Mass` – клас для роботи з кульками/масами;
- `class Spring` – клас для роботи з пружинками;
- `class Solid` – клас для роботи з тілом в цілому

Зауважимо, що нумерація рядків у наведених фрагментах коду наведена для зручності опису, у реальних програмах нумерація відрізняється.

3.3.1. Клас Mass

```
class Mass:
    """ Клас, що описує кульку """
    def __init__(self, pos, radius = BALL_R,
                 color = BALL_COLOR,
                 mass = BALL_MASS):
        self.pos = pos
        self.mass = mass
        self.v = vec(0,0,0)
        self.springs = []
        self.ball = sphere(pos = pos,
                           radius = radius,
                           color = color)

    def add_spring(self, new_s):
        self.springs.append(new_s)

    def redraw(self):
        self.ball.pos = self.pos

    def a(self):
        f = vector(0,0,0)
        for spr in self.springs:
            if self == spr.m1:
                another = spr.m2
            else:
                another = spr.m1
            r = another.pos - self.pos
            delta_1 = mag(r) - spr.length
            f += spr.k * delta_1 * hat(r)
            return f/self.mass + g * vector(0,-1,0)
```

Рис. 3.1 Програмний код класу Mass

Атрибути класу Mass:

`pos` – координати поточного положення кульки;

`mass` – маса кульки;

`v` – поле швидкості;

`springs` – список пружин, якими ця кулька з'єднана з іншими;

`ball` – сфера у положенні `pos`, відповідного кольору та радіусу.

Методи класу Mass:

`__init__()` – екземпляр класу `Mass` ініціюється власним унікальним положенням, інші параметри (маса, колір та радіус кульки, що використовується при візуалізації) можуть мати значення за замовчуванням. При ініціалізації об’єкта створюється нульове поле швидкості (рис. 3.1, рядок 8), а також порожній в момент ініціалізації список пружин, якими це тіло сполучене з іншими (рис. 3.1, рядок 9). У момент створення маса відразу відображається у вигляді кульки, розташованої у потрібному місці (рис. 3.1, рядки 10-12).

`add_spring()` – метод для поповнення списку пружин новою пружиною (рис. 3.1, рядки 14-15);

`redraw()` – метод перемальовування кульки (рис. 3.1, рядки 17 – 18); цей метод дозволяє проводити обчислення та коректувати реальні положення об’єктів, що визначається параметром `pos`, частіше, ніж здійснювати достатньо повільний процес перемальовування, пов'язаний з параметром `ball.pos`. Цей підхід використаний у класі `Spring`, де він ще важливіший, оскільки відображення пружини ще більш повільний процес.

Прискорення кульки визначимо не як властивість, а як функцію `a()` – метод класу `Mass` (рис. 3.1, рядки 20 – 30). При його описі врахуємо, що на кожну масу діють сили пружності приєднаних до неї пружин та сила тяжіння (рис.3.2).

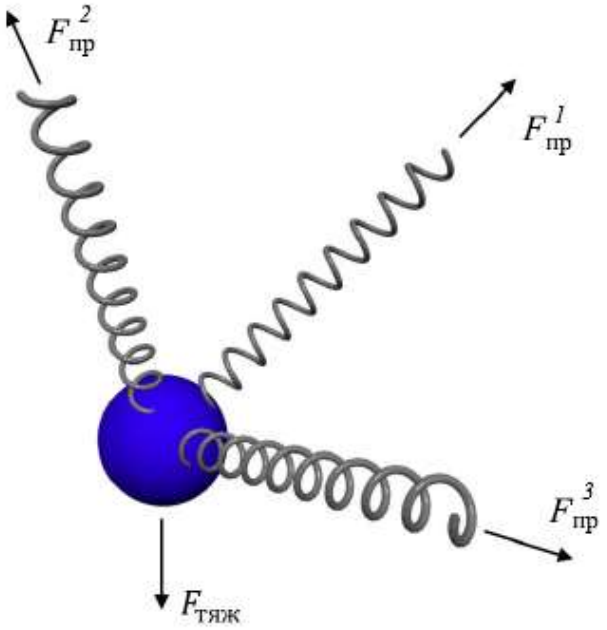

Рис.3.2 Сили, що діють на кульки

Сила пружності обчислюється в циклі по списку пружин. Щоб знайти довжину пружини, спочатку визначається друге тіло (змінна `another`), як тіло, що пов’язане з пружиною, але таке, що не співпадає з даним. Підсумкова сила пружності обчислюється як добуток коефіцієнта жорсткості пружини на її подовження та на одиничний вектор, що йде від заданого тіла до знайденого другого (рис. 3.1, рядок 29). Підсумкове прискорення обчислюється як сума сил пружності, поділених на масу тіла, та прискорення вільного падіння.

3.3.2. Клас Spring

```
1  class Spring:
2      """ Клас, що описує пружину """
3      def __init__(self, m1, m2, k = HELIX_STIFFNESS,
4                   radius = HELIX_R, coils = 25,
5                   thickness = HELIX_R/2.5):
6          self.m1 = m1
7          self.m2 = m2
8          self.length = mag(m1.pos - m2.pos)
9          self.k = k
10         self.spr = helix(pos = m1.pos,
11                          axis = m2.pos - m1.pos,
12                          radius = radius,
13                          thickness = thickness,
14                          coils = coils)
15
16     def cur_length(self):
17         return(mag(self.m2.pos - self.m1.pos))
18
19     def redraw(self):
20         self.spr.pos = self.m1.pos
21         self.spr.axis = self.m2.pos - self.m1.pos
```

Рис.3.3 Програмний код класу Spring

Атрибути класу Spring:

`m1, m2` – кульки, що з'єднує пружина;

`length` – початкова довжина пружини, тобто довжина пружини у недеформованому стані;

`k` – жорсткість пружини;

`spr` – візуальний елемент – об'єкт `helix` бібліотеки `vpython` – створюється у рядках (рис.3.3, рядки 10 – 14). Початок цього об'єкта розташовано у центрі однієї з кульок, а вісь – це вектор, що з'єднує центри кульок.

Методи класу Spring:

`__init__()` – екземпляр класу Spring ініціалізується двома масами, які ця пружина з'єднує (рис.3.3, рядки 6-7). Що стосується жорсткості, та параметрів, що візуалізують пружину, то вони можуть бути задані за замовчуванням. При

створенні пружини обчислюється та зберігається у полі `length` її початкова довжина, тобто довжина у недеформованому стані (рис.3.3, рядок 8). `cur_length()` – призначений для обчислення довжини деформованої пружини, що знадобиться для обчислення сили, з якою пружина діє на пов'язані з нею тіла.

`redraw()` – дозволяє відокремити обчислювальні процеси від візуалізації та виконувати візуалізацію за необхідністю.

3.3.3. Клас Solid. Створення тіла для моделювання

```
1  class Solid:
2      """ Клас, що описує пружинну модель цілком """
3      def __init__(self,coords,links=None):
4          self.body = []
5          self.coords = [tmp for tmp in coords]
6          tmp = 0
7          for v in coords:
8              self.body.append(Mass(pos=vector(v[0],v[1]+BALL_R,v[2])))
9              tmp += 1
10         self.n_mass = len(self.body)
11         if links==None:
12             self.links = [[0 for i in range(self.n_mass)] for j in range(self.n_mass)]
13         else:
14             self.links = links
15         self.springs = []
16         for i in range(self.n_mass):
17             for j in range(i+1, self.n_mass):
18                 if self.links[i][j]:
19                     new_spring = Spring(self.body[i],self.body[j])
20                     self.springs.append(new_spring)
21                     self.body[i].add_spring(new_spring)
22                     self.body[j].add_spring(new_spring)
```

Рис. 3.4 Фрагмент коду класу Solid. Конструктор класу

Оскільки клас `Solid` має значний об'єм коду та використовується не тільки для моделювання руху механічної системи, а також для редагування тіл, то опишемо лише атрибути та методи необхідні для моделювання руху.

Атрибути класу `Solid`:

`body` – список об'єктів класу Mass, що складають пружинну модель;

`coords` – за координатами з цього списку створюються об'єкти класу Mass;

`n_mass` – кількість кульок, з яких складається модель;

`springs` – список пружин моделі;

`links` – матриця зав’язків між кульками.

Деяки методи класу `Solid`:

`__init__()` – конструктор класу, що безпосередньо створює пружинну модель;

Група методів `getIsoList()`, `isIsolated()`, `removeIsolated()`, `removeAllIsolated()`, `rePaintIsolated()` пов’язана з пошуком та видаленням ізольованих кульок, тобто кульок, що не з’єднані з іншими жодною пружиною;

Методи `addSpring()`, `delSpring()` – призначені для додавання та видалення пружини між двома обраними кульками;

`rise()` – метод для зміни положення моделі за координатою y;

`energy()` – метод обчислення повної енергії системи кульок (детальніше див. п.3.4);

`com_y()` – метод обчислення координати у центру мас (детальніше див. п.3.4);

`animate()` – призначений для візуалізації руху та побудови графіків (детальніше див. п.3.4).

Для візуалізації тіла, що моделюється, необхідно створити потрібну кількість об’єктів у потрібних положеннях (рис3.4, рядки 16-22). Набір об’єктів та їх координати формуються з використанням редактора стандартних форм. Було обрано два шаблони з яких можна побудувати кілька розповсюджених геометричних тіл: тетраедр, куб, призму і т. ін.

```
#
# Координати вершин шаблонів
#
prizm3 = [[0,0,0],[d,0,0],[d/2,0,sqrt(3)/2*d],[d/2,0,sqrt(3)/6*d],
          [0,sqrt(6)/3*d,0],[d,sqrt(6)/3*d,0],[d/2,sqrt(6)/3*d,sqrt(3)/2*d],
          [d/2,sqrt(6)/3*d,sqrt(3)/6*d]]
cube = [[0,0,0],[d,0,0],[d,0,d],[0,0,d],[d/2,0,d/2],
        [0,d,0],[d,d,0],[d,d,d],[0,d,d],[d/2,d,d/2]]
```

Рис.3.5 Фрагмент коду програми ShapeEditor.

Для зв’язку між масами використовується матриця `links`, елемент [i][j] якої дорівнює 1, якщо між і-ю та j-ю кульками є пружинка, та 0 – якщо пружинка відсутня. Матриця `links` формується за допомогою редактора стандартних форм, але для більш складних форм зв’язки доведеться формувати вручну. Створені форми завантажуються у програму візуалізації.

Після створення матриці зв’язків у доповнення до списку кульок створюється список `springs` – пружин, що з’єднують кульки (рис.3.4, рядки16-22). Пружина, створена у випадку ненульового значення у матриці звя'зків, додається і в загальний список пружин (рис.3.4, рядок 19), і в список пружин кожної з двох кульок, які вона з’єднує (рис.3.4, рядки 21-22).

3.3.4. Клас Solid. Інтегрування та створення анімації

Розглянемо основний метод класу `Solid – animate()`. Повний код можна знайти у додатках, а тут розглянемо лише окремі фрагменти.

Змінна `display` використовується для того, щоб достатньо повільні операції – малювання трьохвимірних об’єктів та відображення точок на графіку – виконувалися не на кожному кроці інтегрування за часом, а, наприклад, на кожному двадцятому кроці.

```
9           display = 0
10          while t < T and repeat:
11              t += DELTA_T
12              if param.INTMETHOD == 1: # Метод Ейлера - Кромера
13                  for m in self.body:
14                      m.v += m.a() * DELTA_T
15                  for m in self.body:
16                      m.pos += m.v * DELTA_T
17                      if m.pos.y < ground.pos.y + ground.size.y:
18                          m.v.y = abs(m.v.y)
19              elif param.INTMETHOD == 2: # Метод Ейлера
20                  for m in self.body:
21                      m.acc=m.a()
22                  for m in self.body:
23                      m.pos += m.v*DELTA_T
24                      if m.pos.y < ground.pos.y + ground.size.y:
25                          m.v.y = abs(m.v.y)
26                  for m in self.body:
27                      m.v += m.acc*DELTA_T
28              elif param.INTMETHOD == 3: # Метод середньої точки (Middle point)
29                  for m in self.body:
30                      m.acc = m.a()
31                      m.v += m.acc*DELTA_T
32                  for m in self.body:
33                      m.pos += m.v*DELTA_T + 0.5*m.acc*DELTA_T**2
34                      if m.pos.y < ground.pos.y + ground.size.y:
35                          m.v.y = abs(m.v.y)
```

Рис. 3.6 Методи інтегрування

Стосовно методу інтегрування, то у фрагменті на рис.3.6 наведено кілька методів інтегрування. У рядках 13-18 рис. 3.6 наведено інтегрування методом Ейлера – Кромера, коли спочатку обчислюються прискорення всіх кульок, потім визначаються їх швидкості, а потім уточнюються положення. В рядках 17-18 рис.3.6 коректуються швидкості кульок, які досягли рівня землі: вертикальна компонента швидкості змінює знак.

Для даної реалізації класу `Mass` метод Ейлера-Кромера реалізується найбільш компактно. Для того, щоб отримати схему інтегрування класичним методом Ейлера, недостатньо поміняти місцями набір рядків 15-18 рис. 3.6 з рядками 13-14. Справа в тому, що за такою заміною швидкості будуть знаходитися за прискореннями, обчисленими у момент часу t+delta t, а не у момент часу t, як вимагає класичний варіант. Для його реалізації у клас `Mass` потрібно додати додаткове поле, наприклад, `acc`, для збереження значень прискорення даної кульки, ініціювавши його, як і швидкість `v`, нульовим вектором.

Рядки 13-18 рис.3.6 потрібно переписати так, як наведено на малюнку 3.6. рядки 20-27.

Збереження прискорень знадобиться також при реалізації метода апроксимації за серединною точкою. Реалізація метода апроксимації за серединною точкою наведена у рядках 29-35 рис. 3.6.

```
1  def energy(self):
2          """ Обчислення енергії системи """
3          e1 = sum(m.mass * (m.pos.y * g + mag2(m.v)/2) for m in self.body)
4          e2 = sum(spr.k * (spr.cur_length()-spr.length)**2/2 for spr in self.springs)
5          return e1 + e2
6
7      def com_y(self):
8          """ Обчислення координати у центу мас """
9          total_mass = sum(m.mass for m in self.body)
10         y = sum(m.mass * m.pos.y for m in self.body)
11         return y/total_mass
```

Рис. 3.7 Обчислення енергії системи та координати у центу мас

Повна енергія системи представляє собою суму кінетичної енергії кульок

$$E_k = \sum \frac{m_i v_i^2}{2},$$

їх потенційної енергії у полі сили тяжіння

$$E_p = \sum m_i \, g y_i$$

та енергії пружної деформації пружин

$$E_{\text{пр.}} = \sum \frac{k_i \Delta l_i^2}{2}$$

де Δl - зміна довжини пружини. Функція обчислення повної енергії наведена у фрагменті коду на рис., рядки 1-5. Оскільки `energy()` є методом класу, то розрахунки виконуються над членами класу: списками кульок, тобто об'єктів класу `Mass`, та списком пружинок, тобто об'єктів класу `Spring`.

У фрагменті коду рис. 3.7 (рядки 7-11) наведена також функція обчислення центру мас (Center of Mass) системи кульок, точніше координати `y` цього центра за формулою:

$$y_{\text{ц.м.}} = \frac{\sum m_i y_i}{\sum m_i}.$$

Ця величина, так само як і енергія, може використовуватися для порівняння результатів чисельних методів.

Обчислені величини енергії та координати у центу мас відображаються у вигляді графіків.

РОЗДІЛ 4

ОПИС СТВОРЕНИХ ПРОГРАМ

Для запуску створених програм необхідно, щоб на комп'ютері був встановлений інтерпретатор мови Python, а також кілька необхідних для роботи бібліотек: `vpython, numpy, scipy, tkinter` (стандартно встановлюється разом з Python, в окремих випадках потребує встановлення додатково).

До складу проекту входять такі файли:

- ShapeEditor_1.3.py – редактор форм;
- SpringModel_1.0.py – програма для візуалізації руху механічної системи;
- Mass.py – код класу Mass;
- Spring.py – код класу Spring;
- Solid.py – код класу Solid;
- settings.py - файл з початковими налаштуваннями;
- набір .mod – файлів – файли з готовими моделями.

4.1. Програма ShapeEditor

Програма призначена для полегшення визначення координат кульок моделі та створення між кульками зав'язків з пружин. Програма дозволяє обрати два шаблони – правильна трикутна призма та куб з центрами.

Можливості програми:

- Візуальне моделювання форм на основі двох стандартних шаблонів.
- Можливість створення та видалення зв'язків між кульками.
- Пошук та видалення ізольованих кульок.
- Запис інформації про модель у файл (зберігаються координати кульок та матриця зв'язків).

Інтерфейс програми поданий на малюнку :

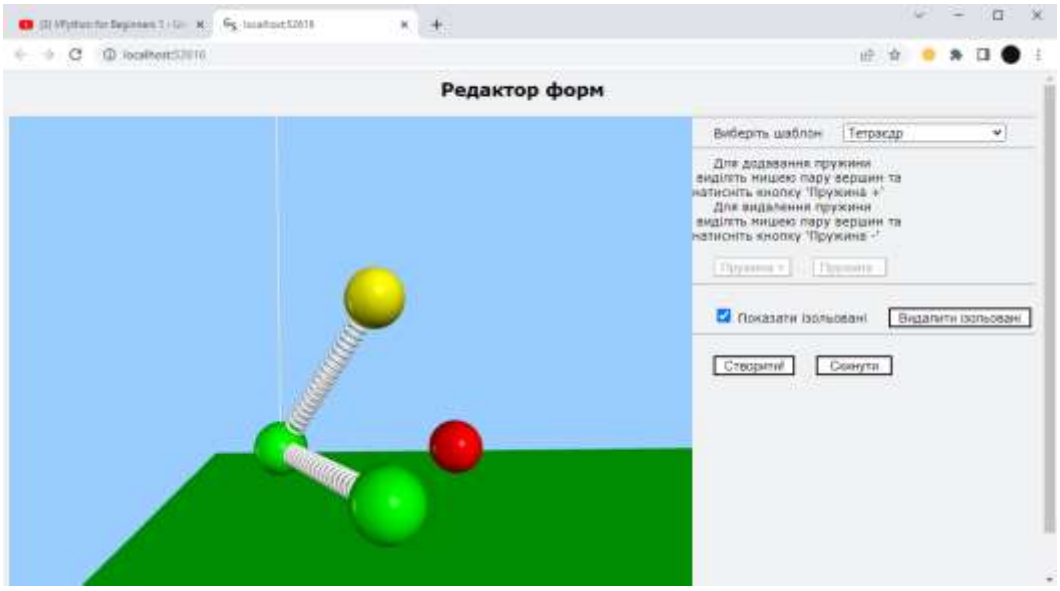

Рис. 4.1. Інтерфейс програми ShapeEditor

Інтерфейс програми складається з таких елементів:

- Робоча область, 3D-сцена – область, де безпосередньо виконується створення та зміна форми. Кульки у сцені можуть бути трьох кольорів:
  - Зелена – звичайна, не виділена кулька.
  - Жовта – виділена. Між парою виділених кульок можна створити зв'язок-пружину за допомогою кнопки «Пружина+», якщо зв'язок вже існує – його можна видалити кнопкою «Пружина–». Виділити кульку можна за допомогою клацання по кульці лівою кнопкою миші. Щоб зняти виділення достатньо клацнути виділену кульку лівою кнопкою миші.
  - Червона – ізольована кулька, тобто така, що не має жодного зв’язку. Ізольовані кульки можна побачити з допомогою прапорця «». Видалити ізольовані кульки можна за допомогою за допомогою кнопки «Видалити ізольовані»

Навігація 3D-сценою:

- Для переміщення «камери», тобто позиції спостерігача, потрібно натиснути ПРАВУ КНОПКУ МИШІ у межах сцени та, утримуючи її, переміщувати вказівник. Аналогічний ефект можна отримати, утримуючи

клавішу Ctrl та переміщуючи вказівник при натисненій ЛІВІЙ КНОПЦІ МИШІ.

- Перетаскування при натисненій ЛІВІЙ КНОПЦІ МИШІ і при затиснутій клавіші Shift призводить до переміщення камери в одній площині (зображення при цьому рухається «паралельно» екрану).
- Віддаляти та наближати об’єкт можна обертанням колеса миші або переміщенням миші при одночасно натиснутих лівій та правій кнопках миші.

- Список «Виберіть шаблон» – дозволяє вибрати один з двох наданих шаблонів. Відкритий код програми дозволяє легко додати нові форми.
- Кнопки «Пружина+», «Пружина –» - призначені для створення та видалення пружин-зав’язків між кульками. Зараз, на малюнку, ці кнопки неактивні, оскільки відсутня пара виділених вершин.
- Прапорець «Показати ізольовані» - дозволяє виділити червоним кольором ізольовані кульки.
- Кнопка «Видалити ізольовані» - призначена для видалення ізольованих вершин. Будьте обережні! Ізольовані вершини видаляються без підсвічення та підтвердження.
- Кнопка «Створити» - дозволяє зберегти створену форму у файл для використання у програмі SpringModel.

Кнопка «Скинути» - повертає інтерфейс у початковий стан.

Повний код програми наведено у Додатку В.

4.2. Програма SpringModel

Програма призначена для моделювання падіння тіла, що має нульову початкову швидкість, на горизонтальну тверду поверхню, вважаючи удар абсолютно пружним.

Можливості програми:

- Завантаження попередньо створеної моделі тіла у вигляді простого геометричного тіла (куб, призма, піраміда тощо);

- Візуалізація поведінки моделі у 3D-сцені;
- Візуалізація супроводжується побудовою графіка залежності повної енергії системи в залежності від часу;
- Можливість вибору способу чисельного розв’язування диференціальних рівнянь;
- Можливість задати такі початкові параметри: висота тіла над поверхньою, маса кульок моделі, жорсткість пружин, час та крок інтегрування.

Інтерфейс програми поданий на малюнку :

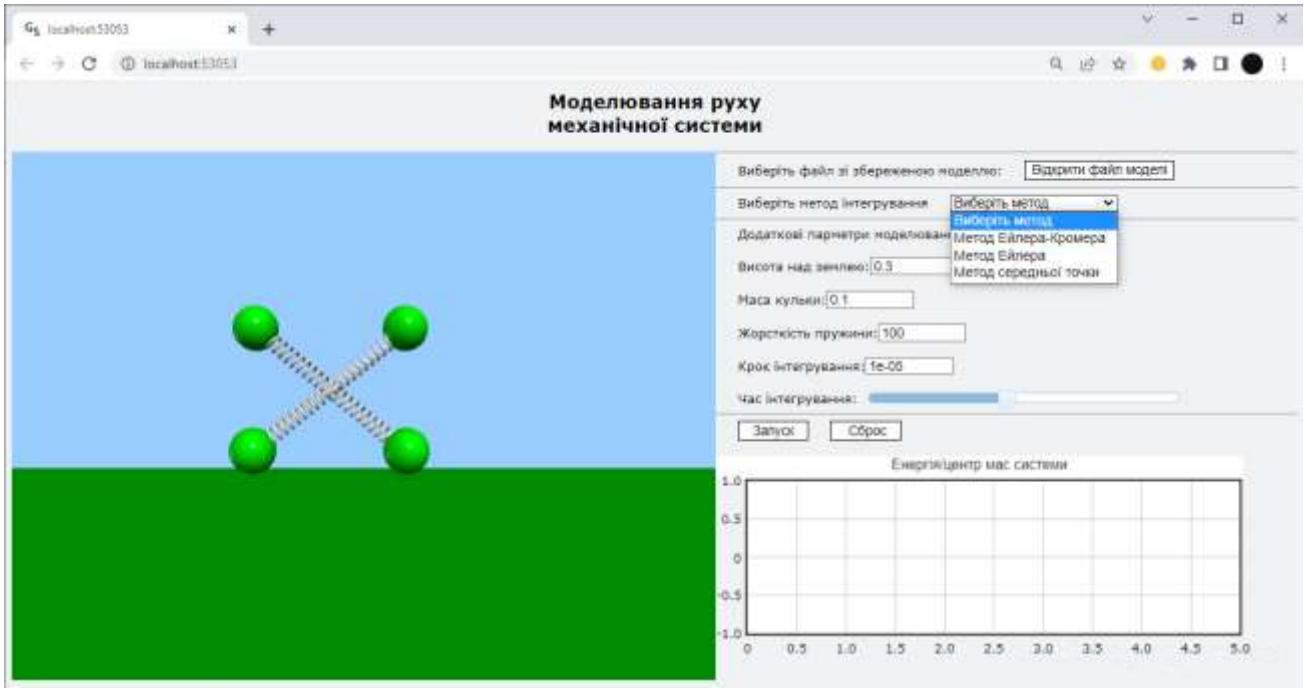

Рис. 4.2. Інтерфейс програми SpringModel

Інтерфейс програми складається з таких елементів:

- Робоча область, 3D-сцена – область, де безпосередньо виконується моделювання руху механічної системи.
- Кнопка вибору моделі. Ця кнопка викликає додаткове діалогове вікно для завантаження моделі, що підготована у програмі ShapeEditor. Якщо модель не завантажити, то процес моделювання запустити не вдасться – кнопки «Запуск» та «Сброс» будуть неактивними.
- За допомогою списку «Виберіть метод інтегрування» можна вибрати метод інтегрування: метод Ейлера-Крмера, метод Ейлера, метод середньої точки.

- Поля введення дозволяють задати параметри моделі у системі СІ, тобто, Всі поля введення заповнені значеннями за замовчуванням, але допускають редагування.
- Час інтегрування задасться за допомогою слайдера в межах 1-10 с.
- Процес анімації супроводжується побудовою графіків для повної
- Кнопки «Запуск» та «Сброс» дозволяють відповідно запустити процес анімації та повернути програму до початкового стану.

Повний код програми наведено у Додатку Д.

ВИСНОВКИ

При виконанні дипломної роботи виконані наступні завдання:

1. Проаналізовані різні чисельні методи розв’язування задачі Коши;
2. Розглянута бібліотека 3D моделювання VPython 7;
3. Розроблена програма моделювання пружинної моделі твердого тіла SpringModel та простий редактор ShapeEditor з використанням мови програмування Python;

За результатами роботи можна зробити наступні висновки:

1. Мова програмування Python та бібліотека VPython 7 придатні для створення простих інтерактивних 3D-моделей, для використання у навчальному процесі та у науковій діяльності.
2. Набір віджетів бібліотеки VPython має всі необхідні елементи, але не дуже зручний для побудови складних розвинутих інтерфейсів. Нажаль, спроби інтегрувати 3D-сцену у вікно tkinter виявилися невдалими.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Blickhan R. The spring-mass model for running and hopping // J. Biomechanics. – 1989. – Vol. 22, № 11/12. – Pp. 1217–1227. doi: 10.1016/0021-9290(89)90224-8
2. Cromer A. Stable solutions using the Euler approximations // American Journal of Physics. – 1981. – Vol. 49. – Pp. 455–459. doi: 10.1119/1.12478
3. De la Cruz S. T., Rodrнguez M. A., Hernбndez V. Using Spring-Mass Models to Determine the Dynamic Response of Two-Story Buildings Subjected to Lateral Loads [Електронний ресурс] // Proceedings of the 15th World Conference on Earthquake Engineering, Lisbon, 2012. – Vol. 31. – Pp. 24719–24726. URL: http://toc.proceedings.com/24574webtoc.pdf (дата звернення 03.06.2022)
4. Differential Equation: Modeling: Example: Spring Mass [Електронний ресурс]. – URL:http://www.sharetechnote.com/html/DE_Modeling_Example_SpringMass.html (дата звернення 03.06.2022)
5. Gezerlis A. Numerical Methods in Physics with Python. – Cambridge University Press, 2020. – 586 p.
6. Giovanni Moruzzi Essential Python for the Physicist. – Springer, 2020. – 304 p. https://doi.org/10.1007/978-3-030-45027-4
7. GlowScript VPython and VPython 7. Documentation [Електронний ресурс] – URL:https://www.glowscript.org/docs/VPythonDocs/index.html (дата звернення 03.06.2022)
8. Jakobsen T. Advanced Character Physics. [Електронний ресурс] – URL: https://www.researchgate.net/publication/228599597_Advanced_character_physics (дата звернення 03.06.2022)
9. Jarrousse O. Modified Mass-Spring System for Physically Based Deformation Modeling. KIT Scientific Publishing, 2012. 222 p. [Електронний ресурс]. – URL: https://www.ksp.kit.edu/9783866447424 (дата звернення 03.06.2022)
10. Kiusalaas J. Numerical Methods in Engineering with Python 3. – Cambridge University Press, 2013. – 422 p.
11. Kong Q., Siauw T., Bayen A. Python Programming and Numerical Methods: A Guide for Engineers and Scientists. – Academic Press, 2020. – 480 p.

12. Landau R.H., Pбez M. J. Computational Problems for Physics with Guided Solutions Using Python. – CRC Press, 2018. – 389 p.
13. Linge S., Langtangen H. P. Programming for Computations – Python. Second ed. – Springer, 2020. – 224 p.
14. Morgan W.A., English L.Q. VPython for Introductory Mechanics: Complete Version [Електронний ресурс] – URL: https://scholar.dickinson.edu/vpythonphysics/1 (дата звернення 03.06.2022)
15. Olenick R., Case D., Peping E., Spearman W. VPython Simulations in a Computational Physics Course [Електронний ресурс] – URL: https://doi.org/10.13140/RG.2.1.3661.7202 (дата звернення 03.06.2022)
16. Philhour B. Physics through GlowScript – an Introductory Course [Електронний ресурс] – URL: https://bphilhour.trinket.io/physics-through-glowscript-an-introductory-course (дата звернення 03.06.2022)
17. Pravdin S., Ushenin K., Sozykin A., Solovyova O. Human heart simulation software for parallel computing systems // Procedia Computer Science. – 2015. – Vol. 66. – Pp. 402–41. doi: 10.1016/j.procs.2015.11.046
18. Schroeder D. V. Physics Simulations in Python: [Електронний ресурс]. – URL: https://physics.weber.edu/schroeder/scicomp/PythonManual.pdf (дата звернення 03.06.2022)
19. Verlet L. Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules // Physical Review. – 1967. – Vol. 159, № 1. – Pp. 98–103. doi: 10.1103/PhysRev.159.98
20. Wang J. Computational modeling and visualization of physical systems with Python. – Hoboken, NJ: John Wiley & Sons, 2015. – 475 p.
21. Weichen Qiu VPython – Visual Python, [Електронний ресурс] – URL: https://engcourses-uofa.ca/wp-content/uploads/Visual-Python-VPython-ver-2.pdf (дата звернення 03.06.2022)
22. Авдюшев В.А. Численное моделирование орбит небесных тел. – Томск: Издательский Дом Томского государственного университета, 2015. – 336 с.

23. Асланов В. С., Алексеев А. В. Концепции математического моделирования механических систем и процессов. – Самара: Изд-во Самарского университета, 2017. – 128 с.
24. Вабищевич П.Н. Численные методы: Вычислительный практикум. – М.: Едиториал УРСС, 2021. – 320 с.
25. Забавникова Т.А. Масс - пружинные модели физики твердого тела в МАТЛАБ R19b и Simulink [Електронний ресурс] – URL: https://hub.exponenta.ru/post/mass-pruzhinnye-modeli-fiziki-tverdogo-tela-v-matlab-r19b-i-simulink567 (дата звернення 03.06.2022)
26. Николаев С. Н. Нелинейная система масс-с-пружинками для моделирования больших деформаций мягких тканей // Научно-технический вестник информационных технологий, механики и оптики. – 2013, № 5 (87). – С. 88–93.
27. Поттер Д. Вычислительные методы в физике. – М.: Мир, 1975. – 392 с.
28. Страшнов Е. В., Торгашев М. А., Тимохин П. Ю. Моделирование пружин в системах виртуального окружения с помощью метода мягких ограничений // Информационные технологии и вычислительные системы. – 2017. – Выпуск 3. – С. 70–78.
29. Сусликов П. И., Фроловский В. Д. Модели, методы, алгоритмы построения квазиразверток поверхностей // Евразийский Союз Ученых = Eurasian Uninon of Scientists. – 2015. – № 4 (13), ч. 5. – С. 54–56.
30. Канал «Let’s code|Physics», плейлист “VPython for beginners” [Електронний ресурс] – URL: https://www.youtube.com/playlist?list=PLdCdV2GBGyXOnMaPS1BgO7IOU_00ApuMo (дата звернення 03.06.2022)

ДОДАТКИ

Додаток А. Код класу Mass

```
#-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-#
#     Class Mass, (c) 2022                     |
#         Описує кульку                        |
#-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-#

from settings import *

class Mass:
    """ Клас, що описує кульку """
    def __init__(self, pos, num=0, radius = BALL_R,
                 color = BALL_COLOR,
                 mass = BALL_MASS):
        self.pos = pos
        self.mass = mass
        self.id = num
        self.v = vec(0,0,0)
        self.acc = vec(0,0,0)
        self.springs = []
        self.ball = sphere(pos = pos,
                           radius = radius,
                           color = color)

    def add_spring(self, new_s):
        self.springs.append(new_s)

    def redraw(self):
        self.ball.pos = self.pos

    def a(self):
        f = vector(0,0,0)
        for spr in self.springs:
            if self == spr.m1:
                another = spr.m2
            else:
                another = spr.m1
            r = another.pos - self.pos
            delta_1 = mag(r) - spr.length
            f += spr.k * delta_1 * hat(r)
            return f/self.mass + g * vector(0,-1,0)
```

Додаток Б. Код класу Spring

```
#-=========================-#
#   Class Spring, (c) 2022      |
#         Описує пружину        |
#-=========================-#

from settings import *

class Spring:
    """ Клас, що описує пружину """
    def __init__(self, m1, m2, k = HELIX_STIFFNESS,
                 radius = HELIX_R, coils = 25,
                 thickness = HELIX_R/2.5):
        self.m1 = m1
        self.m2 = m2
        self.length = mag(m1.pos - m2.pos)
        self.k = k
        self.spr = helix(pos = m1.pos,
                         axis = m2.pos - m1.pos,
                         radius = radius,
                         thickness = thickness,
                         coils = coils)

    def cur_length(self):
        return(mag(self.m2.pos - self.m1.pos))

    def redraw(self):
        self.spr.pos = self.m1.pos
        self.spr.axis = self.m2.pos - self.m1.pos
```

Додаток В. Код класу Solid

```
#-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-#
#     Class Solid, (c) 2022      |
# Описує пружинну модель цілком |
#-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-#
from settings import *
from Mass import *
from Spring import *

class Solid:
    """ Клас, що описує пружинну модель цілком """
    def __init__(self,coords,links=None):
        self.body = []
        self.coords = [tmp for tmp in coords]
        tmp = 0
        for v in coords:

self.body.append(Mass(pos=vector(v[0],v[1]+BALL_R,v[2]),num
=tmp))
            tmp += 1
        self.n_mass = len(self.body)
        if links==None:
            self.links = [[0 for i in range(self.n_mass)]
for j in range(self.n_mass)]
        else:
            self.links = links
        self.springs = []
        for i in range(self.n_mass):
            for j in range(i+1, self.n_mass):
                if self.links[i][j]:
                    new_spring =
Spring(self.body[i],self.body[j])
                    self.springs.append(new_spring)
                    self.body[i].add_spring(new_spring)
                    self.body[j].add_spring(new_spring)

    def addSpring(self,v1,v2):
        """ Метод для додавання пружини між кулями з
номерами v1 та v2 """
        new_spring = Spring(self.body[v1],self.body[v2])
        self.springs.append(new_spring)
        self.body[v1].add_spring(new_spring)
        self.body[v2].add_spring(new_spring)
        self.links[v1][v2]=1
        self.links[v2][v1]=1
```

```
        self.body[v1].ball.color = BALL_COLOR
        self.body[v2].ball.color = BALL_COLOR

    def delSpring(self,v1,v2):
        """ Метод для видалення пружини між кулями з
номерами v1 та v2 """
        for sp in self.springs:
            if (sp.m1.id == sel[0] and sp.m2.id == sel[1])
or (sp.m1.id == sel[1] and sp.m2.id == sel[0]):
                break
        self.body[v1].springs.remove(sp)
        self.body[v2].springs.remove(sp)
        sp.spr.visible = False
        self.springs.remove(sp)
        self.links[v1][v2]=0
        self.links[v2][v1]=0
        self.body[v1].ball.color = BALL_COLOR
        self.body[v2].ball.color = BALL_COLOR

    def getIsoList(self):
        """ Метод для пошуку ізольованих куль (без жодної
пружини)Повертає список ізольованих вершин """
        iso = []
        for v in range(self.n_mass):
            if self.isIsolated(v):
                iso.append(v)
        return iso

    def isIsolated(self,v):
        """ Метод для визначення чи куля ізольована """
        if sum(self.links[v]) == 0:
            return True
        else:
            return False

    def removeIsolated(self,v):
        """ Метод для видалення ізольованої кулі """
        del(self.coords[v])
        self.body[v].ball.visible = False
        del(self.body[v])
        self.n_mass = len(self.body)
        del(self.links[v])
        for row in self.links:
            del(row[v])
```

```
    def removeAllIsolated(self):
        """ Метод для видалення всіх ізольованих куль """
        iso = self.getIsoList()
        iso.reverse()
        for id in iso:
            self.removeIsolated(id)

    def rePaintIsolated(self,col):
        """ Метод для виділення ізольованих куль """
        iso = self.getIsoList()
        for id in iso:
            self.body[id].ball.color = col

    def rise(self,h):
        """ Зміна положення моделі по y """
        for m in self.body:
            m.pos.y += h
            m.redraw()
        for s in self.springs:
            s.redraw()

    def rePaintAll(self,col):
        for v in self.body:
            v.ball.color = col

    def rePaintBall(self,v,col):
        self.body[v].ball.color = col

    def energy(self):
        """ Обчислення енергії системи """
        e1 = sum(m.mass * (m.pos.y * g + mag2(m.v)/2) for m
in self.body)
        e2 = sum(spr.k * (spr.cur_length()-spr.length)**2/2
for spr in self.springs)
        return e1 + e2

    def com_y(self):
        total_mass = sum(m.mass for m in self.body)
        y = sum(m.mass * m.pos.y for m in self.body)
        return y/total_mass

    def animate(self,param):
        """ Основний метод інтегрування та анімації """
        t = param.t
        T = param.T
```

```
        DELTA_T = param.DELTA_T
        repeat = param.ANIMATE
        display = 0
        while t < T and repeat:
            t += DELTA_T
            if param.INTMETHOD == 1: # Метод Ейлера -
Кромера
                for m in self.body:
                    m.v += m.a() * DELTA_T
                for m in self.body:
                    m.pos += m.v * DELTA_T
                    if m.pos.y < 0: #ground.pos.y +
ground.size.y:
                        m.v.y = abs(m.v.y)
            elif param.INTMETHOD == 2: # Метод Ейлера
                for m in self.body:
                    m.acc=m.a()
                for m in self.body:
                    m.pos += m.v*DELTA_T
                    if m.pos.y < 0: #ground.pos.y +
ground.size.y:
                        m.v.y = abs(m.v.y)
                for m in self.body:
                    m.v += m.acc*DELTA_T
            elif param.INTMETHOD == 3: # Метод середньої
точки (Middle point)
                for m in self.body:
                    m.acc = m.a()
                    m.v += m.acc*DELTA_T
                for m in self.body:
                    m.pos += m.v*DELTA_T +
0.5*m.acc*DELTA_T**2
                    if m.pos.y < 0: #ground.pos.y +
ground.size.y:
                        m.v.y = abs(m.v.y)
            display += 1
            if display == 50:
                display = 0
                for m in self.body:
                    m.redraw()
                for s in self.springs:
                    s.redraw()
                f1.plot([t, self.energy()])
                f2.plot([t, self.com_y()])
            repeat = param.ANIMATE
```

Додаток Г. Код програми ShapeEditor

```
#-=========================-#
#                           |
# ShapeEditor, v.1.3, (c) 2022  |
#                           |
#-=========================-#

from settings import *
from Mass import *
from Spring import *
from Solid import *

stdbody = None

#
# Координати вершин шаблонів
#
prizm3 =
[[0,0,0],[d,0,0],[d/2,0,sqrt(3)/2*d],[d/2,0,sqrt(3)/6*d],

[0,sqrt(6)/3*d,0],[d,sqrt(6)/3*d,0],[d/2,sqrt(6)/3*d,sqrt(3
)/2*d],[d/2,sqrt(6)/3*d,sqrt(3)/6*d]]
cube = [[0,0,0],[d,0,0],[d,0,d],[0,0,d],[d/2,0,d/2],
        [0,d,0],[d,d,0],[d,d,d],[0,d,d],[d/2,d,d/2]]

#
# Налаштування 3D-сцени
#
scene = canvas(align="left",width = 800, height = 600,
               background = vector(0.6,0.8,1.0),
               title = "<h2 align='center'>Редактор форм,
v.1.3</h2>")
scene.range = 5*h
scene.camera.pos = vector(d,1.5*h,h)
scene.forward = vector(0.0,0.1,-1)

ground = box(pos = vector(0.20,-0.001,0.20),
             size = vector(0.5,0.001,0.5),
             shininess = 0, color = color.green)

oY = arrow(pos=vector(0,0,0),axis =
vector(0,0.25,0),shaftwidth = 0.001,
      headwidth = 0.005,
      headlength = 0.01)
```

```
sel = []
def selectMass(ev):
    """ Вибір кульок для додавання/видалення пружин """
    obj=scene.mouse.pick
    for v in stdbody.body:
        if v.ball==obj:
            tmp = v.id
            if tmp in sel:
                sel.remove(tmp)
                obj.color = BALL_COLOR
            elif tmp not in sel and len(sel)<2:
                sel.append(v.id)
                obj.color = SEL_BALL_COLOR
    if len(sel)==2:
        if stdbody.links[sel[0]][sel[1]] == 1:
            delh.disabled = False
        else:
            addh.disabled = False
    elif len(sel)<2:
        addh.disabled = True
        delh.disabled = True

scene.bind('click',selectMass)

#
# Интерфейс
#
# Вибір шаблону
#
filler = ' '*5
scene.append_to_caption('<hr>')
scene.append_to_caption(filler)
scene.append_to_caption('Виберіть шаблон')
scene.append_to_caption(filler)

def listBox():
    global stdbody
    if pattern.index == 0:
        reset()
    else:
        if pattern.index == 1:
            stdbody=Solid(prizm3)
        elif pattern.index == 2:
            stdbody=Solid(cube)
        generateButton.disabled = False
```

```
        resetButton.disabled = False
        chiso.disabled = False
        isolatedButton.disabled = False

pattern = menu(choices=['Виберіть шаблон','Тригранна
призма','Куб з центром'],
     bind=listBox,selected=0)
scene.append_to_caption('<br><hr>')
scene.append_to_caption(filler)

#
# Додавання та видалення пружин
#
wtext(text = "Для додавання пружини<br> виділіть мишею пару
вершин та<br>\
натисніть кнопку 'Пружина +'<br>")
scene.append_to_caption(filler)
wtext(text = "Для видалення пружини<br> виділіть мишею пару
вершин та<br>\
натисніть кнопку 'Пружина -'<br>")

def addHelix():
    stdbody.addSpring(sel[0],sel[1])
    sel.clear()
    addh.disabled = True
    delh.disabled = True

scene.append_to_caption('<br>')
scene.append_to_caption(filler)
addh = button(bind=addHelix, text = 'Пружина +',
              pos=scene.caption_anchor,
              disabled = True)

def delHelix():
    stdbody.delSpring(sel[0],sel[1])
    sel.clear()
    addh.disabled = True
    delh.disabled = True

scene.append_to_caption(filler)
delh = button(bind=delHelix, text = 'Пружина -',
              pos=scene.caption_anchor,
              disabled = True)
scene.append_to_caption('<br><hr>')
scene.append_to_caption('<br>')
```

```
#
# Показ та видалення ізольованих куль
#
def showIsolated():
    if chiso.checked == True:
        stdbody.rePaintIsolated(ISOL_BALL_COLOR)
    elif chiso.checked == False:
        stdbody.rePaintAll(BALL_COLOR)
        for v in sel:
            stdbody.rePaintBall(v,SEL_BALL_COLOR)    ##
список вибраних в класс

def delIsolated():
    stdbody.removeAllIsolated()

scene.append_to_caption(filler)
chiso = checkbox(bind = showIsolated, text = 'Показати
ізольовані', checked = False)
chiso.disabled = True
scene.append_to_caption(filler)
isolatedButton = button(bind = delIsolated, text = '
Видалити ізольовані ')
isolatedButton.disabled = True

#
# Збереження та скиданння
#
scene.append_to_caption('<br><hr>')
scene.append_to_caption('<br>')

def generate(): # Збереження
    win=Tk()
    #win.withdraw() # Стандартний спосіб сховати вікно, але
тут не працює
    win.geometry("0x0")
    win.resizable(0,0)
    try:
        file_name = fd.asksaveasfilename(
            defaultextension = '.mod',
            initialfile = 'noname',
            filetypes = (("mod files", "*.mod"),("All
files", "*.*")))
        with open(file_name, 'wb') as filehandle:
            pickle.dump(stdbody.coords, filehandle)
```

```
                pickle.dump(stdbody.links, filehandle)
        except OSError:
            pass
        win.destroy()

scene.append_to_caption(filler)
generateButton = button(bind=generate, text = ' Створити!
')
generateButton.disabled = True

def reset(): #Скидання
    pattern.selected = 'Виберіть шаблон'
    generateButton.disabled = True
    resetButton.disabled = True
    chiso.checked = False
    chiso.disabled = True
    isolatedButton.disabled = True
    for obj in scene.objects:
        if isinstance(obj, sphere) or
isinstance(obj,helix):
            obj1=obj
            obj.visible = False
            del(obj1)

scene.append_to_caption(filler)
resetButton = button(bind=reset, text = '  Скинути  ')
resetButton.disabled = True
```

Додаток Д. Код програми SpringModel

```
#-=============================-#
#                               |
# SpringModel, v.1.0.0, (c) 2022  |
#                               |
#-=============================-#

from settings import *
from Mass import *
from Spring import *

#
# Опис класів
#
class Param:
    """ Інформаційний клас для збереження параметрів """
    def __init__(self):
        self.d = 0.1
        self.h = 3*d
        self.BALL_R = 0.15*d
        self.BALL_COLOR = color.red
        self.SEL_BALL_COLOR = color.yellow
        self.ISOL_BALL_COLOR = color.red
        self.BALL_MASS = 0.1
        self.HELIX_R = BALL_R/3
        self.HELIX_STIFFNESS = 100
        self.g = 9.8
        self.t = 0.0
        self.T = 5.0
        self.DELTA_T = 0.00001
        self.T_END = 5.0
        self.INTMETHOD = 1
        self.ANIMATE = False
        self.ISLOAD = False

    def info(self):
        print('Маса кульки', self.BALL_MASS)
        print('Жорсткість пружини:', self.HELIX_STIFFNESS)
        print('Час  інтегрування', self.T)
        print('Крок інтегрування', self.DELTA_T)

class Solid:
    """ Клас, що описує пружинну модель цілком """
    """ Код класу наведено у Додатку В """
```

```
#
# Налаштування 3D-сцени
#
scene = canvas(width = 800, height = 600,
               background = vector(0.6,0.8,1.0),
               title = '<h2 align="center">Моделювання
руху<br>механічної системи</h2>',align="left")
ground = box(pos = vector(0,-0.1,0),
             size = vector(100,0.1,100),
             shininess = 0, color = color.green)
scene.range = 5*h
scene.camera.pos = vector(d,1.5*h,h)
scene.forward = vector(0.0,0.1,-1)
gd = graph(width = 600, height = 200,
           title = 'Енергія/центр мас системи',
           xmin = 0, xmax = T_END, align="left")
f1 = gcurve(color = color.red)
f2 = gcurve(color = color.blue)
par = Param()
stdbody = None

#
# Інтерфейс
#
filler = ' '*5
#
# Відкриття файлу для завантаження
#
def load(): #load
    global stdbody
    global ISLOAD
    win=Tk()
    #win.withdraw() # Стандартний спосіб сховати вікно, але
тут не працює
    win.geometry("0x0")
    win.resizable(0,0)
    tmp=[]
    try:
        file_name = fd.askopenfilename(
            defaultextension = '.mod',
            filetypes = (("mod files", "*.mod"),("All
files", "*.*")))
        if file_name!='':
            with open(file_name, 'rb') as filehandle:
                coords = pickle.load(filehandle)
```

```
                links = pickle.load(filehandle)
                stdbody = Solid(coords,links)
                stdbody.rise(par.h)
            par.ISLOAD = True
            startModel.disabled = False
            resetModel.disabled = False
        except OSError:
            pass
        finally:
            win.destroy()

def listBox():
    global INTMETHOD
    if method.index == 1:
        scene.title='<h2 align="center">Метод Ейлера-
Кромера</h2>'
        INTMETHOD = 1
        par.INTMETHOD = 1
    elif method.index == 2:
        scene.title='<h2 align="center">Метод Ейлера</h2>'
        INTMETHOD = 2
        par.INTMETHOD = 2
    elif method.index == 3:
        scene.title='<h2 align="center">Метод
апроксимації<br>за середньою точкою</h2>'
        INTMETHOD = 3
        par.INTMETHOD = 3

def masa():
    global BALL_MASS
    par.BALL_MASS=mass.number
    BALL_MASS = mass.number

def stiff():
    global T
    par.HELIX_STIFFNESS = stiff.number
    HELIX_STIFFNESS = stiff.number

def step():
    par.DELTA_T= step.number
    DELTA_T = step.number

def time():
    par.T=time.value
    T = time.value
```

```
def hei():
    tmp = height.number
    if stdbody != None:
        stdbody.rise(-par.h+tmp)
    par.h = tmp
    h = tmp

def start():
    if par.ISLOAD == True:
        height.disabled = True
        model.disabled = True
        method.disabled = True
        mass.disabled = True
        stiff.disabled = True
        step.disabled = True
        time.disabled = True
        gd.xmax = time.value
        par.ANIMATE = True
        f1 = gcurve(color = color.red)
        f2 = gcurve(color = color.blue)
        stdbody.animate(par)
    else:
        pass

def reset():
    par.ANIMATE = False
    scene.title = '<h2 align="center">Моделювання
руху<br>механічної системи</h2>'
    height.disabled = False
    model.disabled = False
    method.disabled = False
    method.index = 0
    mass.disabled = False
    stiff.disabled = False
    step.disabled = False
    time.disabled = False
    for obj in scene.objects:
        if isinstance(obj, sphere) or
isinstance(obj,helix):
            obj1=obj
            obj.visible = False
            del(obj1)
    startModel.disabled = True
    resetModel.disabled = True
```

```
    f1.delete()
    f2.delete()
    pass

scene.append_to_caption('<hr>')
scene.append_to_caption(filler)
wtext(text = 'Виберіть файл зі збереженою моделлю:')
scene.append_to_caption(filler)
model=button(bind=load, text = 'Відкрити файл моделі')
scene.append_to_caption(filler)
scene.append_to_caption('<hr>')
scene.append_to_caption(filler)
scene.append_to_caption('Виберіть метод інтегрування')
scene.append_to_caption(filler)
method = menu(choices=['Виберіть метод','Метод Ейлера-
Кромера','Метод Ейлера','Метод середньої точки'],
     bind=listBox,selected=0)
scene.append_to_caption('<br><hr>')
scene.append_to_caption(filler)
wtext(text = 'Додаткові парметри моделювання:<br><br>')
scene.append_to_caption(filler)
wtext(text = 'Висота над землею:')
height =
winput(bind=hei,type='numeric',text=str(round(par.h,3))) #
Висота над поверхньою
scene.append_to_caption('<br><br>')
scene.append_to_caption(filler)
wtext(text = 'Маса кульки:')
mass =
winput(bind=masa,type='numeric',text=str(par.BALL_MASS)) #
Маса кульки
scene.append_to_caption('<br><br>')
scene.append_to_caption(filler)
wtext(text = 'Жорсткість пружини:')
stiff =
winput(bind=stiff,type='numeric',text=str(par.HELIX_STIFFNE
SS)) # Жорсткість пружини
scene.append_to_caption('<br><br>')
scene.append_to_caption(filler)
wtext(text = 'Крок інтегрування:')
step =
winput(bind=step,type='numeric',text=str(par.DELTA_T)) #
Крок інтегрування
scene.append_to_caption('<br><br>')
scene.append_to_caption(filler)
```

```
wtext(text = 'Час інтегрування:')
time = slider(bind=time,min=1.0, max=10.0,
step=0.5,value=5.0) # Час інтегрування
scene.append_to_caption('<hr>')
scene.append_to_caption(filler)
startModel = button(bind=start, text = '  Запуск
',disabled=True)
scene.append_to_caption(filler)
resetModel = button(bind=reset, text = '   Сброс
',disabled=True)
scene.append_to_caption('<br><br>')
```