МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Кафедра програмної інженерії

КУРОСВИЙ ПРОЄКТ

Розробка інформаційної системи чисельного розв'язання диференційного рівняння

з дисципліни «Об’єктно-орієнтоване програмування»

ПОЯСНЮВАЛЬНА ЗАПИСКА

КППІ. 2101170.00.00.00 ПЗ

Студент групи ПЗ-20-1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. О. Маринчак

(підпис, дата)

Керівник роботи

канд. техн. наук, доцент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ю. В. Форкун

(підпис, дата)

2021

ЗМІСТ

[ВСТУП 3](#_Toc91344026)

[1. Дослідження предметної області 4](#_Toc91344027)

[1.1.Характеристика функціональної структури предметної області 4](#_Toc91344028)

[1.2 Аналіз останніх публікацій, досліджень та існуючих рішень 6](#_Toc91344029)

[1.3 Постановка задачі та перелік задач для реалізації 8](#_Toc91344030)

[2. ПроЄктування структури застосування 13](#_Toc91344031)

[2.1 Математична модель об’єкту проЄктування 13](#_Toc91344032)

[2.2 Розробка структури інформаційної системи 17](#_Toc91344033)

[2.3 Вибір засобів розробки інформаційної системи 18](#_Toc91344034)

[3. Програмна реалізація 20](#_Toc91344035)

[3.1. Структура і функціональне призначення модулів системи, їх взаємозв’язок 20](#_Toc91344036)

[3.2 Розробка програмних модулів 28](#_Toc91344037)

[3.3 Інструкція користувача 36](#_Toc91344038)

[3.4 Вимоги до технічних засобів 37](#_Toc91344039)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 40](#_Toc91344040)

[ДОДАТОК А 41](#_Toc91344041)

[Додаток Б 42](#_Toc91344042)

ВСТУП

Метод Монте-Карло належати до групи чисельних методів для дослідження випадкових процесів. Окреме місце займає метод Монте-Карло для визначення площ плоских фігур. Наразі метод не конфігурує в сучасній математиці, оскільки він вважається досить примітивним серед даної групи методів і ніяк не досліджується, але самі методи Монте-Карло активно використовуються для вивчення не лише випадкових процесів, а й для статичних.

Метою КП є вивчення на закріплення навичок програмування з використання методів мови C# з використанням технології, що реалізує графічний користувацький інтерфейс. Вивчення математичних методів групи Монте-Карло. Розробка програмного додатка для проєктування та автоматизації процесу розв'язання задач методом Монте-Карло для обчислення площ плоских фігур.

Курсовий проєкт (Далі КП) розроблено за допомогою методів мови програмування C# та платформи, що реалізує графічний користувацький інтерфейс, WPF.

Для обчислення було розроблено декілька алгоритмів, які пов'язані на роботі з трикутниками. Дані алгоритми названі "Метод трикутників" та використовуються для визначення, чи знаходиться точка всередині полігону; для обрахунку площі полігону за допомогою сум площ трикутників всередині полігону.

# 1. Дослідження предметної області

# 1.1.Характеристика функціональної структури предметної області

Метод Монте-Карло - це загальна назва групи методів для обчислення результату дослідження методів найбільшої кількості ітерацій (Повторних спроб досліджень). Метод носить назву на честь міста Монте-Карло, яке відоме на весь світ своєю кількістю азартних закладів, в тому числі казино. Відштовхуючись від азартної тематики, основним способом дослідження є довільна кількість спроб для обчислення які є випадковими, а результат виноситься на основі загальної суми спроб і ніколи не береться за основу одна спроба (Інтеграція). Дана група методі належати до розділу методів для дослідження випадкових процесів і не тільки. Одним з прикладів може слугувати алгоритм Бюффона для обчислення числа π, а також алгоритм для обчислення площ фігур (Не тільки плоских) який має велику популярність при вступі до вивчення деяких з областей математики, математичного аналізу тощо.

Останнім часом, з розвитком технологій, збільшенню максимальної та середньої обчислювальної потужності технічних засобів процес досліджень різного роду з використанням даних методів, крім того, що набуває популярності, тобто, майже кожен може спробувати особисто дослідити деякі явища; також багатократно збільшується точність різного роду обчислень при дослідах. Також важливо розуміти, що сам метод не ставить ніяких глобальних питань, а лише пропонує новий погляд на методи рішення деякого роду задач. Метод позиціонується як чисельний метод прикладної математики для вирішення чисельних розв’язків задач з можливістю графічного відображення. Метод використовується, як було сказано вище, для обчислення числа π, для обчислення значень інтегралів з чисельним розв’язком, систем диференціальних та інтегральних рівнянь та багато іншого. Згідно з завданням курсового проєктування (Далі КП) було вибрано та досліджено метод Монте-Карло для обчислення площ плоских фігур, яких буде описано в наступних сторінках.

Серед теоретично-графічних методів дослідження потрібно також наголосити на задачах, які потребуються методів автоматизації досліджень. Загалом даний метод належати до розділу методів імітаційного моделювання, які діляться лише на дві групи, однією з них є сам метод Монте-Карло, а точніше, група методів, який також можна назвати методом статичних випробувань. Друга група - статистичне моделювання, яке охоплює низку підгруп для статичного моделювання які описуються такі моделі:

— Лінійна регресія (OLS)

— Регресії на бінарні змінні

— Авторегресійна модель

— Система одночасних рівнянь (SEM)

— Модель лінійної ймовірності (LMP)

— Логіт модель (Logit)

— Пробіт модель (Probit)

Моделювання за допомогою методу Монте-Карло вирішує ряд питань які зв'язані з методами визначення площі довільних фігур. Саме пошук площі довільних та неправильних фігур завдає багато клопоту через свою важкість, оскільки майже немає загальних формул та правил для знаходження площ довільних фігур. Дана проблема частково вирішується за допомогою інтегрування, але не повністю, оскільки є значна кількість "Невирішуваних" інтегралів, так саме як для площини, яка описана довільним набором точок досить складно підібрати потрібне формулювання або, якщо фігура описана досить великою кількістю графіків, то це буде призводити до збільшення споживання ресурсів обчислювальної техніки або доставляти значні проблеми при дослідженнях "На папері".

Натомість метод Монте-Карло пропонує більш оптимізований спосіб для пошуку значень складних обчислень використовуючи значно менше ресурсі. Логічно, що даний метод далеко не являється кращим при дослідженні площ простих фігур, які легко визначаються за формулами, він надає можливість для статистичного дослідження складних задач.

За допомогою програмного додатка, що є додатком до даної пояснювальної записки (Далі ПЗ) можливо власноруч визначити площу довільної фігури з деякою точністю. Користувачу пропонується провести дослідження та зробити особисті висновки.

# 1.2 Аналіз останніх публікацій, досліджень та існуючих рішень

При пошуку останніх публікацій на тему даного курсового проєкту (Далі КП) - метод Монте-Карло для обчислення площ плоских фігур, вам потрібно буде провести багато часу, щоб знайти якісь результати досліджень, оскільки на даний момент не використовується на практиці а, як було описано вище, використовується лише для навчання в різних сферах. На разі ми робимо технічно-навчальну програму для вивчення алгоритмізації та освоєння технології WPF та мови C# в цілому.

Метод Монте-Карло використовується не лише для дослідження та розв'язання подібних задач (Обчислення площ плоских фігур) а й для досить великої кількості інших тематичних задач, сенс яких виходить за рамки стандартної математики та повчальних завдань. На рисунку 1.1 можна побачити приклад використання методу Монте-Карло для алгоритмічних задач з використанням диференціальних рівнянь другого порядку.

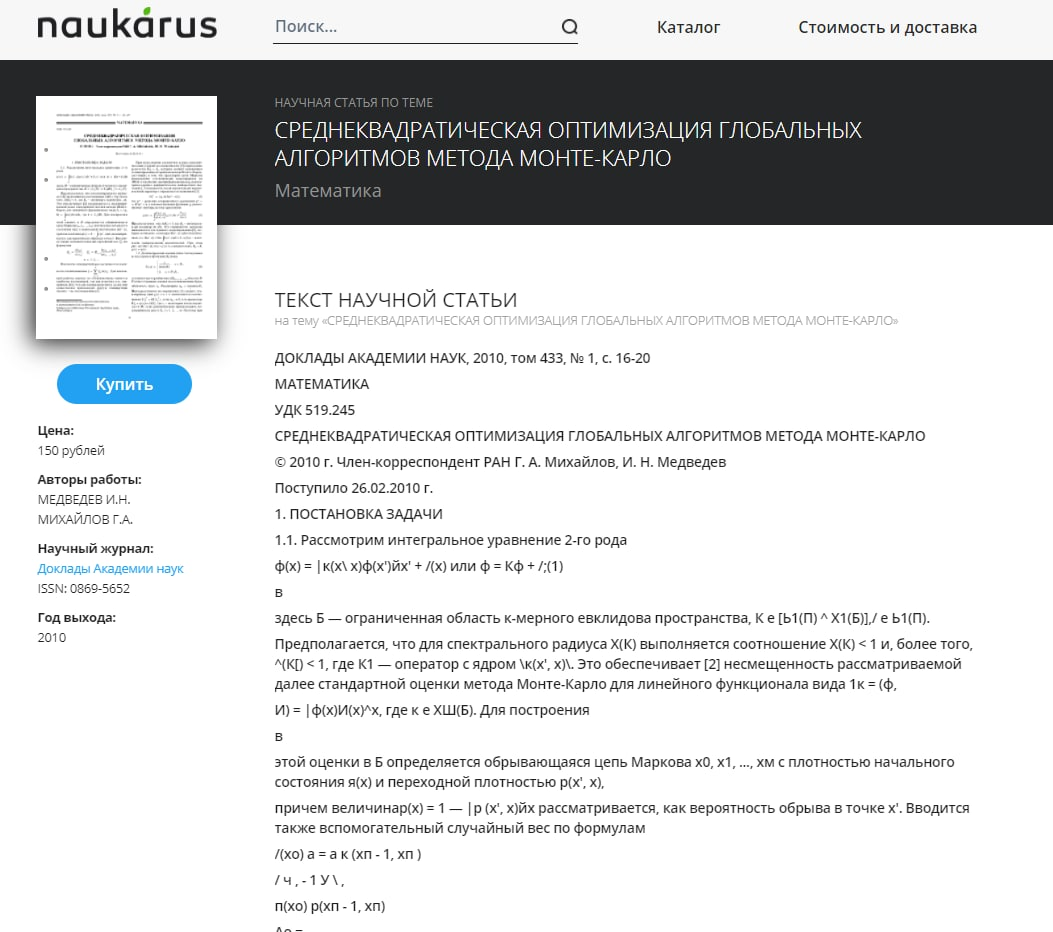


Рисунок 1.1 – Монте-Карло для глобальних алгоритмів

Далі на рисунку 1.2 наведено приклад використання методу у сфері оптики та спектроскопії.

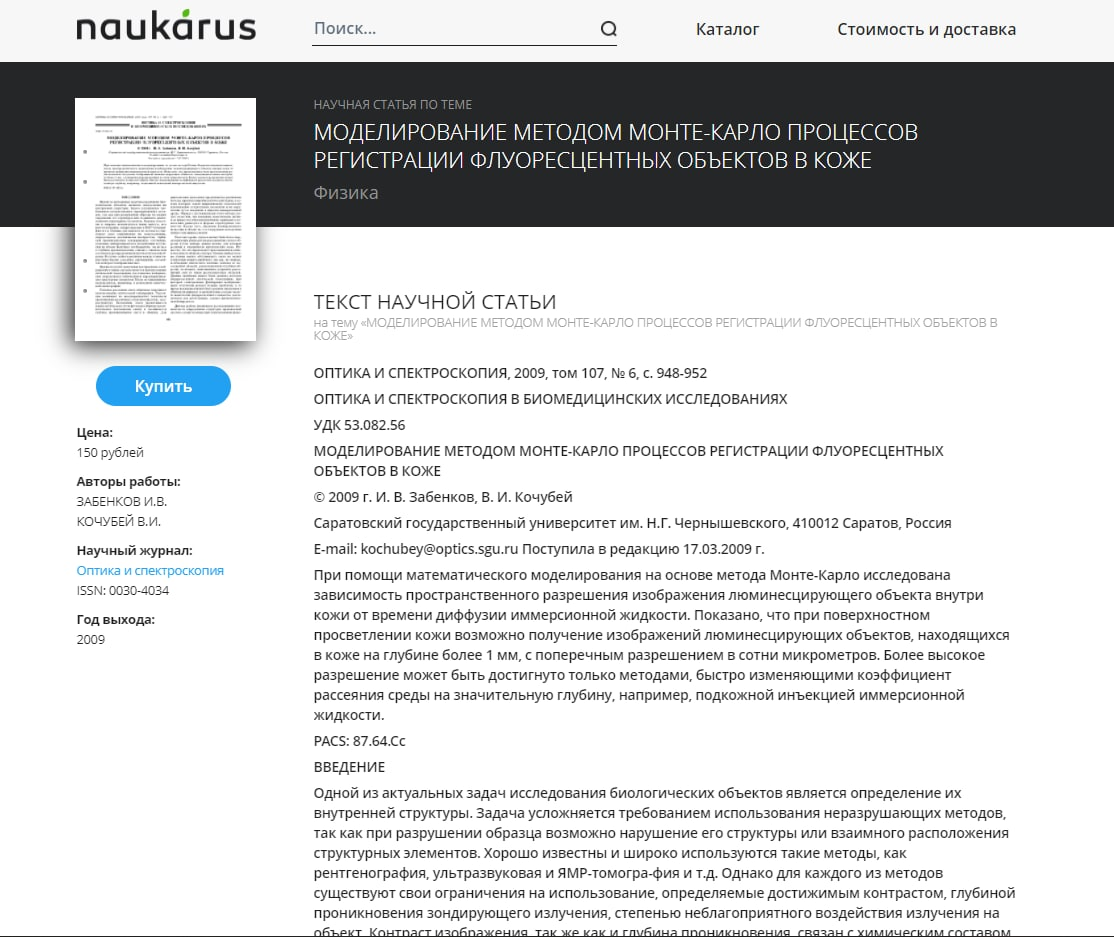


Рисунок 1.2 – Використання методу в медичній сфері

А також використання метод у сфері фізики зображено на рисунку 1.3.

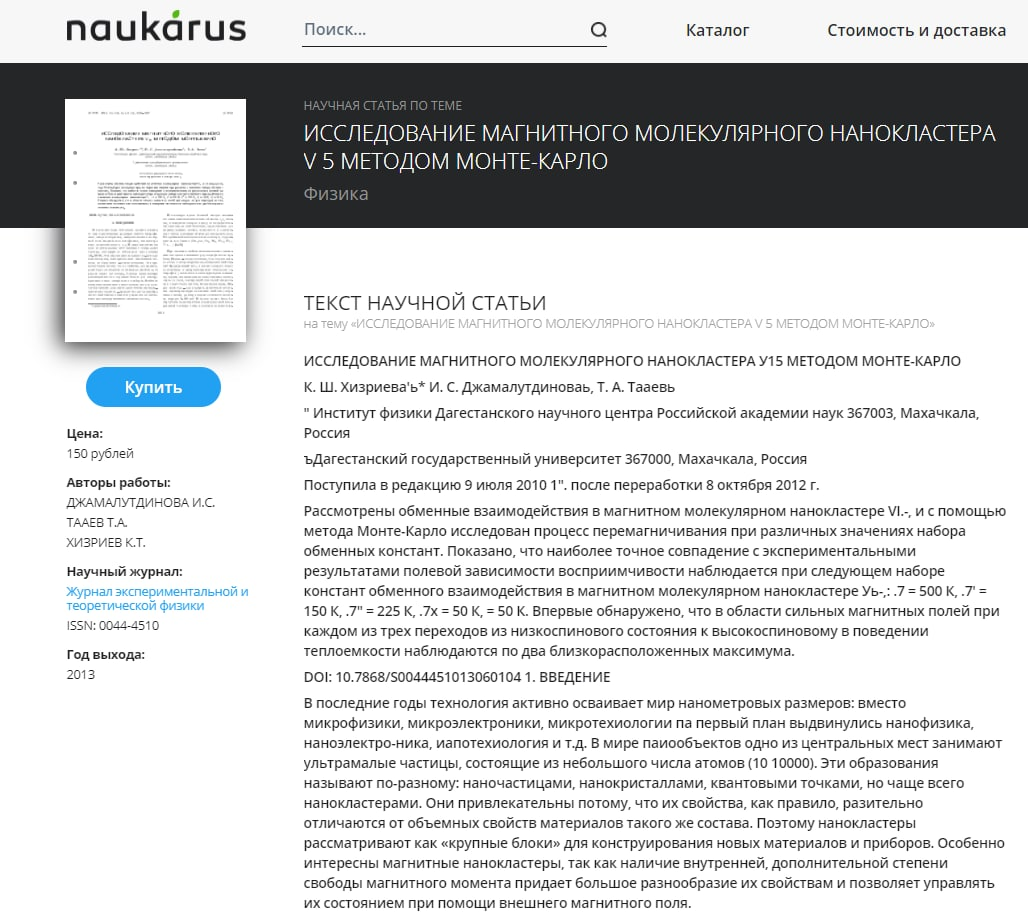


Рисунок 1.3 – Використання методу в сфері фізики

Звідси ми приходимо до висновку, що наразі ніхто на серйозному рівні не використовує даний метод для розв'язання лінійних задач.

Для власного проєкту ми будемо використовувати платформу WPF на мові C# з використанням системних бібліотек. Для побудови структури програмного додатка ми будемо використовувати користувацькі класи, які базуються на основі системної стандартної бібліотеки System. Також замість звичайних упорядкованих елементів у вигляді масивів ми будемо використовувати системні списки типу Generic - List<> для зручного упорядкування елементів.

# 1.3 Постановка задачі та перелік задач для реалізації

Актуальність вирішуваної задачі стає мінімальною при дослідженні сторонніх ресурсів, які присвячені розв'язання задач методом Монте-Карло в глобальних цілях. Функціонал розробленого програмного додатка допоможе швидко та з мінімальними затратами знаходити та обраховувати площі фігур. Більш детальніше про дослідження методом Монте-Карло далі в пункті 1.2 (Аналіз останніх публікацій, досліджень та існуючих рішень) на рисунках 1.1 - 1.3 та за посиланнями[3].

Дана задача призначена в більшості для пошуку площі неправильних фігур, оскільки чисельні методи розв'язання стають більш "Голодними" до обчислювальних ресурсів та громіздкість паперових записів додають багато проблем та збільшують час для обчислення. Автоматизація даного методу дозволяє в десятки, а то і в сотні раз швидше проводити обчислення площ та звіряти з деякою фактичною величиною. Також звірення з фактичною величиною в майбутньому можливо прибрати для більшої швидкості обчислення з більшою точністю. Процес вичислення фактичної площі фігури досить громіздкий та затратний зі сторони ресурсів ПК, але використовується як контрольне значення на перших етапах користування програмним додатком.

Задача була вирішена за допомогою інструментів платформи Windows Presentation Foundation та методів мови C#. Програмна реалізація дозволяє зробити дослідження по даній темі використовуючи графічні елементи отримання, обробки та виведення інформації. Такі елементи являти собою графічні елементи управління, за допомогою яких користувач може взаємодіяти з формою. Форма має наступні графічні елементи:

* Button
* Slider
* InkCanvas
* Label

Button - елемент управління який представляє кнопку. Елемент управління Button є нащадком класу ButtonBase який є батьківським класом для ряду інших елементів, які є "Братськими" для елемента кнопки. Загальна ієрархія сімейства класів ButtonBase зображена на рисунку 1.4. Також Елемент Button наслідує багато обробників подій для взаємодії та коректної роботи елементу. Обробник Click є базовим для взаємодії та показує саму суть роботи кнопки, оскільки в тілі обробника буде виконуватися код при натисканні кнопки.



Рисунок 1.4 – Наслідування від базового класу ButtonBase

Елемент Slider - елемент управління який являє собою повзунок на горизонтальній або вертикальній шкалі значень, який реагує на дії користувача. Шкала дозволяє вибрати користувачу деяке значення від 0 до максимального значення, яке зазначене програмістом. Також мінімальне значення може змінюватися, але значення 0 є стандартним. На шкалі можна включити наявність поділок та відображення поточного значення. Приклади використання елементу Slider можна побачити на рисунку 1.5. Також даний елемент має подію ValueChanged який буде виконувати деякі дії при зміні поточного значення.



Рисунок 1.5 – Елемент Slider

Label - не вважається елементом управління, а використовується для відображення інформації. На платформі WPF C# є такий елемент як LinkLabel який використовується в контексті як посилання на інший фрагмент та працює аналогічно html тегу <a></a>. Елемент можна використовувати як елемент управління та в деяких випадках замінювати ним елемент управління Button, оскільки Label має подію Click, тобто можна налаштувати взаємодію із користувачем за допомогою натискання клавіші миші.

InkCanvas - центральний елемент управління який за час роботи програми використовується найбільше. Даний являється досить потужним елементом для малювання користувачем та має досить потужний функціонал та режими взаємодії з полотном, серед яких:

– Ink - режим для вільного малювання по полотну, тобто, користувач має можливість малювати мишею або пером на місці яких з'являються штрихи. Режим за замовчуванням;

– GestureOnly - InkCanvas не дозволяє користувачеві малювати анотації, але привертає увагу до деяких визначених жестів (gesture), таких як переміщення пера в одному напрямку або підкреслення вмісту. Повний список жестів визначено у переліку System.Windows.Ink.ApplicationGesture;

– InkAndGesture - InkCanvas дозволяє користувачеві малювати штрихові анотації й також розпізнає зумовлені жести;

– EraseByStroke - InkCanvas видаляє штрих при натисканні, тобто виконує роль віртуальної гумки. Якщо користувач має перо, він може перейти в цей режим, використовуючи його зворотний кінець. (Визначити поточний режим можна, перевіривши значення доступного тільки для читання властивості ActiveEditingMode, а для зміни режиму, який використовується зворотним кінцем пера, необхідно модифікувати властивість EditingModeInverted);

– EraseByPoint - InkCanvas видаляю лише частину штриха при натисканні, на яку буде наведене перо або миша;

– Select - InkCanvas дозволяє користувачеві вибирати елементи, що зберігаються у колекції Children. Щоб вибрати елемент, користувач повинен натиснути на ньому або обвести "ласо" вибору навколо нього. Як тільки елемент вибраний, його можна переміщати, змінювати розмір або видаляти;

На момент виконання програми, елемент InkCanvas використовує режим взаємодії None та не перемикатися в інші режими. Для виведення даних на полотно використовується подія MouseLeftButtonDown в результаті роботи якої на полотно програмно додається точка або лінія.

# 2. ПроЄктування структури застосування

# 2.1 Математична модель об’єкту проЄктування

Чисельні методи (далі ЧМ) – методи наближеного або точного розв’язування задач прикладної математики, які ґрунтуються на побудові послідовності дій над скінченною множиною чисел. Згідно з основними вимогами, чисельні методи мають бути стійкими та збіжними. Основне питання теорії чисельних методів: отримання чисельних методів, які задовольняють вимоги високої точності, стійкості та економічності. Знаходження чисельних методів, що задовольняють ці вимоги, є складною задачею оптимізації чисельних методів. Статистичне опрацювання експериментальних даних зазвичай ґрунтується на граничних теоремах теорії ймовірностей та обчисленні порівняльних оцінок. Однак, для підвищення якості оцінок необхідна велика кількість даних, обсяг обчислень може виявитися дуже великим. Чисельні методи націлені на скорочення кількості обчислень при збереженні якості результатів. До найбільш ефективних чисельних методів в цій галузі відноситься метод Монте-Карло. Для розв’язування задач апроксимації та обчислення функцій різних класів застосовують чисельні методи інтерполювання, найменших квадратів, ортогоналізації, врівноваження значень, умовної мінімізації тощо.

Людям чия робота пов’язана з обчисленнями, часто доводиться вирішувати алгебраїчні та трансцендентні рівняння і системи рівнянь, що можуть бути самостійною задачею (наприклад, аналіз рівноваги сил в жорсткій системі балок або дослідження умов та параметрів рівноваги хімічної реакції тощо) або частиною більш складних задач. В обох випадках практична цінність чисельного методу значною мірою визначається швидкістю та ефективністю отримання розв‘язку. Розглянемо найбільш відомі чисельні методи та ефективні алгоритми розв‘язування систем лінійних алгебраїчних рівнянь.

Чисельне інтегрування та диференціювання починається із визначення відповідних операцій. Однак, з урахуванням необхідності економії обсягу обчислень та з урахуванням некоректності задачі диференціювання з’являється велика кількість чисельних методів для різних класів функцій та різного роду вихідних даних.

Інженери часто зіштовхуються з диференційними рівняннями та системами диференційних рівнянь при розробці нових виробів чи технологічних процесів, оскільки більша частина законів фізики формалізується саме у вигляді диференційних рівнянь. Будь-яка задача проєктування, яка пов’язана з розрахунком потоків енергії чи руху тіл, в кінцевому результаті зводиться до розв’язку диференційних рівнянь. На жаль, лише дуже малу частину з них можливо вирішити без використання обчислювальних машин. Тому чисельні методи розв’язку диференційних рівнянь відіграють важливу роль у практиці інженерних розрахунків.

Основою чисельних методів розв’язування багатьох класів рівнянь є дискретизація задачі з наступним приведенням отриманих нелінійних рівнянь до послідовності систем алгебраїчних рівнянь. У зв’язку з цим чисельні методи можна поділити за способом дискретизації на проєкційний, скінченнорізницеві та проекційно-різницеві, а за способом розв’язування лінійної системи – на прямі методи, ітераційні методи та комбіновані.

Розв’язування різних класів рівнянь та багатьох інших задач зводиться до задач мінімізації функцій та функціоналів за наявності або відсутності обмежень. Чисельні методи розв’язування задач мінімізації випливають із методів швидкого спуску по поверхні (мінімізація функції мети). Наприклад, методи швидкого спуску, градієнтного, загального градієнтного та найшвидшого спуску, методу можливих та спряжених напрямів тощо. Чисельні методи використовують в обчислювальній математиці для розв’язування відповідного типу задач.

Обчислювальна математика це математика, що містить питання, які пов’язані з обчисленнями та використанням комп’ютерів. ОМ – теорія чисельних методів розв’язування математичних задач. Задачі математики поділяють на:

• чисельне розв’язування інтегральних рівнянь;

• задачі апроксимації функцій;

• чисельне розв’язування систем диференціальних рівнянь;

• розв’язування лінійних рівнянь;

• знаходження власних значень та векторів матриці;

• знаходження сингулярних значень і векторів матриці;

• чисельне розв’язування нелінійних алгебраїчних рівнянь та їх систем;

• чисельне розв’язування систем нелінійних алгебраїчних рівняння;

• задачі екстраполяції;

• задачі оптимізації;

• обернені задачі.

• задачі інтерполяції функцій;

• чисельне інтегрування та обчислення похідної;

Чисельні методи використовувалися ще за часів Ньютона (1642-1727 рр.) для розв’язування задач з астрономії, геодезії та обчислення механічних конструкцій. На той час обчислення з використанням чисельних методів виконувалися з доволі високою точністю (до восьми знаків після коми). Наприклад, французький математик і астроном Урбен Левер’є (1811-1878 рр.), уточнюючи траєкторію руху планети Уран, виявив відхилення від обчисленої траєкторії. Він припустив, що ці відхилення спричиняє інша планета, яка до того не спостерігалась астрономами. Використовуючи чисельні методи, він за пів року обчислив масу та орбіту невідомої планети, що справляє дію на Уран і виводить планету із рівноваги. Один примірник своїх розрахунків Левер’є відразу ж послав Йогану Галле з Берлінської обсерваторії, який отримавши лист 23 вересня 1846 року, негайно почав спостереження і в ту ж ніч, дуже близько від місця вказаного Левер’є, знайшов невідому планету, яку пізніше назвали Нептуном.

Статистичне опрацювання експериментальних даних зазвичай ґрунтується на граничних теоремах теорії ймовірностей та вимагає обчислення оцінок в порівнянні з простими формулами. Однак, для підвищення якості оцінок необхідна велика кількість даних, і обсяг обчислень може виявитися дуже великим. Тому чисельні методи тут націлені на скорочення обсягу обчислень при збереженні якості результатів. Найефективнішими чисельними методами в цій галузі є методи, які застосовують швидке перетворення Фур’є.

Для розв’язування задач апроксимації та обчислення функцій різних класів застосовують чисельні методи інтерполювання, найменших квадратів, ортогоналізації, врівноваження значень, умовної мінімізації тощо. Найактуальнішими є методи кусково-многочленної та раціональної сплайн-апроксимації, а також адаптивної апроксимації та нелінійної за параметром апроксимації.

Чисельне інтегрування та диференціювання здійснюється на основі означення відповідних операцій. Однак, через необхідність економії обсягу обчислень та некоректність задачі диференціювання розроблено велику кількість чисельних методів для різних класів функцій та різного роду вихідних даних.

Основою чисельних методів розв’язування багатьох класів рівнянь є дискретизація задачі з наступним зведенням отриманих, взагалі кажучи, нелінійних рівнянь до послідовності систем алгебраїчних рівнянь. У зв’язку з цим, чисельні методи можна поділити за способом дискретизації на проекційні, скінченно-різницеві та проекційно-різницеві, а за способом розв’язування лінійної системи — на прямі, ітераційні та комбіновані методи.

# 2.2 Розробка структури інформаційної системи

Структура програмного додатка повинна бути архітектурно розбита на модулі за завданням та з вимог архітектурних патернів для побудови програмних додатків. В програмі передбачені наступні бібліотеки:

- MonteCarloMethod.Main - модуль який містить точку входу в програму Main() з якої розпочинається роботи програми.

- MonteCarloMethod.Services - бібліотека, яка використовує сутності з бібліотеки MonteCarloMethod.Common та являє собою набір методів для обчислення фігур.

- MonteCarloMethod.Common - бібліотека для зберігання та опису даних у вигляді класів, які представляються фігури.

Дерево викликів представляє з себе ілюстрацію у вигляді дерева, де кожна вершина представлена у вигляді назви метода, а ребра являють собою виклик того чи іншого метода послідовно від коренів графа до вершини. Дерево викликів розміщено у додатку А.

Функціональність модуль обумовлена їх важливістю з точки зору архітектури програмного забезпечення.

Важливість бібліотека MonteCarloMethod.Common визначена її функціоналом, тобто, дана бібліотека складається виключно з класів для опису моделей даних. Даний модуль є найнижчим в ієрархії програмного додатка, від якого залежать всі інші, оскільки працюють на основі моделей, які описані в цій бібліотеці.

Бібліотека MonteCarloMethod.Service надає можливості для обчислення деяких значень, тобто, вона описує поведінку класів з іншої бібліотеки MonteCarloMethod.Common. Дана бібліотека вважається найбільш важливою з усієї серверної частини, оскільки саме тут проводяться всі обрахунки площ, кутів, сторін тощо. Дана бібліотека використовується головним модулем MonteCarloMethod.Main для обчислення відповідних значень, які задає користувач.

Центральний проєкт MonteCarloMethod.Main використовується для запуска програми. В даному модулі присутні GUI елементи які знаходяться на вікні MainWindow.xaml, саме вміст цього файлу відображається користувачу при запуску програми.

При взаємодії з елементами управління, користувач викликає деяку послідовність програмних дій. З MainWindow.cs викликається відповідний об'єкт сервісного класу з бібліотеки MonteCarloMethod.Services, який обраховує значення відповідних об'єктів бібліотеки MonteCarloMethod.Common та повертає або значення, які запитуються з форми або суцільні об'єкти відповідних класів.

# 2.3 Вибір засобів розробки інформаційної системи

Для розв'язання задач подібного роду з графічним інтерфейсом є досить багато окремих технологій, які дають можливість та певний набір інструментів для розробки рішень з метод певних математичних або частково математичних задач з графічним користувацьким інтерфейсом GUI (Graphic User Interfaca).

GUI (англ. - Graphic User Interface) - технологія, яка дозволяє взаємодіяти з системою програмного додатка через периферійні пристрої, такі як, для прикладу, комп'ютерна миш, клавіатура тощо. Головною відмінністю GUI від, вже застарілих, консольних методів взаємодії полягає в можливості поділення графічного інтерфейсу на горизонтальні та вертикальні секції, в яких можна розміщувати елементи управління. Також GUI надає можливість для постійної роботи програми, що стало великим проривом після закінчення ери консольних додатків. GUI дає можливість користувачу самому вирішувати в якій послідовності використовувати компоненти графічного інтерфейсу та налаштовувати умови виконання програми в момент її виконання.

На сьогодні є досить багато технологій та платформ, які надають можливість роботи з графічним інтерфейсом та відрізняються лише за критеріями платформи для використання та гнучкість можливостей для налаштування та зображення, але це вже визначається дизайнером, а не програміста-розробника.

Згідно з завданням, ми відштовхуємося від платформ для настільних додатків, тому наш вибір полягає лише в виборі однієї із трьох платформ:

– Windows Forms C/C++ (+ CLR)

– Windows Forms C#

– WPF

Платформа Findows Forms C/C++ не підходить по причині низькорівневих рішень та застарілої архітектури. Також, у порівнянні з "Братською" технологією Windows Forms C#, остання надає значно більше можливостей для розробки графічного інтерфейсу та гнучке налаштування за допомогою засобів мови C#. Також сам вибір мови для рішення C# дає багато переваг для спрощення розробки. Дані переваги можна знайти у відкритих електронних ресурсах, тому в рівницю між ними ми не будемо відокремлювати.

WPF - сучасна платформа для розробки настільних додатків засобами мови C# та з використанням мови графічної розмітки для настільних додатків XAML. Тільки сама наявність XAML надає досить багато переваг, оскільки дозволяє відокремити розробки графічної частини від програмної/серверної. Також технологія має досить великий перелік зручних елементів управління та написана з повним дотриманням архітектурних патернів для розробки. Усі елементи розділені на сімейства класів, чого немає у платформ типу Windows Forms. Також, однією із головних відмінностей між платформами є засіб для рендеринга інтерфейсу. Графічно WPF використовую відеокарту для рендеринга зображень та елементів управління, у той час я Windows Forms використовую процесор.

# 3. Програмна реалізація

# 3.1. Структура і функціональне призначення модулів системи, їх взаємозв’язок

Структура програмного додатка містить деякий набір програмних модулів для збереження та обробки даних. Програмний додаток складається з 3-х окремих модулів, 2 з яких є бібліотеками динамічної компоновки (.dll) та використовуються в основному проєкті (.Main).

Ієрархія програмних модулів являє собою зв'язок через залежності проєктів та посилання на ці проєкти. Центральним або головним проєктом являється проєкт MonteCarloMethod.Common (Далі .Common) який використовує функціонал інших підключених бібліотек. За замовчуванням даний модуль проєкту має елемент Вікно (WPF). Ієрархія проєкту зображена на рисунку 3.1.

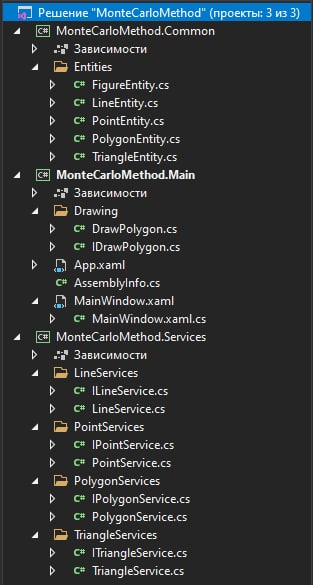


Рисунок 3.1 – Ієрархія проєкту

Розглянемо всю ієрархію додатку та визначимо значення та роль кожного із модулів. Діаграма взаємозв'язків зображена на рисунку 3.2.

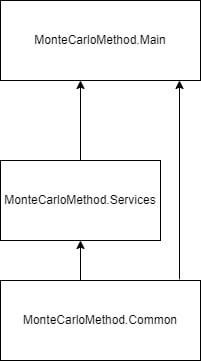


Рисунок 3.2 – Залежності програмного додатку

Найнижчим модулем в ієрархії є бібліотека .Common. Дана бібліотека використовує класи для опису моделей даних таких класів як: PointEntity, LineEntity, PolygonEntity (Багатогранник), FigureEntity тощо. Кожна з сутностей будується використовуючи зовнішні дані про точки, які встановлюють сторони фігури, які, своєю чергою, обмежують площу фігури віртуально; або набір координат для встановлення локації сутності.

Розглядаючи даний модуль, можна відокремити сутності PointEntity та LineEntity, які собою представляються опис моделі даних точки та відрізку (Лінії) відповідно. Центральним об'єктом бібліотеки являється клас PointEntity який будується комбінацією координат по прямій OX та OY відповідно, та будують об'єкт класу PointEntity віртуально. Клас LineEntity використовує для ініціалізації об'єкту комбінацію точок які можуть визначити саму пряму як віртуально, так і на площині. Також клас LineEntity має значення довжини відрізку, яке обчислюється та встановлюється в конструкторі, інакше користувач повинен власноруч обчислити довжину відрізку та встановити його через властивість (Propertie) класу.

Досить складним класом являється клас TriangleEntity який віртуально зображає трикутник та має такі властивості як: координати вершин трикутника, сторони трикутника (Об'єкти класу LineEntity) та кути трикутника. Даний клас має два конструктори, кожен з яких приймає по три параметри для ініціалізації об'єкту. Перший конструктор використовує комбінацію із трьох точок для встановлення вершин віртуального трикутника. Далі, за допомогою цих точок знаходяться сторони трикутника та ініціалізуються всі сторони. Після цього знаходяться кути трикутника за допомогою формули косинусів, з якої ми "Витягуємо" значення кутів трикутника та ініціалізуємо відповідні властивості. Розміщення класів фігур в ієрархії додатку зображено на рисунку 3.3

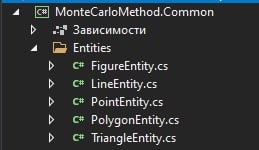


Рисунок 3.3 – Класи фігур

Модель даних PolygonEntity просто використовує список точок для встановлення вершин фігури та не має в собі властивостей сторін та кутів, а визначається виключно набором вершин які в майбутньому передаються з використанням сервісів та відсутні у вигляді параметрів конструктора.

Модуль MonteCarloMethod.Services виконує функцію посередника між класами опису моделей даних та верхнім шаром проєкту у вигляді наборів класів та інтерфейсів які в сукупності виконуються ролі сервісів. Описані в програмному додатку сервіси використовуються об'єкті сутностей для визначення найчастіше використовуваного функціонала та для звернення та повернення даних об'єктів сутності сервісу, як було сказано раніше, виконує роль посередника, а також зберігаючи часткову інкапсуляцію. Також використання сервісів дозволяє розділити логіку проєкт на класи для опису моделі та зберігання даних та іншої інформації.

Бібліотека .Services має 4 сервіси для обробки даних які мають відповідні сутності в бібліотеці .Common. Потрібно також наголосити, що бібліотека .Services напряму залежить від бібліотеки .Common та не може побудувати ні один свій об'єкт без використання об'єктів бібліотеки .Common

Сервіс PointServoice використовуються лише для визначення довжини від початку координат до самої точки. Даний сервіс не використовується майже ніде в рішені, але його описання та проєктування передбачено деякими архітектурними патернами.

Сервіс для сутності LineEntity дозволяє отримати відповідні точки А та B з яких будується відрізок. Також сервіс дозволяє отримати об'єкти класу PointEntity який є координатою середини даного відрізку.

Сервіс TriangleService є досить легким в порівнянні із відповідною йому сутністю TriangleEntity. Даний сервіс також використовую посередницькі методи для отримання та повернення даних відповідної моделі, серед яких методи, які повертають значення відповідних кутів, об'єкти класів що описуються сторони та вершини трикутника. Також даний сервіс дозволяє обчислити значення периметру для площі трикутника.

Найскладнішим сервісом являється PolygonService. Сервіс має відповідним метод для додавання вершини до полігону, задаючи його нову сторону. Розглянемо всі методи сервіс по черзі.

— Метод FindArea() повертає значення площі довільного полігону використовуючи координати вершин. Для знаходження площі були використано формулу для знаходження площі довільного багатокутника. На рисунках 3.4 на 3.5 показаний порядок множення значень координат вершин полігону.

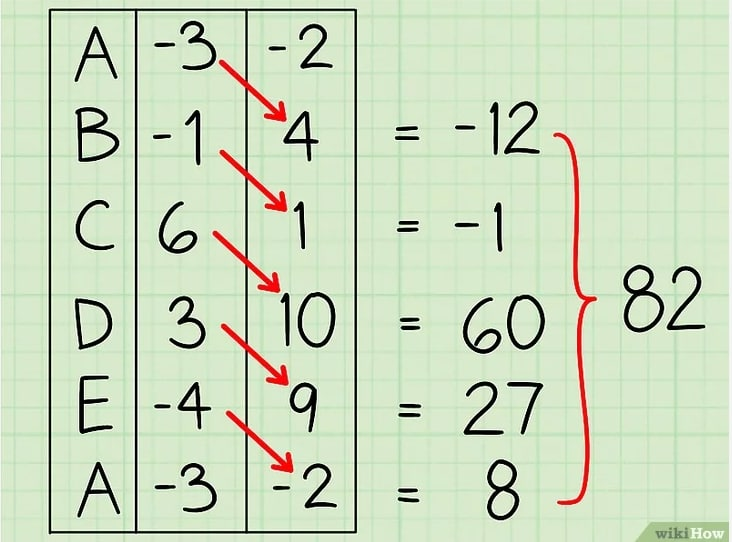


Рисунок 3.4 – Перший крок множення

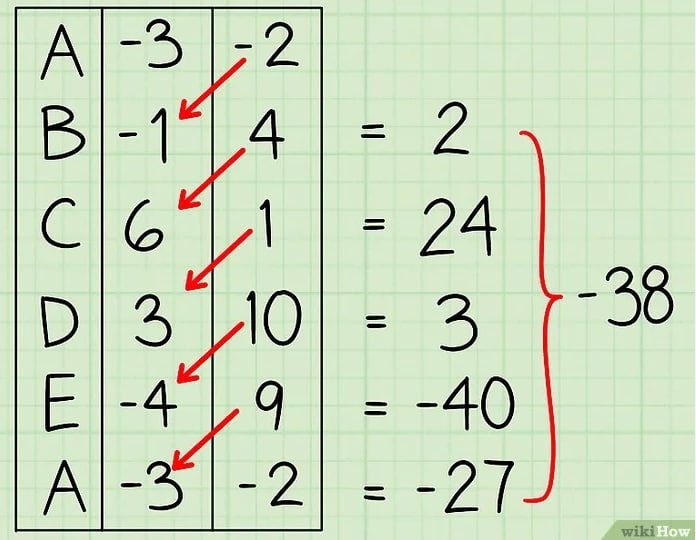


Рисунок 3.5 – Другий крок множення

Далі значення, які було визначено в кроках, що зображені на рисунках 3.4 та 3.5 потрібно відняти так та поділити у відповідному порядку. Процес зображений на відповідних рисунках 3.6 та 3.7.

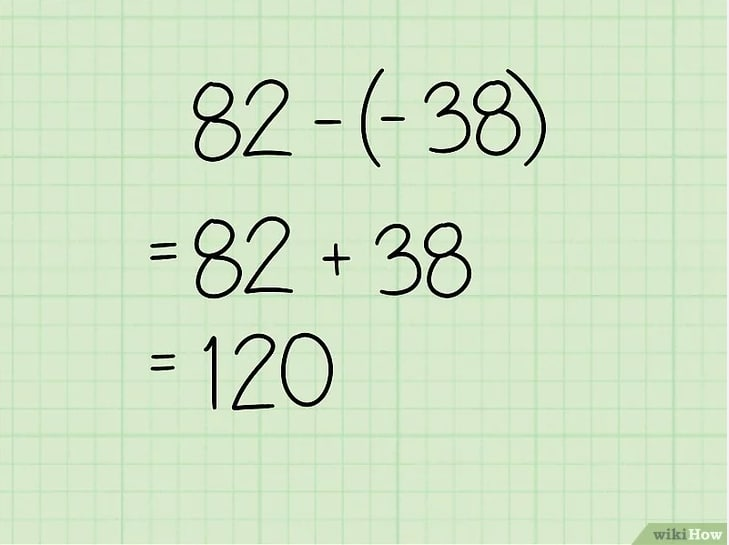


Рисунок 3.6 – Віднімання відповідних результатів

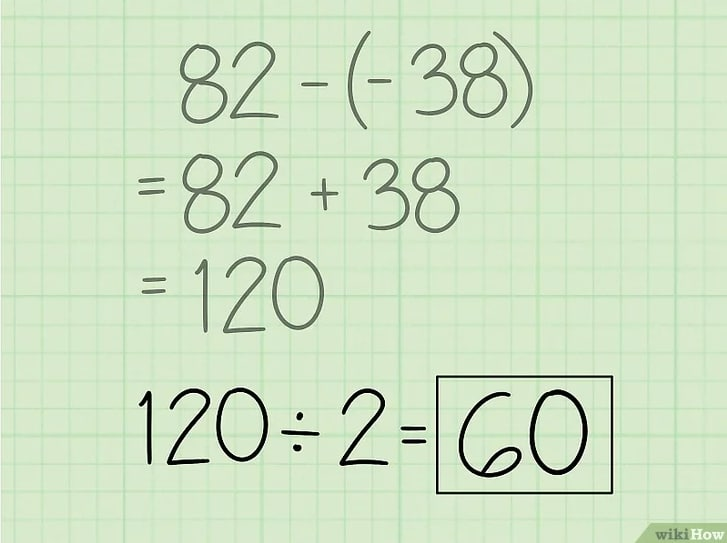


Рисунок 3.7 – Знаходження площі

— Метод IsInside(PointEntity point) повертає відповідне значення, чи знаходиться вказана точка всередині полігону чи ні. Даний метод було реалізовано з використанням "Методу трикутників". Починаючи з деякої вершини полігону будується трикутник, середнім кутом якого є вказана точка, а третьою вершиною - сусідня точка полігону відносно початкової. Після цього трикутник передається в сервіс звідки повертається значення відповідного центрального кута. Далі це повторяється по для всіх точок по черзі та сумуються значення всіх центральних кутів після чого зрівняється із фактичним значенням в 360 градусів. Значення 360 градусів повинне вийти за умови, якщо точка знаходиться всередині полігону, в такому випадку повертається відповідне значення true. Якщо значення відрізняються від 360 градусів із похибкою, тоді повертається значення false.

На рисунках 3.8 та 3.9 показана візуалізація точки, що знаходиться в середині полігону та поза нею методом трикутників.

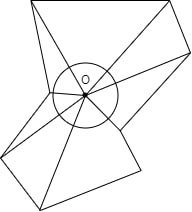


Рисунок 3.8 – Точка, що знаходиться всередині полігона

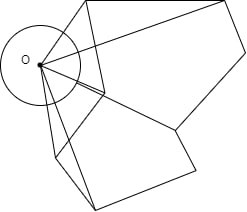


Рисунок 3.9 – Точка, яка знаходиться за межами полігона

— Метод MonteCarlo(List<PointEntity> points, double canvasArea) для знаходження площі полігону методом Монте-Карло. Параметрами ми передаємо в метод список точок, які ми використовуємо в ході дослідження відповідним методом; а також передається значення площі полотна, на якому ми будемо проводитися дослідження. Метод повертає площу полігону та працює за звичайним алгоритмом методу Монте-Карло, який показано на рисунку 3.10.



Рисунок 3.10 – Алгоритм методу Монте-Карло

— Метод TriangleMethod(PointEntity insidePoint) визначає площу полігону використовуючи лише точку, яка обов'язково повинна знаходитися всередині полігону. Даний метод також працює на основі "Методу трикутників" та використовую суму площ всі трикутників, що вдалось побудувати використавши вершини полігону та центральну точку, яка лежить в середині самого полігону. Робота метода майже аналогічна до методу IsInside(PointEntity point), але сумує не кути побудованих трикутників, а їх площі. Так цей метод є найбільш точним та найбільш надійним та використовуються як значення реальної площі полігону для порівняння із роботою метода Монте-Карло

Головним модулем в програмному додатку є модуль .Main який фактично є користувацькою оболонкою для використання всіх бібліотек, від яких залежить цей модуль. Для користувача доступне лише вікно (MainWindows.xaml) за допомогою якого користувач може здійснювати керування іншими програмними модулями. Зображення форми MainWindows.xaml показано на рисунку 3.11.

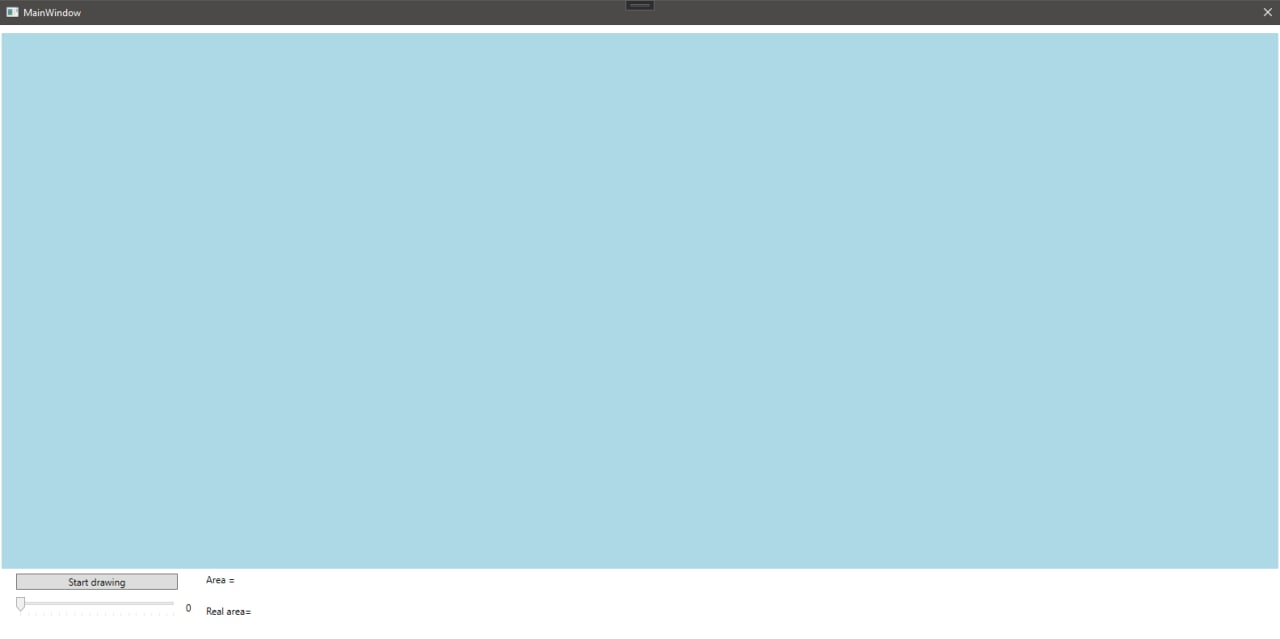


Рисунок 3.11 – Форма MainWindows.xaml

Також, як було написано вище, має вбудований аналог сервісу DrawPolygon за допомогою методів якого зручно малювати на полотні, а також це зв'язано з проблемою передачі об'єктів класів елементів керування в інші проєкти.

# 3.2 Розробка програмних модулів

Виходячи з вище написаного, програмний додаток має чітку ієрархію та складається з модулів (.dll бібліотек) для збереження та обробки даних. Модуль .Common використовується для опису моделі даних та збереження даних. Даний модуль описує наступні моделі даних: Фігура, Лінія, Точка, Полігон та Трикутник. Загальну ієрархію розміщення класів опису моделей фігур було зображено в пункті 3.1 на рисунку 3.3.

Клас опису моделі фігури FigureEntity структуру, яку зображено на рисунку 3.12.

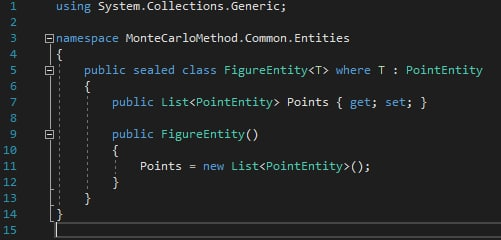


Рисунок 3.12 – Клас FigureEntity

Даний клас описує модель фігури та складається з колекції List<>, яка описує набір точок, необхідних для побудови фігури. Даний клас наразі не використовується в програмі, а лише описує модель даних, доданий з метою майбутньої підтримки програмного додатка та майбутніх реалізацій.

Модель LineEntity описує лінію, яка складається з двох точок. Опис даних, який надає даний клас зображено на рисунку 3.13. Даний клас описує сполучення двох точок на віртуальній площині та визначає довжину між ними та використовується для візуального відображення сторін фігур, які визначені в програмному додатку. Структуру опису даного класу можна побачити на рисунку 3.13.

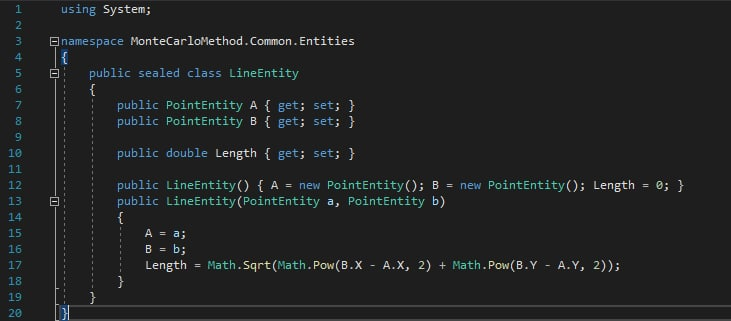


Рисунок 3.13 – Клас LineEntity

Клас PointEntity відіграє роль найменшої одиниці для побудови фігур та використовується майже всюди. Опис точки можна побачити на рисунку 3.14.

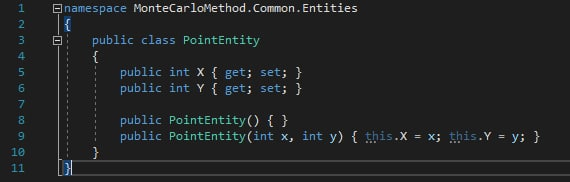


Рисунок 3.14 – Клас PointEntity

Багатокутна фігура, тобто, полігон, описана класом PolygonEntity. Полігон будується виключно зі списку деякої довжини точок (PointEntity), які визначають вершини полігону. Структура класу зображена на рисунку 3.15.

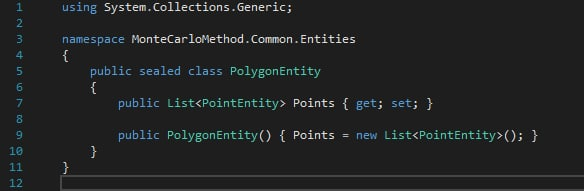


Рисунок 3.15 – Клас PolygonEntity

Класи, що описані вище відіграють роль опису деяких сутностей без функціонала. За деяким архітектурним рішенням, весь функціонал винесено в сервісні класи іншої бібліотеки .Services. Дана бібліотека містить класи з відповідними назвами моделей, а також відповідні інтерфейси, які оголошують функціонал який потім реалізується в відповідних класах згідно з поліморфізмом.

Сервісні класи для сутностей PointEntity та LineEntity описуються мінімальний функціонал, оскільки ці сутності є будівельними блоками для інших сутностей. Клас PointSrvice показано на рисунку 3.16.

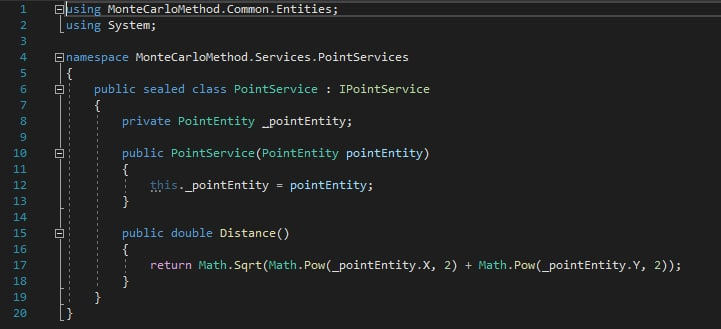


Рисунок 3.16 – Клас PointService

Сервіс PointService надає мінімальний функціонал та реалізує лише одним метод, який повертає відстань від точки до початку координат віртуального полотна. Сервіс LineEntity надає стандартні можливості для відображення поведінки об'єкту LineEntity та може визначити такі значення: значення середньої точки лінії, повертає точки початку та кінця лінії.

Сервісний клас TriangleService відіграє дуже важливу роль в програмі, оскільки за допомогою цього сервісу знаходяться різні значення, такі як площа полігону, визначення, чи належить точка полігону чи ні, тощо. Сам сервіс має структуру, що зображено на рисунку 3.17.

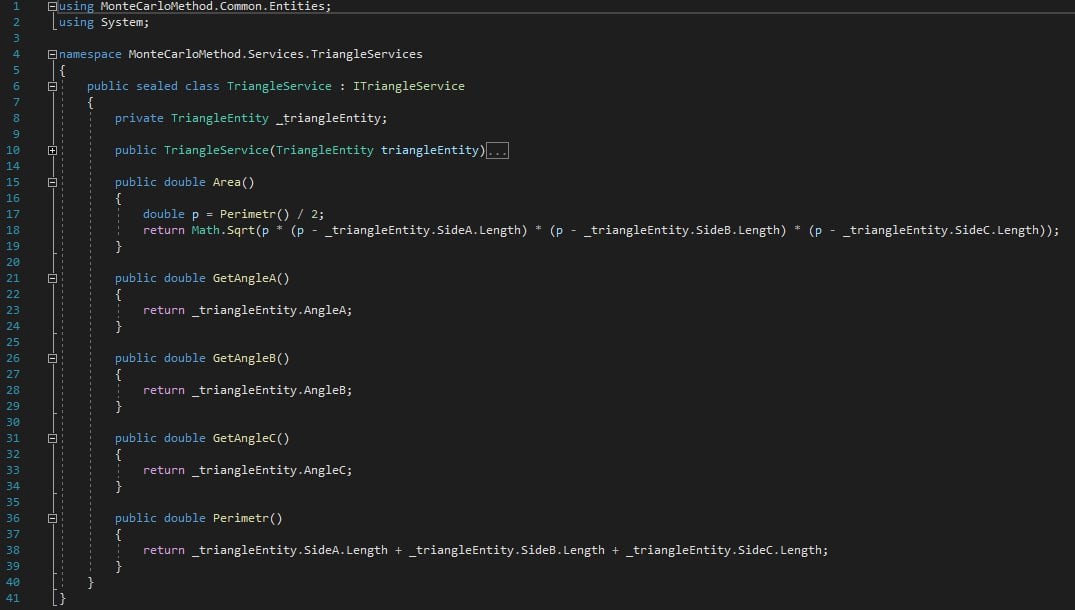


Рисунок 3.17 – Клас TriangleService

Розглянемо методи та розберемо їх.

Метод Area() використовую метод Perimetr() для знаходження площі трикутника. Метод Perimetr() використовує сторони трикутника для знаходження периметру трикутника, який потім викликається в методі Area() для знаходження площі трикутника. Метод Area() використовує формулу Герона для знаходження площі трикутника з використанням півпериметра та повертає значення площі з плаваючою крапкою.

Також сервіс дає можливість отримати значення кутів трикутника і відіграє, як описувалось раніше, роль посередника між класом моделі та класом функціоналу, тобто даним сервісом. За бажанням або за потреби, в майбутньому, можливо також доповнити даний сервіс іншими методами для повертання інших даних з сутності або розрахунку інших значень.

Клас PolygonService являється центральним в програмному додатку, оскільки фігура цього класу будується в ході користувацької взаємодії з графічним інтерфейсом програми, та виконує різного роду обчислення для проведених користувацьких досліджень. Детальніше структуру сервісу можна побачити на рисунках 3.18, 3.19, 3.20.

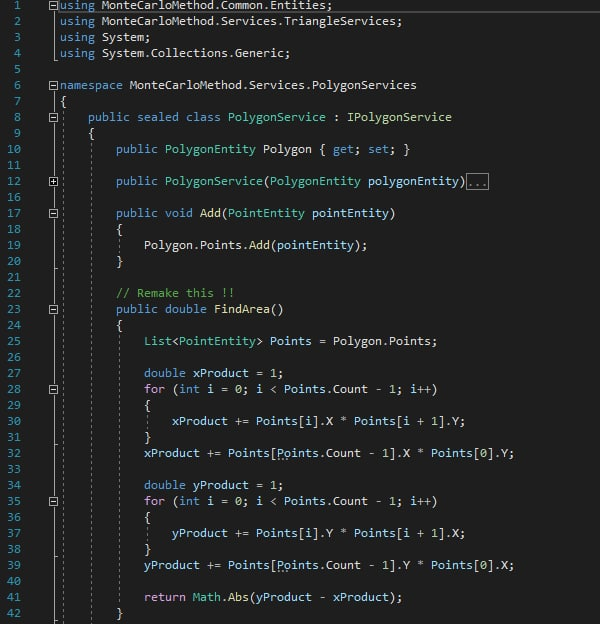
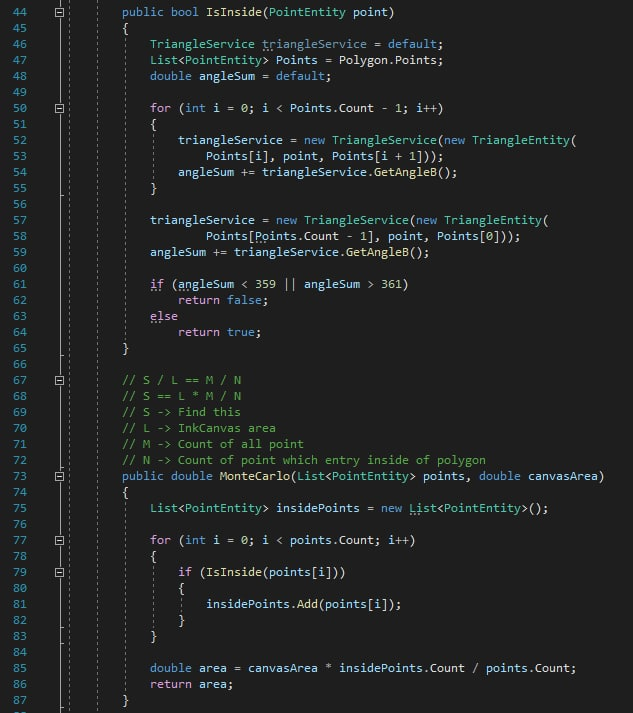
 

Рисунок 3.18 – Клас PolygonService Рисунок 3.19 – Клас PolygonService

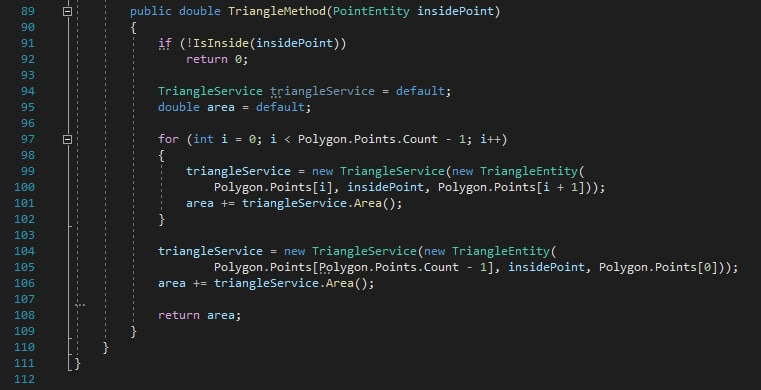


Рисунок 3.20 – Клас PolygonService

Розглянемо та розберемо усі методи та логіку їх роботи.

Метод Add(PointEntity point) використовується для додавання нової вершини до полігону. Метод виступає посередником між сервісом та сутністю PolygonEntity та звертається напряму до списку точок в середині сутності.

Метод FindArea() використовується для знаходження площі та повертає її значення з плаваючою точкою. В тілі метода реалізований процес знаходження площі методом множення відповідних координат. Детальніше процес був описаний в пункті 3.1 та зображено на рисунках 3.4 - 3.7. Даний метод не використовується в програмному додатку та є "Резервним" методом, який в майбутньому може використовуватися в якості контрольного значення або для інших цілей.

Метод MonteCarlo(List<PointEntity> points, double canvasArea) приймає на вхід два параметри іменованими points та canvasArea. Параметр points - це список усіх точок, які були згенеровані в ході дослідження методом Монте-Карло. Другий параметр canvasArea - це значення площі полотна, на якому були згенеровані точки зі списку points.

Метод використовую формулу алгоритму Монте-Карло p/P=n/N , де:

* p – площа полігону, яку ми знаходимо;
* P – площа полотна або більшої фігури, в яку повністю вписаний полігон з площею p;
* n – кількість точок, які належать полігону;
* N – загальна кількість точок, які були згенеровані в процесі дослідження на площині P.

Метод TriangleMethod(PointEntity insidePoint) використовується для вичислення площі полігону методом сум площ трикутників та використовується як фактична площа полігону для порівняння з методом Монте-Карло.

Суть роботи цього методу полягає в тому, що в методі передається точка, яка належить полігону. Далі будується деяка кількість трикутників, середнім кутом яких є ця точка в середині полігону, а далі, через сервіс TriangleService знаходиться площа кожного із трикутників та додаються. Сума цих площ повертається з методу як площа самого полігону з плаваючою точкою.

Центральним модулем програмного додатка є користувацький проєкт .Main. Даний модуль відіграє роль запускаємого проєкту для користувача та містить точку входу в програму Main(). Структура модуля .Main зображено на рисунку 3.21.

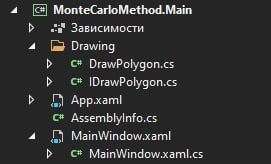


Рисунок 3.21 – Ієрархія .Main

Даний модуль має локальний сервіс DrawPolygon який буде описаний далі. Також тут присутнє вікно для відображення користувачу при проведенні дослідження. Також в даному модулі присутні такі файли як: App.xaml та AssemblyInfo.cs. Ці файли вважаються системними й розглядати ми їх не будемо через те, що вони не використовуються напряму ні користувачем, ні програмістом.

Локальний сервіс DrawPolygon призначений для автоматичного малювання полігону на "Задній частині" програми не навантажуючи основний файл MainWindow.xaml. Структуру сервісного класу DrawPolygon зображено на рисунку 3.22.

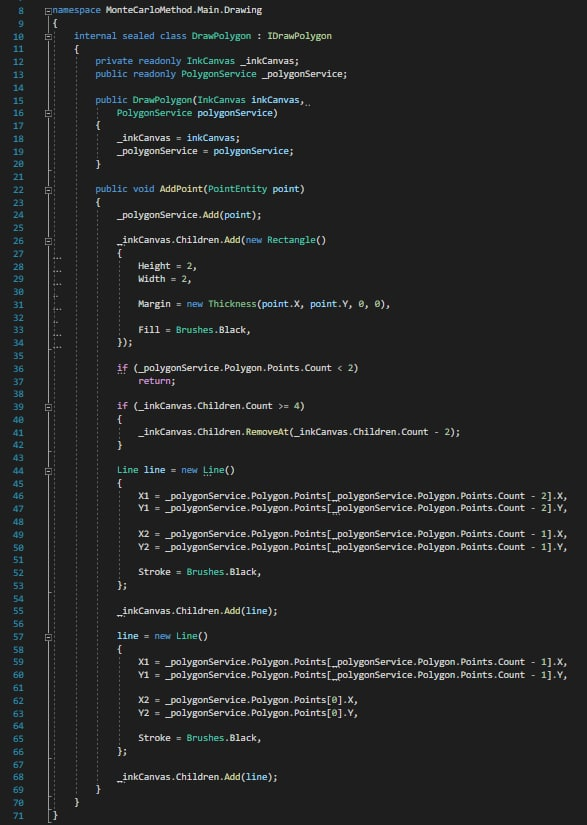


Рисунок 3.22 – Сервіс DrawPolygon

Сервіс PolygonService містить лише один метод AddPoint(PointEntity point), де вхідний параметр point - це нова точка, яка буде додана на полотно та буде відігравати роль нової вершини полігону. Робота методу полягає в тому, що ми додаємо нову точку на полотно у вигляді прямокутника, оскільки клас полотна InkCanvas не підтримує малювання точки, а виключно фігури, які є нащадками класу Shape, ким і являється об'єкт класу Rectange. Після цього на полотні проводиться лінія від заданої нової точки до попередньої точки для сполучення вершин полігону, а також лінія до самої першої точки, яку поставив користувач на полотні. Це використовується для відображення візуальної цілісності фігури на полотні. При наступних викликах цього методу, будуть видалятися попередньо додані сторони, які сполучали попередню точку із самою першою.

Користувацьке вікно має відповідний клас MainWindows.cs, де знаходяться всі обробники подій та їх скрипти. Загальна структура коду даного вікна більше детально описана у відповідному додатку.

# 3.3 Інструкція користувача

Загальні відомості:

Після відкриття додатку з'являється відповідне вікно статичних розмірів на якому знаходяться наступні елементи управління:

* Полотно для малювання;
* Кнопка активації / деактивації режиму малювання;
* Слайдер для задання точності;
* Поле для виведення площі фігури при обчисленні;
* Поле для виведення фактичної площі фігури.

Як користуватися:

Після успішного запуску програми слід натиснути кнопку активації режиму малювання, після чого слід клацнути на полотно для відображення точки та продовжувати далі, доки графічна фігура не буде рівня задуманій. Малювати слід лише в одному напрямі, тобто, за годинниковою стрілкою або проти, оскільки кожна нова точка буде автоматично сполучатися з попередньою та першою (Центральною).

Після завершення дії зображення фігури можна виставити бажане значення на елементі задання точності (Слайдері) або одразу натиснути на кнопку деактивації режиму малювання. Після цього на полотно буде виведено деяку кількість випадково згенерованих точок в залежності від виставленого значення на елементі задання точності або їх кількість за замовчуванням (1000). Після цього на відповідні поля буде виведена досліджувана площа та фактична площа.

Далі в цьому режимі можна змінювати положення Слайдера для зміни значення точності дослідження. При будь-якій змінні положення Слайдера на полотні буде генеруватися відповідна кількість точок для дослідження площі та виводитися її досліджуване значення у відповідне поле.

Для виходу з даного режиму потрібно знову натиснути кнопку для активації режиму малювання, після чого все полотно буде очищене, а відповідно значення - очищені з пам'яті. Фактично програма перейде в початковий стан та буде готова до використання знову.

Важливо: програмний додаток працює лише в користувацькому режимі та після закриття не переходить в фоновий режим. При використанні програмного додатка користувач повністю усвідомлює, що програма знаходиться в режимі альфа-тесту та розуміє, що в програмному додатку можлива наявність "Багів", які можуть призвести до несправності програмного додатка.

# 3.4 Вимоги до технічних засобів

Використовуючи фігуру, що складається з 470 точок та використавши значення точності дослідження 100 одиниці програмний додаток використовує наступні ресурси:

– Процесор: до 50% при максимальній складності обчислення що включаються в себе видалення 100 000 - k \* 2 точок з полотна, де k - кількість точок, що описують полігон.

* ОЗУ: 600 Мб при трьох послідовних спробах обчислення площі.

Для однієї спроби з максимальними параметрами дослідження (470 вершин полігону та значення в 100 одиниць точності дослідження) завантаженість системи:

– Процесор: до 30% максимального значення. Середнє значення 25%

– ОЗУ: 256 МБ

Дослідження проводилися з використання наступних системних характеристик:

– Процесор: Intel Core I5-4460

– ОЗУ: 8 Гб

Рекомендовані системні характеристики пристрою:

– Процесор: Intel Pentium G4560

– ОЗУ: 4 Гб

ВИСНОВОК

Підведемо підсумки нашого курсового проєкту на тему Метод Монте-Карло для обчислення площ плоских фігур. Метою проєктування було створення програмного додатка, який автоматизує обчислення площ методом Монте-Карло та зможе повертати точні значення площ. Перед нами стояла задача побудувати якісний додаток з дружнім інтерфейсом для користувача та простота обчислень. Як висновок можна зазначити, що завдання було виконанне, але програмний додаток потребує подальшої модернізації та оптимізації.

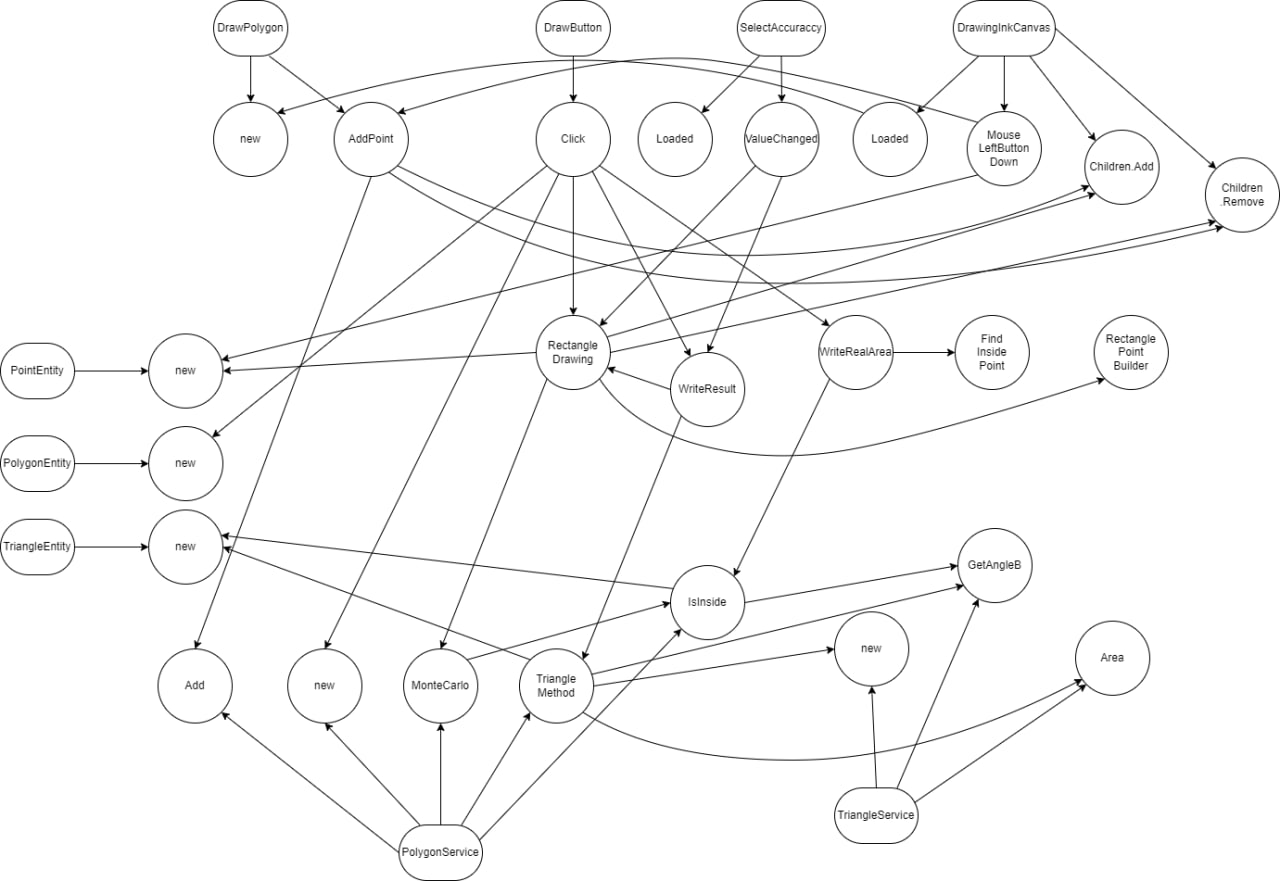
Проєкт був побудований на досить потужній платформі з відповідними технологіями, тому в майбутньому перехід на іншу платформу для настільних додатків не планується. З технічної точки зору, програмний продукт повністю готовий до використання та може буди корисним при розв'язанні конкретних задач.

Надалі програмний продукт можна вдосконалити шляхом перенесення на веб-платформу asp.net core та використовувати як браузерний продукт. Також, програмний продукт потребую доповлення функціоналу, оскільки він був розроблений строго під задачу курсового проєкту та не надає більше можливостей, чим цього потребується. Програмний продукт також можна доповнити вже визначеними фігурами, додати можливість переміщення об'єктів та накладання, виділення обчислюваної площі, яка належить декільком фігурам, також додати інші методи для дослідження та дослідити швидкодію обчислення площ різними методами, тобто вийти набагато далі за рамки задачі курсового проєкту.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

# ДОДАТОК А

Дерево викликів



# Додаток Б

(довідковий)

Приклади коду програмного додатку

Б.1 MonteCarloMethod.Common

Б.1.1 Приклад коду FigureEntity.cs

using System.Collections.Generic;

namespace MonteCarloMethod.Common.Entities

{

public sealed class FigureEntity<T> where T : PointEntity

{

public List<PointEntity> Points { get; set; }

public FigureEntity()

{

Points = new List<PointEntity>();

}

}

}

Б.1.2 Приклад коду LineEntity.cs

using System;

namespace MonteCarloMethod.Common.Entities

{

public sealed class LineEntity

{

public PointEntity A { get; set; }

public PointEntity B { get; set; }

public double Length { get; set; }

public LineEntity() { A = new PointEntity(); B = new PointEntity(); Length = 0; }

public LineEntity(PointEntity a, PointEntity b)

{

A = a;

B = b;

Length = Math.Sqrt(Math.Pow(B.X - A.X, 2) + Math.Pow(B.Y - A.Y, 2));

}

}

}

Б.1.3 Приклад коду PointEntity.cs

namespace MonteCarloMethod.Common.Entities

{

public class PointEntity

{

public int X { get; set; }

public int Y { get; set; }

public PointEntity() { }

public PointEntity(int x, int y) { this.X = x; this.Y = y; }

}

}

Б.1.4 Приклад коду PolygonEntity.cs

using System.Collections.Generic;

namespace MonteCarloMethod.Common.Entities

{

public sealed class PolygonEntity

{

public List<PointEntity> Points { get; set; }

public PolygonEntity() { Points = new List<PointEntity>(); }

}

}

Б.1.5 Приклад коду TriangleEntity

using System;

namespace MonteCarloMethod.Common.Entities

{

public sealed class TriangleEntity

{

/\*

A

/\

/ \

c / \ b

/ \

/\_\_\_\_\_\_\_\_\

B a C

\*/

public PointEntity A { get; set; }

public PointEntity B { get; set; }

public PointEntity C { get; set; }

public LineEntity SideA { get; set; } // B -- C

public LineEntity SideB { get; set; } // A -- C

public LineEntity SideC { get; set; } // A -- B

public double AngleA { get; set; }

public double AngleB { get; set; }

public double AngleC { get; set; }

public TriangleEntity(PointEntity a, PointEntity b, PointEntity c)

{

A = a;

B = b;

C = c;

SideA = new LineEntity(B, C);

SideB = new LineEntity(A, C);

SideC = new LineEntity(A, B);

AngleA = Math.Acos((Math.Pow(SideB.Length, 2) + Math.Pow(SideC.Length, 2) - Math.Pow(SideA.Length, 2)) / (2 \* SideB.Length \* SideC.Length)) \* 180 / Math.PI;

AngleB = Math.Acos((Math.Pow(SideA.Length, 2) + Math.Pow(SideC.Length, 2) - Math.Pow(SideB.Length, 2)) / (2 \* SideA.Length \* SideC.Length)) \* 180 / Math.PI;

AngleC = Math.Acos((Math.Pow(SideA.Length, 2) + Math.Pow(SideB.Length, 2) - Math.Pow(SideC.Length, 2)) / (2 \* SideA.Length \* SideB.Length)) \* 180 / Math.PI;

}

public TriangleEntity(LineEntity a, LineEntity b, LineEntity c)

{

SideA = a;

SideB = b;

SideC = c;

A = SideB.A;

B = SideA.A;

C = SideA.B;

AngleA = Math.Acos((Math.Pow(SideB.Length, 2) + Math.Pow(SideC.Length, 2) - Math.Pow(SideA.Length, 2)) / (2 \* SideB.Length \* SideC.Length)) \* 180 / Math.PI;

AngleB = Math.Acos((Math.Pow(SideA.Length, 2) + Math.Pow(SideC.Length, 2) - Math.Pow(SideB.Length, 2)) / (2 \* SideA.Length \* SideC.Length)) \* 180 / Math.PI;

AngleC = Math.Acos((Math.Pow(SideA.Length, 2) + Math.Pow(SideB.Length, 2) - Math.Pow(SideC.Length, 2)) / (2 \* SideA.Length \* SideB.Length)) \* 180 / Math.PI;

}

}

}

Б.2 MonteCarloMethod.Main

Б.2.1 Приклад коду DrawingPolygon.cs

using MonteCarloMethod.Common.Entities;

using MonteCarloMethod.Services.PolygonServices;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Media;

using System.Windows.Shapes;

namespace MonteCarloMethod.Main.Drawing

{

internal sealed class DrawPolygon : IDrawPolygon

{

private readonly InkCanvas \_inkCanvas;

public readonly PolygonService \_polygonService;

public DrawPolygon(InkCanvas inkCanvas,

PolygonService polygonService)

{

\_inkCanvas = inkCanvas;

\_polygonService = polygonService;

}

public void AddPoint(PointEntity point)

{

\_polygonService.Add(point);

\_inkCanvas.Children.Add(new Rectangle()

{

Height = 2,

Width = 2,

Margin = new Thickness(point.X, point.Y, 0, 0),

Fill = Brushes.Black,

});

if (\_polygonService.Polygon.Points.Count < 2)

return;

if (\_inkCanvas.Children.Count >= 4)

{

\_inkCanvas.Children.RemoveAt(\_inkCanvas.Children.Count - 2);

}

Line line = new Line()

{

X1 = \_polygonService.Polygon.Points[\_polygonService.Polygon.Points.Count - 2].X,

Y1 = \_polygonService.Polygon.Points[\_polygonService.Polygon.Points.Count - 2].Y,

X2 = \_polygonService.Polygon.Points[\_polygonService.Polygon.Points.Count - 1].X,

Y2 = \_polygonService.Polygon.Points[\_polygonService.Polygon.Points.Count - 1].Y,

Stroke = Brushes.Black,

};

\_inkCanvas.Children.Add(line);

line = new Line()

{

X1 = \_polygonService.Polygon.Points[\_polygonService.Polygon.Points.Count - 1].X,

Y1 = \_polygonService.Polygon.Points[\_polygonService.Polygon.Points.Count - 1].Y,

X2 = \_polygonService.Polygon.Points[0].X,

Y2 = \_polygonService.Polygon.Points[0].Y,

Stroke = Brushes.Black,

};

\_inkCanvas.Children.Add(line);

}

}

}

Б.2.2 Приклад коду IDrawPolygon.cs

using MonteCarloMethod.Common.Entities;

using System.Windows.Controls;

namespace MonteCarloMethod.Main.Drawing

{

public interface IDrawPolygon

{

void AddPoint(PointEntity point);

}

}

Б.2.3 Приклад коду MainWindow.xaml.cs

using MonteCarloMethod.Common.Entities;

using MonteCarloMethod.Main.Drawing;

using MonteCarloMethod.Services.PolygonServices;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

namespace MonteCarloMethod.Main

{

public partial class MainWindow : Window

{

private DrawPolygon \_drawPolygon;

private PolygonService \_polygonService;

List<PointEntity> \_pointEntities;

private bool \_drawMode;

public MainWindow()

{

\_polygonService = new PolygonService(new PolygonEntity());

InitializeComponent();

}

private void SelectAccuracy\_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)

{

SelectAccuracy.Value = (int)SelectAccuracy.Value;

AccuracyValueOutputLabel.Content = SelectAccuracy.Value;

for (int i = \_polygonService.Polygon.Points.Count \* 2; i < DrawingInkCanvas.Children.Count; i++)

DrawingInkCanvas.Children.RemoveAt(i);

RectangleDrawing();

WriteResult();

}

private void SelectAccuracy\_Loaded(object sender, RoutedEventArgs e)

{

SelectAccuracy.ValueChanged += SelectAccuracy\_ValueChanged;

}

private void DrawButton\_Click(object sender, RoutedEventArgs e)

{

\_drawMode = !\_drawMode;

if (\_drawMode)

{

DrawingInkCanvas.Children.Clear();

\_polygonService = new PolygonService(new PolygonEntity());

\_drawPolygon = new DrawPolygon(DrawingInkCanvas, \_polygonService);

DrawButton.Content = "Stop drawing";

}

if (!\_drawMode)

{

DrawButton.Content = "Start drawing";

RectangleDrawing();

WriteResult();

WriteRealArea();

}

}

private void DrawingInkCanvas\_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)

{

if (!\_drawMode)

return;

MouseEventArgs mouseEvent = e as MouseEventArgs;

PointEntity point = new PointEntity()

{

X = (int)mouseEvent.GetPosition(this).X,

Y = (int)mouseEvent.GetPosition(this).Y,

};

\_drawPolygon.AddPoint(point);

}

private Rectangle RectanglePointBuilder(double startX, double startY)

{

Rectangle rectangle = new Rectangle()

{

Height = 2,

Width = 2,

Margin = new Thickness(startX, startY, 0, 0),

Fill = Brushes.Black,

};

return rectangle;

}

private void DrawingInkCanvas\_Loaded(object sender, RoutedEventArgs e)

{

\_drawPolygon = new DrawPolygon(DrawingInkCanvas, \_polygonService);

}

private double RectangleDrawing()

{

if (!\_drawMode)

{

DrawButton.Content = "Start drawing";

if (DrawingInkCanvas.Children.Count >= 3)

{

\_pointEntities = new List<PointEntity>();

for (int i = 0; i < SelectAccuracy.Value \* 100; i++)

{

\_pointEntities.Add(new PointEntity()

{

X = new Random().Next(0, (int)ImageContainer.Width),

Y = new Random().Next(0, (int)ImageContainer.Height),

});

DrawingInkCanvas.Children.Add(RectanglePointBuilder(\_pointEntities[i].X, \_pointEntities[i].Y));

}

return \_polygonService.MonteCarlo(\_pointEntities, (int)(ImageContainer.Width \* ImageContainer.Height));

}

}

return 0;

}

private void WriteResult()

{

string accuracyOutput = default;

foreach (char sym in AreaResultLabel.Content.ToString())

{

if (sym == '=')

break;

accuracyOutput += sym.ToString();

}

AreaResultLabel.Content = $"{accuracyOutput}= {RectangleDrawing()}";

}

private void WriteRealArea()

{

string accuracyOutput = default;

foreach (char sym in RealAreaLabel.Content.ToString())

{

if (sym == '=')

break;

accuracyOutput += sym.ToString();

}

RealAreaLabel.Content = $"{accuracyOutput}= {\_polygonService.TriangleMethod(FindInsidePoint())}";

}

private PointEntity FindInsidePoint()

{

foreach(PointEntity point in \_pointEntities)

{

if (\_polygonService.IsInside(point))

return point;

}

return null;

}

}

}

Б.3 MonteCarloMethod.Services

Б.3.1 Приклад коду ILineService.cs

using MonteCarloMethod.Common.Entities;

namespace MonteCarloMethod.Services.LineServices

{

public interface ILineService

{

PointEntity GetPointA();

PointEntity GetPointB();

PointEntity GetMiddlePoint();

}

}

Б.3.2 Приклад коду LineService.cs

using MonteCarloMethod.Common.Entities;

using System;

namespace MonteCarloMethod.Services.LineServices

{

public sealed class LineService : ILineService

{

private LineEntity \_lineEntity;

public LineService(LineEntity lineEntity)

{

\_lineEntity = lineEntity;

}

public PointEntity GetMiddlePoint()

{

return new PointEntity()

{

X = (\_lineEntity.A.X + \_lineEntity.B.X) / 2,

Y = (\_lineEntity.A.Y + \_lineEntity.B.Y) / 2

};

}

public PointEntity GetPointA()

{

return this.\_lineEntity.A;

}

public PointEntity GetPointB()

{

return this.\_lineEntity.B;

}

}

}

Б.3.3 Приклад коду IPointService.cs

namespace MonteCarloMethod.Services.PointServices

{

public interface IPointService

{

double Distance();

}

}

Б.3.4 Приклад коду PointService.cs

using MonteCarloMethod.Common.Entities;

using System;

namespace MonteCarloMethod.Services.PointServices

{

public sealed class PointService : IPointService

{

private PointEntity \_pointEntity;

public PointService(PointEntity pointEntity)

{

this.\_pointEntity = pointEntity;

}

public double Distance()

{

return Math.Sqrt(Math.Pow(\_pointEntity.X, 2) + Math.Pow(\_pointEntity.Y, 2));

}

}

}

Б.3.5 Приклад коду IPolygonService.cs

using MonteCarloMethod.Common.Entities;

using System.Collections.Generic;

namespace MonteCarloMethod.Services.PolygonServices

{

public interface IPolygonService

{

void Add(PointEntity pointEntity);

bool IsInside(PointEntity point);

double FindArea();

double TriangleMethod(PointEntity insidePoint);

double MonteCarlo(List<PointEntity> points, double canvasArea);

}

}

Б.3.6 Приклад коду PolygonSevice.cs

using MonteCarloMethod.Common.Entities;

using MonteCarloMethod.Services.TriangleServices;

using System;

using System.Collections.Generic;

namespace MonteCarloMethod.Services.PolygonServices

{

public sealed class PolygonService : IPolygonService

{

public PolygonEntity Polygon { get; set; }

public PolygonService(PolygonEntity polygonEntity)

{

Polygon = polygonEntity;

}

public void Add(PointEntity pointEntity)

{

Polygon.Points.Add(pointEntity);

}

// Remake this !!

public double FindArea()

{

List<PointEntity> Points = Polygon.Points;

double xProduct = 1;

for (int i = 0; i < Points.Count - 1; i++)

{

xProduct += Points[i].X \* Points[i + 1].Y;

}

xProduct += Points[Points.Count - 1].X \* Points[0].Y;

double yProduct = 1;

for (int i = 0; i < Points.Count - 1; i++)

{

yProduct += Points[i].Y \* Points[i + 1].X;

}

yProduct += Points[Points.Count - 1].Y \* Points[0].X;

return Math.Abs(yProduct - xProduct) / 2;

}

public bool IsInside(PointEntity point)

{

TriangleService triangleService = default;

List<PointEntity> Points = Polygon.Points;

double angleSum = default;

for (int i = 0; i < Points.Count - 1; i++)

{

triangleService = new TriangleService(new TriangleEntity(

Points[i], point, Points[i + 1]));

angleSum += triangleService.GetAngleB();

}

triangleService = new TriangleService(new TriangleEntity(

Points[Points.Count - 1], point, Points[0]));

angleSum += triangleService.GetAngleB();

if (angleSum < 359 || angleSum > 361)

return false;

else

return true;

}

// S / L == M / N

// S == L \* M / N

// S -> Find this

// L -> InkCanvas area

// M -> Count of all point

// N -> Count of point which entry inside of polygon

public double MonteCarlo(List<PointEntity> points, double canvasArea)

{

List<PointEntity> insidePoints = new List<PointEntity>();

for (int i = 0; i < points.Count; i++)

{

if (IsInside(points[i]))

{

insidePoints.Add(points[i]);

}

}

double area = canvasArea \* insidePoints.Count / points.Count;

return area;

}

public double TriangleMethod(PointEntity insidePoint)

{

if (!IsInside(insidePoint))

return 0;

TriangleService triangleService = default;

double area = default;

for (int i = 0; i < Polygon.Points.Count - 1; i++)

{

triangleService = new TriangleService(new TriangleEntity(

Polygon.Points[i], insidePoint, Polygon.Points[i + 1]));

area += triangleService.Area();

}

triangleService = new TriangleService(new TriangleEntity(

Polygon.Points[Polygon.Points.Count - 1], insidePoint, Polygon.Points[0]));

area += triangleService.Area();

return area;

}

}

}

Б.3.7 Приклад коду ITriangleService.cs

using MonteCarloMethod.Common.Entities;

namespace MonteCarloMethod.Services.TriangleServices

{

public interface ITriangleService

{

double GetAngleA();

double GetAngleB();

double GetAngleC();

double Area();

double Perimetr();

}

}

Б.3.8 Приклад коду TriangleService.cs

using MonteCarloMethod.Common.Entities;

using System;

namespace MonteCarloMethod.Services.TriangleServices

{

public sealed class TriangleService : ITriangleService

{

private TriangleEntity \_triangleEntity;

public TriangleService(TriangleEntity triangleEntity)

{

\_triangleEntity = triangleEntity;

}

public double Area()

{

double p = Perimetr() / 2;

return Math.Sqrt(p \* (p - \_triangleEntity.SideA.Length) \* (p - \_triangleEntity.SideB.Length) \* (p - \_triangleEntity.SideC.Length));

}

public double GetAngleA()

{

return \_triangleEntity.AngleA;

}

public double GetAngleB()

{

return \_triangleEntity.AngleB;

}

public double GetAngleC()

{

return \_triangleEntity.AngleC;

}

public double Perimetr()

{

return \_triangleEntity.SideA.Length + \_triangleEntity.SideB.Length + \_triangleEntity.SideC.Length;

}

}

}