МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

. другий (магістерський) .
(рівень вищої освіти)

Дослідження методів та підходів для вирішення задачі оцифрування метрологічних вимірювань

Виконав студент: . 2 . курсу групи . ІПЗм-20-3 .
. Залужний Д.В. .
(прізвище, ініціали)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми . Освітньо-наукова .
Керівник . проф. Білоус Н.В. .
(посада, прізвище, ініціали)

Допускається до захисту
Зав. Кафедри . . З.В. Дудар

Харків 2022

Харківський національний університет радіоелектроніки

Факультет . Комп'ютерних наук .

Кафедра . Програмної інженерії .

Рівень вищої освіти . другий (магістерський) .

Спеціальність . 121 – Інженерія програмного забезпечення .
(код і повна назва)

Тип програми . освітньо-наукова програма .

Освітня програма . Інженерія програмного забезпечення .

ЗАТВЕРДЖУЮ:
Зав. кафедри . .
(підпис)
«___»_____ 202_ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту . Залужному Денису Віталійовичу .
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів та підходів для вирішення задачі оцифрування метрологічних вимірювань»

затверджена наказом університету від «___»_____ 202 __ р. №___

2. Термін подання студентом роботи до екзаменаційної комісії «__» _____ 202__ р.

3. Вихідні дані до роботи: код клієнтської частини на JavaScript, код серверної частини на Python, технологія border following, система для валідації лінійок, пояснювальна записка.

4. Перелік питань, що потрібно опрацювати в роботі мета роботи, аналіз предметної галузі і постановка задачі, дослідження методів розпізнавання контурів зображення, моделювання та програмна реалізація системи, проведення експериментів та тестування.

КАЛЕНДАРНИЙ ПЛАН

Номер	Назва етапу роботи	Термін виконання етапу роботи	Примітка
1	Аналіз предметної галузі	17.01.2022	виконано
2	Постановка задачі	31.01.2022	виконано
3	Формування вимог	07.02.2022	виконано
4	Проведення експериментів	07.03.2022	виконано
4	Проектування	15.03.2022	виконано
5	Реалізація системи	20.03.2022	виконано
6	Проведення тестування	20.04.2022	виконано
6	Підготовка звіту до кваліфікаційної роботи	25.04.2022	виконано
7	Перевірка на плагіат	10.05.2022	виконано
8	Перевірка на нормоконтроль	11.05.2022	виконано
9	Попередній захист	20.05.2022	виконано
10	Захист кваліфікаційної роботи	23.05.2022	виконано

Дата видачі завдання 17.01.2022 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Білоус Н.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 72 с., 19 рис., 5 дод., 21 джерел.

МЕТРОЛОГІЯ, ЛІНІЙКИ, КОМП'ЮТЕРНИЙ ЗІР, ШТУЧНИЙ ІНТЕЛЕКТ, API, OPEN CV, REST, DJANGO.

Об'єктом дослідження є процес оцифрування метрологічних вимірювань шляхом використання алгоритмів комп'ютерного зору.

Метою роботи є дослідження методів та підходів для вирішення проблеми оцифрування метрологічних вимірювань. Використовуючи оцифрування, оптимізувати процес перевірки метрологічних приладів державним станадртам

Методи розробки базуються на таких технологіях, як Python, Django, Open CV, Docker, AWS.

В результаті роботи було досліджено та порівняно методи виділення кордонів зображення, приведення зображення до чорно-білого формату, проведення аналіз та моделювання предметної області та програмно реалізовано базовий інтерфейс для валідації металевих лінійок.

METROLOGY, RULES, COMPUTER VISION, ARTIFICIAL INTELLIGENCE, API, OPEN CV, REST, DJANGO.

The object of research is approaches of digitizing metrological measurements using computer vision algorithms.

The aim of the work is to increase the efficiency of metrological tools validation according to state standards.

Development methods are based on such technologies as Python, Django, Open CV, Docker, AWS.

As a result, the problems of cortour recognition and gray-scale convertion were investigated and compared. Analysis and modeling of subject area was performed. The basic interface to validate metal rules was implemented.

Умови публікації пояснювальної записки

Я, . Залужний Денис Віталійович .
(прізвище, ім’я, по батькові)

студент групи . ІПЗм-20-3 . здобувач вищої освіти на другому (магістерському) рівні

кафедра . програмної інженерії .,
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему «Дослідження методів та підходів для вирішення задачі оцифрування метрологічних вимірювань», що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Сьогодення цілком і повністю можна назвати світом технологій. Це стосується будь-який науковий вектор і тим паче підприємства, що постійно ведуть боротьбу на ринку за лідерство. В тому числі варто відмітити військову галузь, що надзвичайно актуальна під час агресивних дій з боку російської федерації.

Проте будь-яка технологія не можлива без фізичних деталей, що для будь-якого приладу повинні відповідати заданим стандартам. Тобто ключовим гравцем у процесі створення будь-якої деталі постає метрологічний процес, що надає чіткі розміри та форму.

Насправді проблема може бути не тільки у висоті, ширині і т. д. Будь-яка фізична величина може стати слабким місцем для приладу (наприклад тиск, концентрація певної речовини, густина та інше). Зазвичай для вирішення таких проблем використовуються спеціальні прилади вимірювання, що в своїй складовій мають шкалу, за якою і підбирається необхідне значення.

І тут постає інша не менш актуальна проблема: хто встановлює саму шкалу, та відповідає за її достовірність? Адже навіть при невеликій погрішності результати можуть бути катастрофічними (наприклад вибухи атомних ЕС, несправність літаків).

Взагалі кажучи, на даний момент для цього існують ДСТУ та спеціальні метрологічні центри, що перевіряють на відповідність цим ДСТУ. Як не дивно у світі технологій перевірка ДСТУ все ще, в переважній більшості, робота мануальна.

Як результат маємо актуальну проблему: швидкість перевірки та кількість задіяної людської роботи. Тобто для перевірки однієї людини як мінімум задіяний один спеціаліст (і найчастіше другий спеціаліст для повторної перевірки та зменшення вірогідності людського фактору). А з урахуванням кількості пунктів у ДСТУ, то перевірка одного елементу може зайняті до одного дня. І це знову ж таки при наших реаліях, розвитком комп’ютерних технологій.

Варто зазначити, що у питаннях ДСТУ важко повністю перекинути відповідальність на ПО, проте можна врази полегшити пошук очевидних проблем, які програма може ідентифікувати майже моментально з вказівкою на відповідний пункт у ДСТУ.

Серед наукових досліджень кафедри ПІ є декілька досліджень пов’язаних з системами комп’ютерного зору. В першу чергу зазначимо досліди сегментації зображення алгоритмом Watershed [1, 2], а також досліди з оптимізації waywlet перетворювань [3, 4]. Також зазначимо, що розміри метричних шкал можуть бути будь-якої довжини, тому також було розглянуто дослід tracing алгоритмів [5].

Метою роботи є оптимізація процесу перевірки метрологічних приладів на відповідність державним стандартам.

Об’єктом дослідження є методи комп’ютерного зору для аналізу контурів зображення, встановлення об’єктів на зображенні та віповідні величини цих об’єктів.

Предметом дослідження є підходи і методи для оцифрування метрологічних вимірювань.

В основному використано емпіричні методи дослідження, а саме експеримент та спостереження. На основі них побудовано гіпотезу та перевірено її на всьому об’ємі даних.

В результаті створено унікальний метод оцифрування, що буде ефективно справлятися у вказаному середовищі.

Як було вказано вище, метричні прилади використовуються у широкому колі задач, а тому прискорення ефективності перевірки їх стосовно стандартів напряму впливатиме на економіку цілої країни.

На основі даної роботи було написано і опубліковано роботу в дванадцятій міжнародній науково-технічній конференції [6], де описано досягнуті результати та найефективніше поєднання методів аналізу зображення.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

В першу чергу варто зазначити, що для різних шкал використовуються різні ДСТУ. Зокрема звернемо увагу на металеві лінійки розміром до 3 м. Для цього типу використовують ДСТУ 427-75 [7]. Оскільки для різних шкал використовуються різні ДСТУ, тобто різні правила, що необхідно перевіряти та дотримувати, то розробити універсальне рішення перевірки також доволі складно (що найменш ДСТУ можуть оновлюватися з часом). Тому в якості предметної області було обрано саме металеві лінійки розміром до 3 м, як наглядний кандидат для демонстрації автоматизованих процесів з використанням систем машинного зору як інструмент для спрощення процесу перевірки на відповідність ДСТУ.

Існують певні правила, що необхідно дотримуватися при створенні та випуску нової металевої лінійки, розміром до 3 м. В першу чергу відстань між поділками повинна становити 1мм. Іншими словами необхідно перевірити всі пари сусідніх поділок на відповідне правило.

Відповідно до типу шкал, то лінійки можуть бути з однією шкалою, з двома шкалами (зверху та знизу), а також з двома шкалами (оцифрування яких направлене у різних напрямках).

Також накладаються обмеження на розмірності поділок та лінійки в залежності від меж вимірювань. Для вимірювання відстаней до 50 см, ширина лінійки повинна бути відповідно в межах 18 – 22 мм. А от для відстаней до 300 см такі межі вже становлять 36 – 40 мм.

Відповідні обмеження існують також для товщини лінійки. Для відстаней до 50 см, товщина повинна складати 0,4 – 0,6 мм. А от для відстаней до 300 см такі межі вже становлять 0,8 – 2 мм.

Аналогічні обмеження накладаються на довжини поділок. Зокрема довжина міліметрової поділки для відстаней до 50 см повинна складати 3,5 мм. А для

відстаней до 300 см довжина міліметрової позначки вже повинна відповідати 5 мм.

Довжина пів сантиметрових поділок для відстаней до 50 см повинна складати 5 мм, а для відстаней до 300 см вже 7 мм відповідно. Для сантиметрових поділок такі довжини відповідно будуть 6,5 мм та 9 мм.

На числові позначки також встановлені обмеження. Їх довжина не повинна бути меншою за 3 мм. І відповідно ширина кожної поділки повинна бути в межах 0,2 мм.

Ще одним відомим обмеженням є те, що різниця між довжинами міліметрових, пів сантиметрових та сантиметрових поділок повинна бути не менше ніж 1,5 мм.

1.2 Аналіз підходів та методів рішення

Для того, щоб перевірити дану металеву лінійку на відповідність до ДСТУ використовують спеціальний еталон, що приймається за ідеальний екземпляр. І вже наступні порівняння відштовхуються від цього еталону. Таким чином в процесі перевірки приймають участь вже дві лінійки (еталон також може бути лінійкою).

Описаний вище підхід використовується на даний момент в метрологічних центрах. Також існують системи з використанням комп’ютерного зору, що надають можливість автоматично визначити відстань між об’єктами. Проте в таких системах критичним місцем є потенційні похибки, що можуть випливати з нечіткості самого зображення.

Для максимальної ефективності та чіткості вимірювань варто використовувати одночасно як людські ресурси, так і алгоритми комп’ютерного зору. Таким чином отримуємо переваги двох підходів: можливість чіткого корегування та швидке розпізнавання однотипних елементів зображення. Залишається головна проблема, а саме під час фотографування чи відео зображення не знаходиться в ідеальному 2D положенні декартової системи координат. Це в свою чергу призводить додаткові похибки.

Для вирішення проблеми описаної вище можна скористатися декількома підходами: поділом зображення на невеликі частини (дозволяє максимально зменшити величину похибки) та трансформація зображення (переводить спотворене зображення в декартову систему).

Варто зазначити, що першим кроком необхідно розділити еталонну лінійку та лінійку над якою проводиться перевірка. Це необхідно зробити, щоб наступними кроками чітко проводити алгоритми комп'ютерного зору над конкретним зображенням еталону чи тестового зображення.

На цьому етапі можна розглянути декілька широко відомих алгоритмів сегментування зображення (адже саме сегментування необхідне для розбиття зображення на відповідні логічні елементи).

Яскравим прикладом є алгоритм Watershed [8]. Його основна концепція пішла з географії та полягає у тому, щоб розбити зображення на логічні мінімуми (географічні заглибини) та поступово заповнювати їх водою. На кожному кроці заповнення, краї місцевості будуть становитися все коротшими і в результаті вода із одних заглибин об'єднається з водою інших заглибин. Щоб обійти такий ефект необхідно для рідини кожної заглибини надати свою мітку. Таким чином рідини з різними мітками не можуть об'єднуватися в суцільну водойму.

Переводячи на мову зображень отримаємо, що заглибини це мінімуми на матриці. Проте першим кроком необхідно привести зображення до чорно-білого виду. Тоді відповідно діапазон значень кожного елемента матриці буде становити [0, 255] або відповідно [0, 1]. Де 0 відповідає фону або чорному кольору. З іншого боку основа зображення або білий колір.

Після переведення до описаного вище виду можна вже працювати з зображенням як матрицею цілих чисел. На цьому етапі можемо знайти локальні мінімуми та почати процес заповнення заглибин.

Проте в даному підході є значний мінус. Якщо зображення складається з великої кількості невеликих об'єктів, то отримаємо купу локальних мінімумів і як результат велику кількість елементів що насправді повинні розпізнаватися як єдиний елемент.

Для вирішення цієї проблеми можна скористатися рішенням з мануальним встановленням якорів. Тобто користувач вказує самостійно до якого якоря відноситься та чи інша частина зображення. В цілому чим більше таких якорів встановлено, тим точнішим буде результат. Проте на практиці достатньо декількох уточнюючих якорів (адже сам алгоритм розбиття на сегменти залишається таким самим).

Наступним кроком важливо провести накладання роздробленого зображення в єдине ціле. Проте і тут можна зазначити, що це не є обов'язковим. При умові що кожна частина лінійки буде розмічена користувачем (а це точно можливо, що на зображенні також знаходиться еталонна лінійка, для якої точно відомі розміри), то можна порахувати похибки в незалежності від інших частин лінійки.

Ще одним важливим перетворення є виділення кордонів зображення. Це необхідно для ідентифікації позначок на цільовій лінійці для подальшого порівняння їх з еталоном. Для цього звернемо увагу на алгоритм, що називається "Border following" [9]. В першу чергу зазначимо, що алгоритм обробляє лише бінарні зображення, тобто зображення що представлені пікселями 1 (контури зображення) чи 0 (фон). Та в результаті породжує послідовності між поєднаними пікселями, що дорівнюють 0 чи 1 відповідно.

Основна ідея алгоритму полягає у тому, щоб побудувати ієрархічну систему між кордонами елементів зображення. Зазначимо що кордони можуть бути зовнішніми (outer) та внутрішніми (hole). І так на початку маємо лише один зовнішній кордон, його ще називають фрейм зображення (складається з чотирьох бокових сторін зображення). Також для підтримки ієрархії будемо зберігати дві змінні: NBD (new border) та LNBD (last new board) відповідно. На початку NBD дорівнюватимеме 1, так як знайдено лише один кордон (фрейм зображення). Далі на кожному кроці будемо збільшувати значення цієї змінної.

Сам алгоритм приймає наступний вид. Проходимо по кожному пікселю зображення (i, j), де i відповідає номеру рядка; j відповідно номеру стовпця. Зауважимо, що беремо лише пікселі для яких $j \geq 1$. Далі для кожного обраного

пікселя перевіряємо чи відноситься він до зовнішнього контуру (outer border) чи внутрішнього контуру (hole border). Для зовнішнього контуру беремо за основу піксель ліворуч, а для внутрішнього контуру відповідно праворуч. Назвемо цей піксель (i_2, j_2).

Тепер для (i, j) пікселя починаємо перебирати пікселі за годинниковою стрілкою, починаючи з (i_2, j_2) та обираємо перший ненульвий піксель. Назвемо його (i_1, j_1).

Далі використовуємо циклічний алгоритм, до тих пір поки не повернемося в піксель з якого починали, тобто (i, j). Ідея полягає у тому щоб вже для (i_2, j_2) підбираємо ненульовий піксель проти годинникової стрілки починаючи з (i, j). При знаходженні елемента робимо дві операції: підраховуємо ідентифікатор кордону для пікселя (i, j) та переносим його відповідно на місце пікселя (i_2, j_2).

1.3 Аналіз існуючих систем

Якщо розглядати системи, що на даний момент використовуються в метрологічних центрах, то в основному це стандартні програми такі як Microsoft Word та Microsoft Excel. А основна автоматизація яка використовується це Excel формули для автоматичного розрахунку похибок.

Стосовно самого процесу вимірювання, то все проходить з використанням людського ресурсу.

Проте не зважаючи на це, існують системи для автоматичного вимірювання відстаней. Прикладом такої системи є Aruler [10]. Система надає великий спектр різноманітних вимірювань: довжини, об’єми, висоти і багато іншого. Проте для поставленої задачі є великий недолік: обмеження встановлені з точністю до міліметра, а точність системи вимірюється у сантиметрах. Ще одним недоліком є встановлення мітками в режимі online. Тобто тримаючи телефон у руках, користувач повинен відмітити всі точки прямої, площини чи іншої фігури що необхідно виміряти. Вже на цій стадії доволі просто отримати значні похибки (вважаючи що необхідна точність вимірювання у міліметрах).

Відразу можна зазначити, що для поставленої задачі буде набагато ефективніше мати змогу встановити відмітки вже після отримання зображення (з можливістю його приближення).

Зазначимо, що є розробки для мануального проектування систем машинного зору, даючи при цьому можливість не вдаватися в деталі певної мови програмування, наприклад Merlic [11]. Проте є декілька недоліків такого роду систем: вони не вузько спеціалізовані на певній проблемі. Тобто можна спроектувати рішення, проте його точність не буде максимально можливою. Крім того, при необхідності додати нового функціоналу чи модифікувати існуючий алгоритм, також отримаємо суттєві труднощі.

1.4 Постановка задачі та функціональні вимоги до системи

В першу чергу зазначимо, що система повинна надати користувачу можливість передавати фото лінійки. В результаті надається звіт з похибками лінійки від еталону.

Враховуючи, що лінійки можуть бути різного розміру, то необхідно надати можливість розширення. А саме користувач може передати набір фотографій послідовних частин лінійки.

Окрім цього користувач вказує чому дорівнює поділка еталону. Відштовхуючись від цього значення будуть розраховані всі наступні величини.

Також користувач може при необхідності вказати якорі для лінійки та еталону для більш точної їх сегментації (відповідно до алгоритму Watershed).

Система надає можливість отримати й завантажити результати перевірки. До того ж можна вказувати колір та товщину нанесених результуючих позначок.

Зазначимо, що розмір зображення може доходити до декількох десятків мегабайтів, при цьому алгоритм провинен працювати по часовим рамкам до 3 хв. Система повинна витримувати навантаження 10000 запитів у секунду і повертати при цьому коректний результат.

Користувачу надається зручний інтерфейс для знаходження відповідних параметрів на зображенні лінійки, в тому числі співвідношення пікселів зображення до реальних поділок на лінійці.

Для користуванням системою користувачу необхідно мати пристрій з доступом до мережі Інтернет та можливість завантажити відповідні зображення лінійок. Система повинна працювати на максимально можливому числі пристроїв, тому це повинен бути веб додаток. Вважаємо, що майже кожен комп’ютер чи мобільний пристрій має влаштований браузер.

Мережа передачі даних повинна бути захищена HTTPS протоколом, що надає можливість пересилати будь-які дані, не зважаючи на можливість викрадення.

При виявленні помилок, система повинно максимально чітко надати користувачу інформацію про причини помилки, та що необхідно чи можливо зробити, щоб виправити відповідну помилку.

2. ДОСЛІДЖЕННЯ ТА АНАЛІЗ МЕТОДІВ РОЗРОБКИ

Оскільки система проектується для перевірки гіпотез та аналізу ефективності подальшого використання, то важливо мати під рукою інструмент, з допомогою якого можна швидко вносити зміни.

Такими інструментами є Django (Python фреймворк для проектування серверних рішень) [12], OpenCV (тут також зазначимо, що скористаємося Python оберткою) [13].

Для реалізації поставленої задачі скористаємося архітектурним рішенням REST, тобто розробимо серверне рішення для обробки зображень та клієнтський додаток для отримання даних від користувача та відповідно відображення результату користувачу. Зазначимо, що для зручного проектування можна скористатися бібліотекою Django Rest Framework, що розширює стандартні можливості Django, відповідно до вимог REST.

Розглянемо основні положення, що закладені в RESTful принципи. В першу чергу це принцип єдиного інтерфейсу. Його головна ідея полягає в тому, що клієнт може змінювати стан сервера та отримувати доступ до ресурсів через відповідний інтерфейс. До того ж кожен ресурс має свій унікальний шлях URI за яким клієнт і отримує данні про нього. Відповідно клієнт повинен знати лише один URL – положення самого серверу, а всі інші інтерфейси сервер генерує у відповідь на базовий запит.

Наступним принципом йде клієнтсько-серверна архітектура. Ідея цього принципу у тому, щоб перекласти роботу за відображення та збирання даних у користувача на клієнтський додаток, а відповідну логіку з маніпулювання даними та станом системи на серверну частину. Таким чином полегшується процес додання нових клієнтів системи (таких як веб додатки, мобільні додатки, IoT та смарт годинники, інше). Також в такому випадку простіше проводити розгортання на сервері та розбиття репозиторіїв.

Далі йде принцип, що дослівно називається stateless. Його ідея полягає в тому, що кожен запит складається з усіх необхідних даних, щоб повністю

ідентифікувати необхідну інформацію на серверній частині. А ще кожен запит повністю незалежний, тобто неважливо які запити було проведено до цього. Такий підхід корисний при розвертанні додатка на декількох серверах, тобто горизонтальному розширенні системи. Зважаючи, що обсяги зображень можуть бути доволі великими, важливо продумати шляхи підвищення обчислювальних можливостей. Зазвичай використовують паралельні підходи в системі самої програми (але при цьому відладка та підтримка програми суттєво поскладнюється), або ж просто додають більше серверів щоб розбити між ними рівномірно кількість запитів.

Після цього звернемо увагу на принцип кешування. Завдяки ньому клієнт може запам’ятати на деякий час результати запитів та не повторювати їх певний час. Наприклад, при обробці одного й того ж зображення чи при запиті конфігурацій, що рідко змінюються. Зазначимо, що час на який дані актуальні і відповідно їх можна спокійно зберегти в локальному сховищі, зазначаються самим сервером.

Також важливим принципом являється принцип багатошарності. Він зазначає що система може складатися з декількох пластів, кожен пласт логічно незалежний та повинен комунікувати тільки з сусіднім за абстракцією пластом. Це надає можливість швидко вносити модифікації в архітектуру, при цьому не пребудовуючи систему цілком, що дуже важливо в умовах проектування та експериментів.

Останнім принципом з RESTful виступає код за вимогою. Яскравим прикладом виступає JavaScript чи CSS в стандартних веб додатках. Такий підхід доволі корисний, вже на етапі завантаження додатку, адже можна не чекати коли всі дані завантажаться, а відображати частини сторінки за готовністю. Ще варто відзначити спеціалізовані CDN (content delivery network) сервера, що допомагають прискорити доступ до даних такого типу.

Для зручності розгортання скористаємося технологією Docker [14] та Docker-Compose [15]. Такий підхід надає можливість запускати відразу декілька програм у заздалегідь підготовленому середовищі. Для того щоб скористатися

підготовленим середовищем реалізовано спеціалізовані Dockerfile файли, які зберігають інформацію про необхідні кроки для запуску додатку під спеціальну операційну систему. Далі з такого Dockerfile необхідно побудувати image та залити у спеціальний docker репозиторій, з якого вже на сервері можна побудувати відповідний контейнер. Контейнер у свою чергу представляє реалізацію підходу контейнеризації, що є більш ефективним рішенням над віртуалізацією, бо використовує те саме ядро операційної системи для запуску програми. За замовченням контейнери базуються на інтерфейсі pthread, що підтримуються тільки в UNIX based операційних системах, тому в якості операційної системи на орендованому сервері встановимо відповідний дистрибутив Linux, наприклад Ubuntu (як приклад стабільності). Також зазначимо, що при кожній новій зміні в додатку необхідно проводити перебудову image. В цілому це не довгий процесс, бо основні залежності кешуються і перебудова буде впливати лише на нові частини системи.

У якості HTTP серверу скористаємося стандартним рішенням, а саме Nginx [16]. Він необхідний для роутингу запитів і працює зі статичними файлами. Тобто його основна задача – співвідносити запит до відповідного ресурсу, що знаходится на сервері. Для роботи з Python серверами, одного тільки HTTP серверу буде недостатньо, бо окрім статичних даних ще необхідно обробляти динамічні запити для обробки даних. Проблема в тому, що Nginx не знає архітектури Django серверу і тому не зможе визивати відповідні view, що оброблятимуть запити. Для вирішення цієї проблеми існують спеціальні WSGI сервери. Це спеціальний шлюз на який Nginx буде перенаправляти запит, і вже з нього визиватиметься спеціальна view функція, що й оброблятиме запит. Це працюватиме завдяки тому, що в Python існує спеціальний стандарт для веб серверів, на якому і базується робота WSGI серверу. В якості такого WSGI серверу оберемо Gunicorn, як один з стандартів серед своїх аналогів.

У якості інструмента бекенд сервера обрано Django. В перш за все через те що на Python багато готових бібліотек, а також доволі просто інтегрувати бібліотеки із мови C.

Для обробки зображень скористаємося бібліотекою OpenCV. Це багата бібліотека з різноманітними алгоритмами для обробки зображень, в тому числі в ній є реалізація необхідного алгоритму Watershed (залишається лише підібрати найефективніші коефіцієнти та правильні якоря). Окрім цього вже готова обгортка на Python, тобто можна весь функціонал використовувати на рівні Python. Окрім того, важливою перевагою є відкритий код бібліотеки, що надає можливість при необхідності зробити модифікацію та підналаштувати алгоритм під себе.

В якості інструменту для клієнтської частини скористаємося бібліотекою React.js [17]. Це спеціальна бібліотека під мову програмування JavaScript, що дозволяє зручно будувати інтерфейси клієнтської частини, розбиваючи його на атомарні компоненти. В результаті код написаний на React.js перетворюється в спеціалізовані сжаті файли (зазвичай з використанням спеціалізованого менеджера webpack) і в результаті використовується звичайні HTML, CSS та JavaScript.

Дані будемо передавати в форматі JSON (що в цілому є каноном для REST). Тут варто зазначити, що зображення це матриця пікселів. Такої структури немає в форматі JSON. Тому скористаємося перетворенням зображення в формат Base64 [18]. Тепер зображення в такому виді можна спокійно передавати в JSON форматі.

В той же час, за умови великих зображень формат Base64 може займати багато пам’яті, а тому значний час при передачі. Тому на практиці при передачі файлів зазвичай використовують не JSON + Base64, а multipart formdata [19, 20]. Ідея підходу заключається в тому, що кожен байт кодується у вигляді hexadecimal представлення, що в результаті більш економно стосовно затраченої пам’яті.

На вихід програма буде повертати зображення з результатами валідації лінійки. Зображення буде в форматі PNG, як одного із найчастіше зустрічаємих типів зображення, прте в той же час його доволі просто перевести в будь-який інший тип зображення.

Зазначимо також декілька стратегій оптимізації реалізованої системи. По-перше, враховуючи архітектуру системи, можна просто скористатися горизонтальним розширенням. Найпростіше його можна реалізувати використовуючи так називаємі auto scale groups. Такі сервіси доступні майже на всіх PaaS платформах (наприклад AWS). З іншого боку це додаткові фінансові розходи на підтримку виділених серверів.

Іншим кроком оптимізації є перехід з WSGI на ASGI. Різниця в тому, що у WSGI кожен worker обробляє кожен запит послідовно, а в ASGI – асинхронно. Це значить, що поки worker чекає на відповідь від клієнта, то він може починати обробку іншого запиту (ще такий тип задач називається IO bound задачі, тобто задачі, що базуютьна на ввід/вивід).

Окрім того доступним варіантом є використання JIT компіляторів Python замість звичайного CPython. В даному випадку частини коду, що повторюються автоматично оптимізуються (в деяких випадках підставляється код на C) і виконується швидше. Звичайно, використовуючи бібліотеку OpenCV та NumPy більшість коду вже буде реалізована на C, і тому дуже важливо користуватися цими перевагами і намагатися використовувати по максимуму функції з цих бібліотек.

Крайнім варіантом, проте доволі ефективним буде перехід на ще один вид компілятора, а саме Cython. Різниця буде в тому, що на Cython можна напряму використовувати C бібліотеки та примітиви. Тобто все ще маємо Python інтерфейс, проте напряму поєднаний з C.

Також як аналог всим компіляторам, що були описані вище, зазначимо ще мову програмування Nim. Її перевага в даному випадку в тому, що вона дуже схожа на Python, але в той же час строго типізована і компілюється в нативний код відповідної платформи. Тобто можна скомпілювати під відповідний сервер найважкіші частини коду, використовуючи мову Nim і зробити просто вставки в алгоритм, що реалізований на мові Python.

Деякі алгоритми можна додатково розпаралелити, розбивши виконання на декілька процесів.

3. МОДЕЛЮВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 UML-моделювання

Наведено Use Case діаграму, основуючись на функціональних вимогах системи (рисунок 3.1).

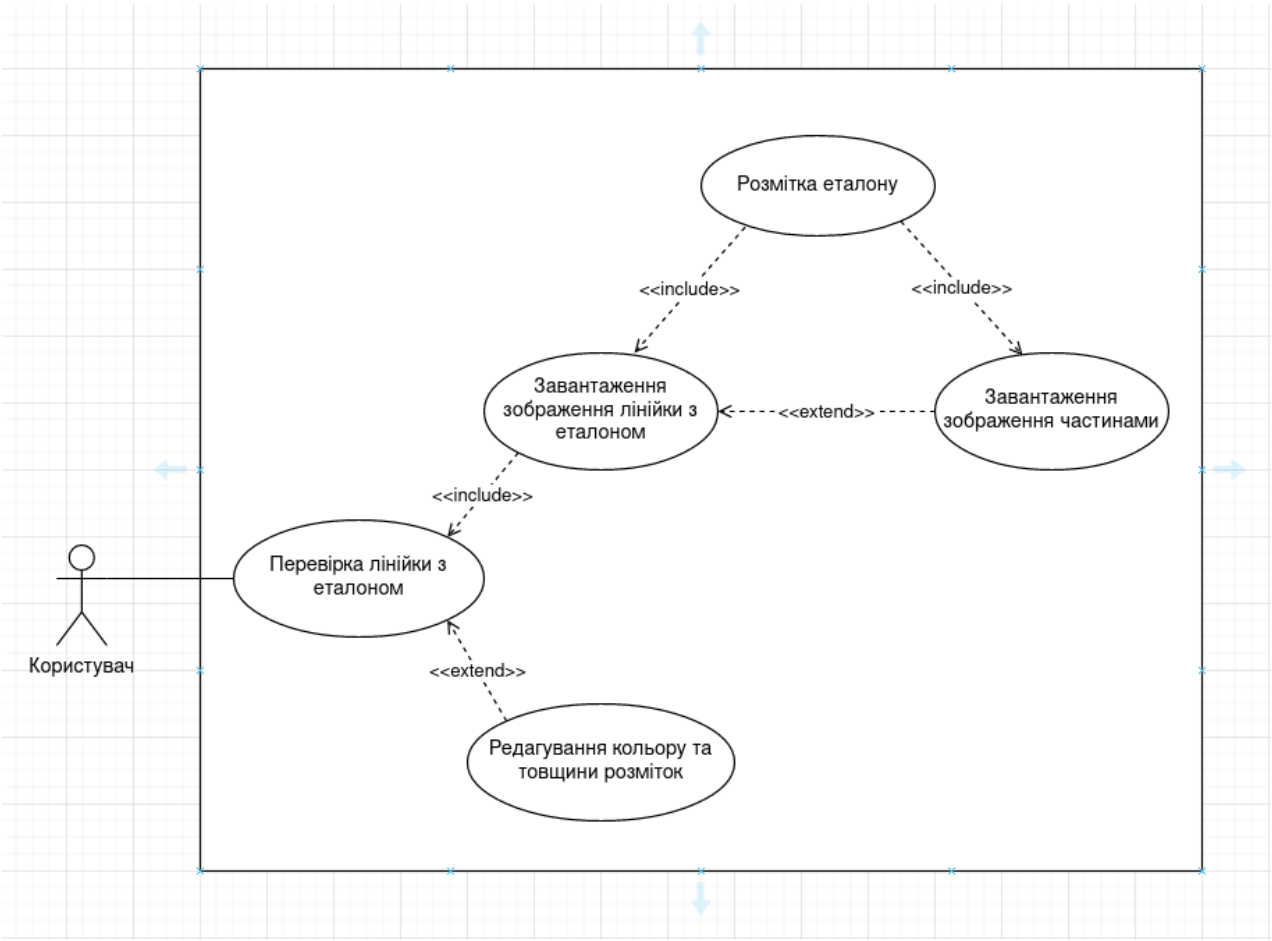

Рисунок 3.1 – Use Case діаграма ПО

Основною функцією системи буде перевірка лінійки на відповідність стандартам, шляхом порівняння з еталоном. Додатково користувач може налаштувати колір та товщину відображення позначок на результуючому зображенні.

Для того, щоб провести перевірку необхідно завантажити саме зображення поряд з еталоном по якому буде проходити перевірка зображення. Оскільки лінійка може бути доволі довгою, то надається додаткова можливість завантажити

зображення частинами. Як було описано вище, це дозволить знизити вплив похибок на результат (зазначимо що ідеальний варіант це фото лінійки під прямим кутом).

Процес завантаження лінійки також включає в себе розмітку еталону. Тобто користувач встановлює дві точки та їх розмірності (вважаємо що для еталону вони точно відомі). Також в цей процес включається процес встановлення якорів для алгоритму Watershed.

Спроектуємо більш детально сам процес аналізу та обробки зображення лінійки та наведемо відповідну діаграму активності (рисунок 3.2).

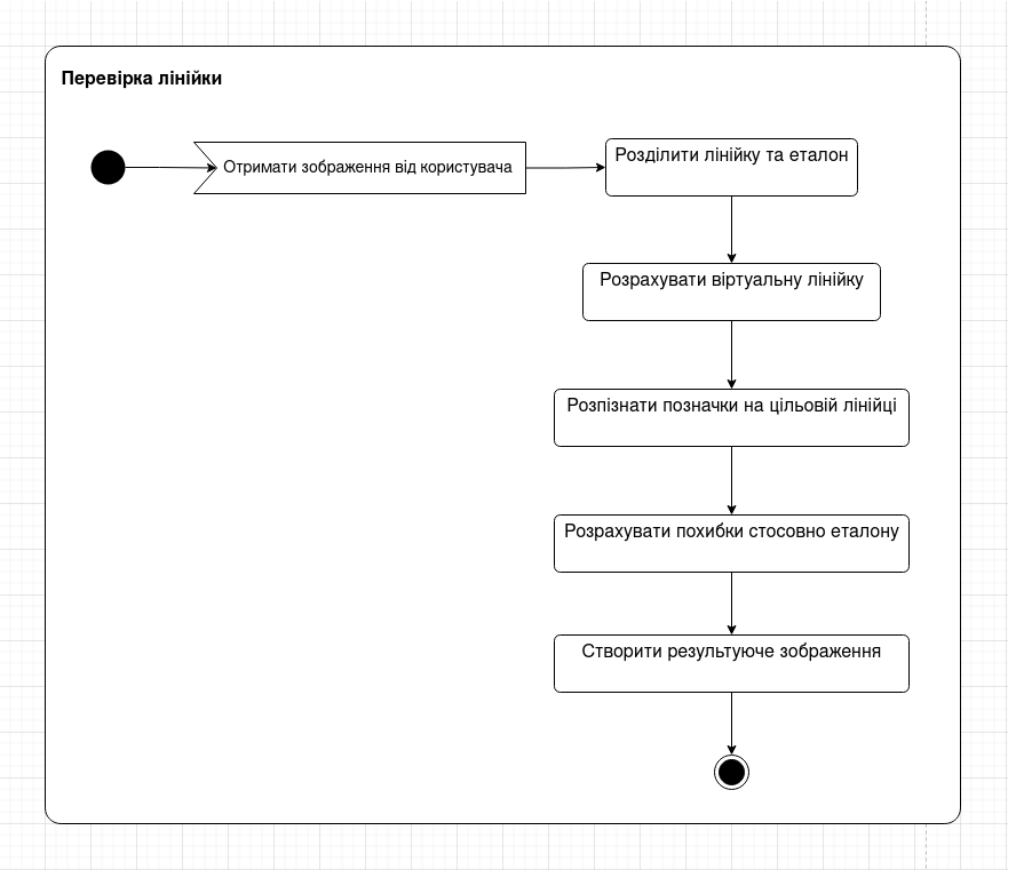

Рисунок 3.2 – Діаграма активності

Першим кроком система отримує запит від користувача на аналіз зображення лінійки. Зазначимо що дані будуть передаватися в форматі JSON, в якому відповідно окремим буде поле з самим зображенням, а також дані про розмітку еталону.

Наступним кроком необхідно розділити на окремі зображення лінійку та еталон. Проте перед цим треба перевести зображення з формату Base64 до матричного виду, кожен елемент якої буде зберігати колір формату RGB.

Для розбиття отриманого зображення, спочатку проставимо мануальні якоря (що також передаються користувачем) і запустимо алгоритм Watershed, деталі якого було описано вище.

На цьому етапі отримаємо зображення еталону та зображення самої лінійки. Можна переходити до розрахунку віртуальної лінійки. Для цього просто виведемо функцію через одну невідому. Щоб це зробити розрахуємо ціну поділки для 1мм за формулою:

$$\frac{x_2 - x_1}{l_2 - l_1}$$

Тут x_i відповідна координата в пікселях, l_i відповідно значення поділки в міліметрах. Знаючи ціну поділки та точку старту наносимо віртуальну лінійку на зображення лінійки для подальшого порівняння. Під час нанесення беремо до уваги параметри, що користувач може передати у запиті для встановлення кольору та товщини.

Тепер необхідно розпізнати відповідно зазначені позначки на лінійці для порівняння з еталоном. Для цього скористаємося методом для виділення внутрішніх кордонів. В результаті отримаємо список позначок, відсортованих за зростанням відповідно до осі координат.

Наступним кроком проходимо по всіх позначках віртуальної лінійки та порівнюємо їх у відповідність з отриманими результатами знайдених позначок тестової лінійки. Якщо порушується поріг заданий у стандарті, то виділяємо відповідну ділянку червоним кольором та зазначаємо значення похибки. В іншому випадку просто наносимо віртуальну лінійку зеленим кольором.

Також розглянемо алгоритм скануючої прямої. Він полягає в тому, щоб пропустити пряму через позначки метричної шкали та перевіряти елементи, що

пересеклися. Перед цим важливо провести початкове очищення зображення, щоб позбутися погрішностей, що залежатимуть від якості зображення (рис. 3.3).

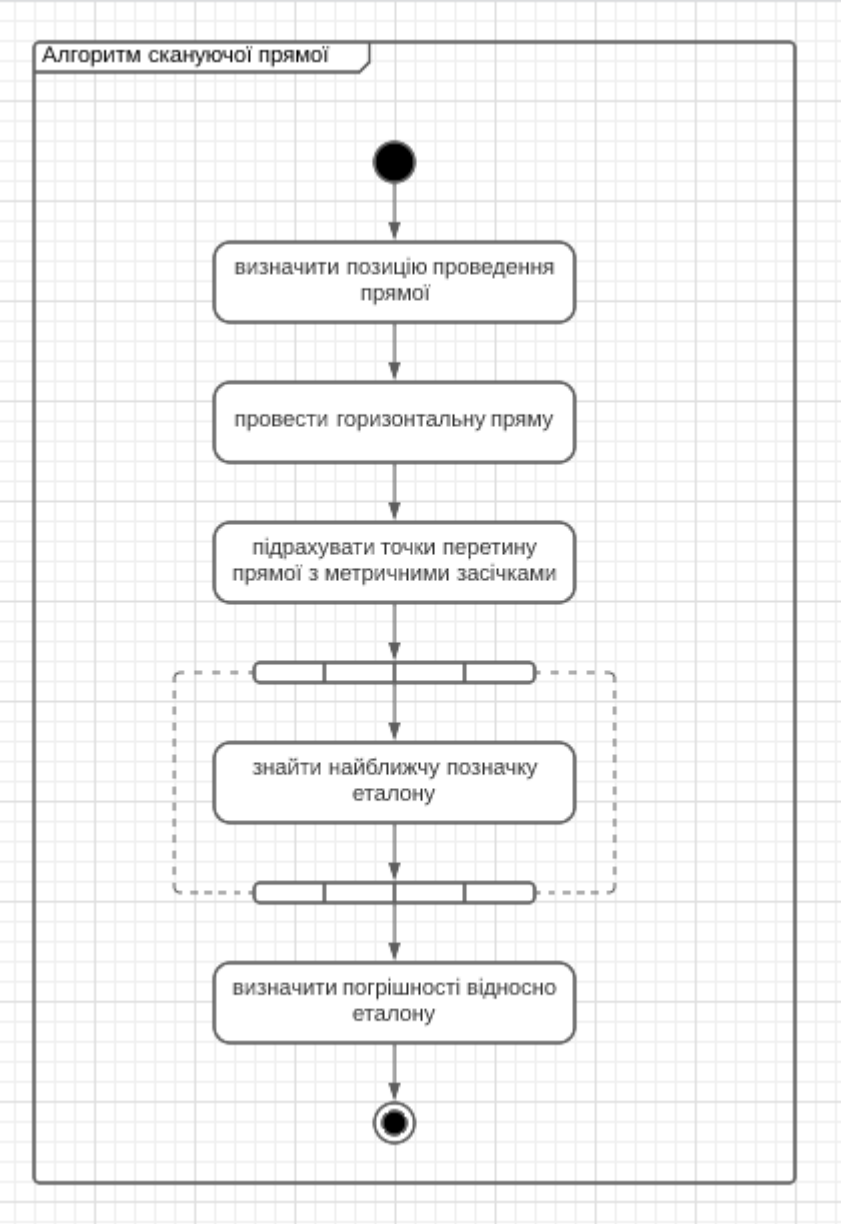

Рисунок 3.3 – Алгоритм скануючої прямої

Як зазначено в діаграмі, для кожного перетину проходимо по позначках еталону, та підбираємо позначку з найменшою відстаню від перетину.

3.2 Опис програмного забезпечення

Програмне забезпечення складається з клієнтської та серверної частин відповідно. Серверна частина відповідає за прийняття, обробку та повернення валідованих даних лінійки. Відповідно клієнтська частина відповідає за зручне та ефективне прийняття даних від користувача, передачу їх на сторону сервера та відображення результату.

Серверна частина представляє собою реалізацію архітектурного підходу REST. Тобто очікуються запити без зберігання стану. Це обумовлено тим, що потрібно розробити систему у короткі строки для перевірки ефективності представленого алгоритму. Також зазначимо, що в якості протоколу передачі даних будумо використовувати HTTP, що обумовлено тими ж причинами.

У якості інструменту реалізації було обрано мову програмування Python та відповідний фреймворк для розробки веб-серверів Django. Окрім того, для більш швидкої реалізації REST використовується спеціалізована бібліотека DRF.

Для зчитування та валідації даних використаємо спеціалізовані серіалізатори, що дістануть дані із HTTP запиту та перенесуть їх відповідно до об’єкту з валідними параметрами.

Тепер варто зазначити, які базові параметри необхідні для виконання валідації отриманого зображення лінійки. В першу чергу, для чіткого встановлення віртуальної лінійки необхідно встановити шкалу. Це можна легко зробити, знаючи дві точки (в пікселях зображення) та яка між цими пікселями відстань (в реальному житті). Тобто просто установити дві точки, відомих вже поділок еталонної лінійки. Кожна точка представляє собою пару значень (адже зображення приходить у 2D). Відповідно очікуємо наступні параметри: known_start_x, known_start_y, known_end_x, known_end_y. І відповідно ще реальний міліметраж цих точок: known_start_mm, known_end_mm. Приклад серіалізатору для валідації та отримання параметрів, що передав користувач, наведено нижче:

```
class CheckMetalRuleSerializer(serializers.Serializer):
    rule_image = serializers.ImageField()
    known_start_x = serializers.IntegerField()
    known_start_y = serializers.IntegerField()
    known_end_x = serializers.IntegerField()
    known_end_y = serializers.IntegerField()
    known_start_mm = serializers.IntegerField()
    known_end_mm = serializers.IntegerField()
```

Після отримання відповідних параметрів, визивається спеціальний метод для валідації, що приймає на вхід параметри, а в результаті повертає зображення з відповідними позначками віртуальної лінійки еталону та валідовані позначки цільової лінійки.

На цьому етапі варто зазначити багато деталей реалізації, адже REST підходи не встановлюють чітких правил реалізації, а лише базові принципи проектування інтерфейсів. В першу чергу звернемо увагу, що серед параметрів також передається саме зображення, що може набувати розмірів більше ніж 10 MB. Тому в якості формату оберемо MultipartFormData [19, 20]. Зображення також можна перевести до формату base64 і передавати у вигляді послідовності ASCII симовлів, проте у випадку великих зображень, кількість затраченої пам’яті та необхідного часу будуть перевищувати зазначені ліміти на HTTP сервер.

Також важливим нюансом є формат самого зображення. Тому щоб звести все до уніфікованого виду, першим кроком переводимо його до RGB (тобто з байтового виду до матриці із трьох значень: red, green, blue відповідно). І на сам кінець, після обробки отриманого зображення матимемо також матрицю RGB, тому необхідно її перевести знову до послідовності байтів і повернути як результат HTTP запиту. Приклад реалізації класу, що відповідає за обробку HTTP запиту стосовно валідації металевої лінійки та передачі відповідної відповіді наведено нижче:

```
class CheckMetalRuleAPIView(APIView):
    serializer_class = CheckMetalRuleSerializer
    parser_classes = [MultiPartParser]

    def post(self, request, *args, **kwargs) -> HttpResponse:
        self._fetch_valid_parameters(request)
```

```
        rule_checker = MetalRuleChecker(
            Image.open(self.rule_image).convert('RGB')
        )
        marked_img = rule_checker.check(
            start=self.known_start_point,
            end=self.known_end_point,
            start_mm=self.known_start_mm,
            end_mm=self.known_end_mm
        )

        img_bytes = convert_image_to_bytes(marked_img)
        return HttpResponse(img_bytes, content_type='image/png')
```

Для валідації було реалізовано клас MetalRuleChecker. Основна логіка цього класу закладена в методі check. Його логіка розділена на два основні сегмети: виокремлення/нанесення позначок віртуальної лінійки на зображення та виокремлення/нанесення позначок цільової лінійки на зображення. Для роботи з зображенням було використано дві бібліотеки: PIL для базової обробки зображень (в тому числі Django використовує цю бібліотеку в своїх цілях, коли обробляє зображення); OpenCV для алгоритмів комп'ютерного зору.

Детальний приклад алгоритму для виокремлення віртуальної лінійки еталонного зображення:

```
img = convert_pil_image_to_cv2(self.rule_image)
painter = RulePainter(img)
virtual_line = LineEquation(start, end)
one_mm = (end.x - start.x) / (end_mm - start_mm)

virtual_rule_points = []
x = start.x
for _ in range(start_mm, end_mm+1):
    y = virtual_line.get_y(x)
    virtual_rule_points.append(
        Point(int(x), int(y))
    )
    x += one_mm
painter.paint_virtual_rule(virtual_rule_points)
```

Серед наведеного алгоритму є декілька важливих моментів, що також варто вказати. Зокрема обробка зображень проходить з використанням бібліотеки OpenCV, а вона в свою чергу використовує спицифічний формат BGR, в той час

як PIL бібліотека працює зі звичайним RGB. Як результат необхідно виконувати спеціалізовані приведення форматів перед обробкою.

Для відображення знайдених позначок було реалізовано домоміжний клас RulePainter.

Відповідно, щоб за координатою x можна було отримувати відповідні координати y, відштовхуючись від двох відомих точок, було реалізовано ще один допоміжний клас LineEquation, що реалізує рівняння прямої, що проходить через дві відомі точки:

```
class LineEquation:
    def __init__(self, first: Point, second: Point):
        self.x1, self.y1 = first.x, first.y
        self.x2, self.y2 = second.x, second.y

    def get_y(self, x: float) -> float:
        up = (x - self.x1) * (self.y2 - self.y1)
        down = self.x2 - self.x1
        return self.y1 + up / down
```

Тепер розглянемо реалізацію алгоритму для знаходження відповідних замірів цільової лінійки.

В першу чергу виокремемо частину зображення з верхніми замірами. Вважаємо, що цільова лінійка знаходиться на зображенні під еталонною. Шляхом експериментів на різних зображеннях було виокремлено оптимальну частину, а саме 20% від ширини зображення:

```
metrics_img = target_rule_img[:round(0.2 * height)]
```

Тепер для наступного виокремлення переводимо зображення до чорно-білого формату. Для цього проведемо два кроки. В першому зведемо зображення так називаємого “gray scale” виду. В такому випадку це вже не буде RGB, де кожен піксель відповідає трійці значень відповідного каналу; а матимемо для кожного пікселя кодування в діапазоні [0; 255].

В такому виді можна рухатись далі та провести так би мовити приведення значень пікселів. Оскільки необхідно отримати чорно-біле зображення, то всі проміжні значення відповідно повинні прийняти значення 0 або 255. Для цього скористаємося бібліотекою OpenCV:

```
gray = cv.cvtColor(metrics_img, cv.COLOR_BGR2GRAY)
_, thresh = cv.threshold(gray, 170, 255, cv.THRESH_BINARY)
```

Розглянемо реалізацію алгоритму виділення із чорно-білого зображення метричних позначок цільового зображення:

```
x = 0
target_points = []
while x < thresh.shape[1]:
    if thresh[70][x] == 0:
        start = x
        while x < thresh.shape[1] and thresh[70][x] == 0:
            x += 1
            target_points.append(
                Point(
                    (start + x) // 2, 0
                )
            )
    else:
        x += 1
```

Головна ідея заключається у використання та називаємого scan line підходу. Тобто беремо пряму і пропускаємо її через певний рядок матриці зображення. І для кожної послідовності із 0 (іншими словами чорний колір) відміряємо середню позицію.

Важливий момент, чого не варто брати скануючу пряму від самого початку. Причиною стають розмитості в зображення на стикуванні декількох лінійок. Щоб обійти вплив розмитостей на кінцевий результат, практичним шляхом було виокремлено відстань в 5% від ширини зображення.

Тепер переглянемо frontend частину систему, головна задача якої полягає у встановленні відповідності між відстаннями в реальному житті до пікселей на

самому зображенні. Також додамо базовий інтерфейс для відправки зображення з відповідними параметрами на серверну частину.

Для знаходження відповідності пікселів зображення, розроблено інтерфейс, що надає можливість завантажити зображення та відображає відносну позицію курсору на даному зображенні. Скористаємося HTML кодом для розмітки:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <script src="./index.js"></script>
    </head>
    <body>
        <input id="src" type="file" />
        <img id="target" />

        <div id="position"
             style="position: fixed; top: 40px; left: 40px;">
        </div>
    </body>
</html>
```

Також наведемо приклад JavaScript коду, для обробки завантаження зображення та відмальовування відповідної позиції у пікселях. Для цього скористаємося спеціальним обробником подій mousemove (для підрахунку нової позиції при руху курсора) та change (для відмальовування зображення при виборі нового файлу):

```
function showImage(src_file, target_image) {
    let fr = new FileReader();
    fr.onload = function(event) {
        target_image.src = this.result;
    };

    src.addEventListener("change", function() {
        fr.readAsDataURL(src_file.files[0]);
    });
}

window.onload = function() {
    let src_file = document.getElementById("src");
    target_image = document.getElementById("target");
    positionBlock = document.getElementById("position");

    showImage(src_file, target_image);
```

```
    target_image.addEventListener("mousemove", function(evt) {
        positionBlock.innerHTML = `${evt.offsetX}, ${evt.offsetY}`;
    });
}
```

Наведемо приклад інтерфейсу для знаходження пікселів. Для цього користувачу надається можливість завантажити зображення та відображається позиція курсору на зображенні в режимі реального часу. Таким чином передвигаючи курсор можна доволі точно встановити співвідношення міліметражу лінійки до її відповідних значень у пікселях (рисунок 3.4). Відповідно при зміні файлу значення оновлюються як самого зображення так і позиції курсору.

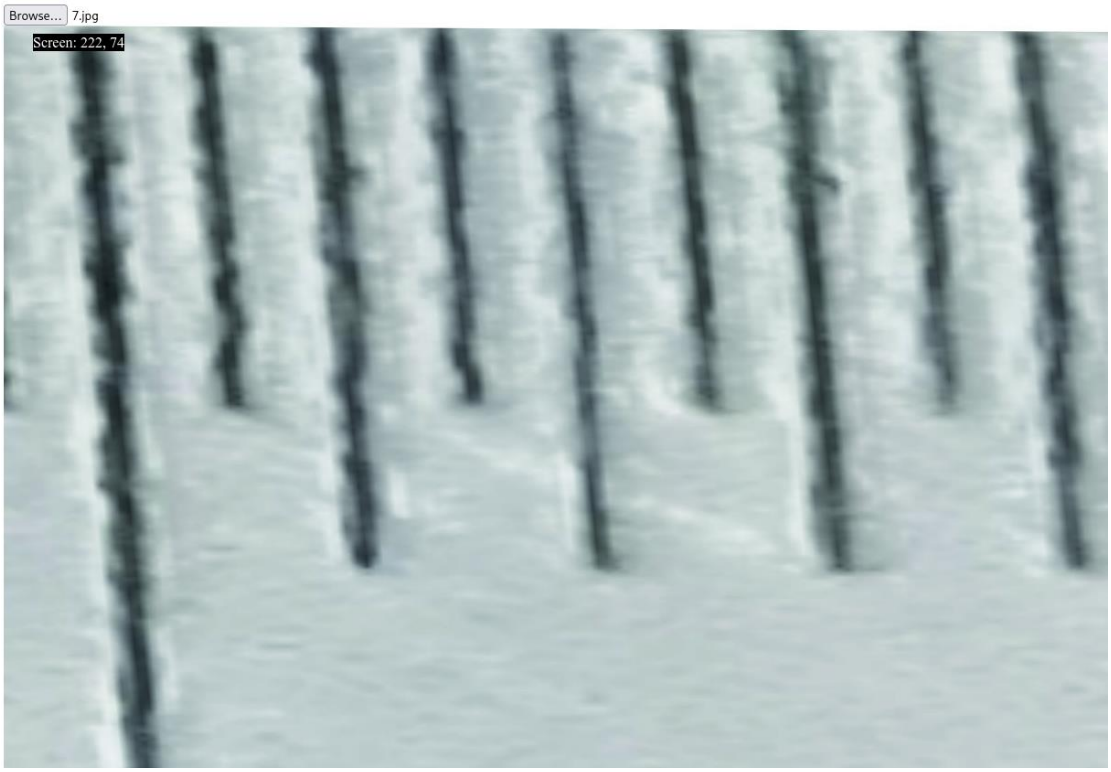

Рисунок 3.4 – Інтерфейс для встановлення пікселів

Розглянемо приклад інтерфейсу для валідації зображення. Для цього користувачу надається можливість передати положення пікселів для позначок відомих міліметрів, самі міліметри (що відповідають вказаним пікселям) та відповідний файл зображення.

В результаті відбувається POST запит з передачею в body параметрах всіх вказаних користувачем замірів (до того зазначимо, що ці значення додатково кодуються та передаються через multipart/formdata).

В якості результату отримуємо відвалідоване зображення лінійки з нанесеними відповідними позначками віртуальної лінійки еталону, цільової лінійки та похибок між цими значеннями (рисунок 3.5). Зазначимо що алгоритм базується саме на невеликих похибках, що не повинні перевищувати 1 мм. Іншими словами ідея системи в знаходженні мало помітних похибок, а не тих, що одразу видно людському зору.

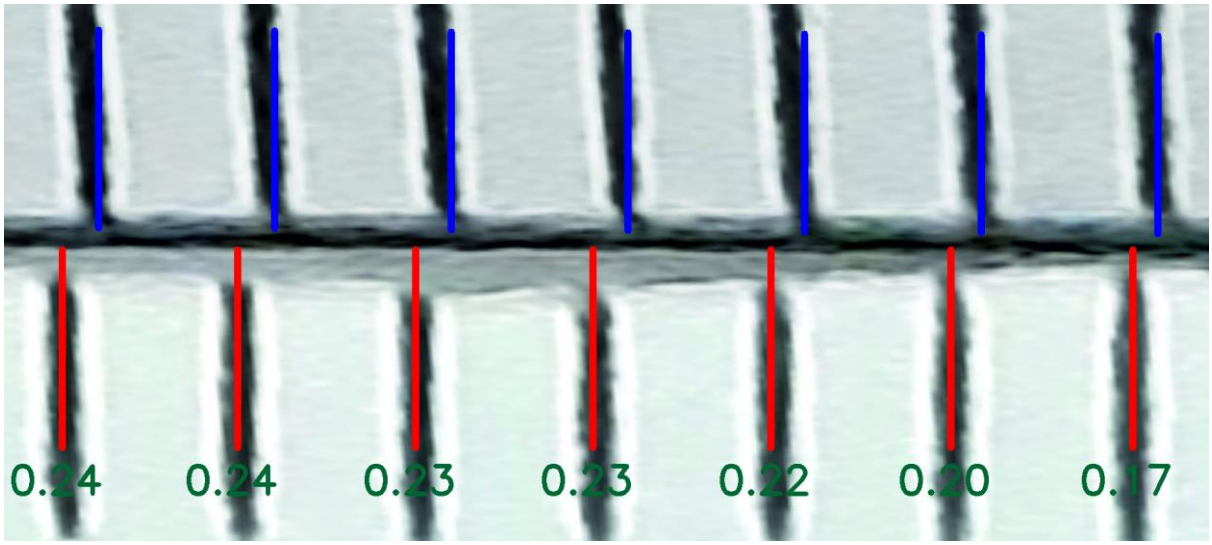

Рисунок 3.5 – Результат валідації металевої лінійки

Зазначемо, що за замовчуванням віртуальна лінійка еталону зображена синім кольором, а знайдені позначки цільової лінійки мають відповідно червоний колір. Позначки похибок зображені зеленим кольором та знаходяться під позначками цільової лінійки.

4. ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ТЕСТУВАННЯ ПЗ

Експериментальні дослідження в основному проводилися над частиною системи, що відповідає за розпізнавання позначок цільової лінійки. Для цього було використанно набір алгоритмів комп'ютерного зору, компоненти якого найбільше доповнюють один одного в контексті поставленої задачі.

Розглянемо експерементальні дослідження над алгоритмом визначення засічок цільової лінійки. В першу чергу розглянемо результати, за умови що алгоритм буде аналізувати зображення цілком. Також зазначемо, що в якості початкового алгоритму було обрано алгоритм визначення контурів Border Following.

Для вказаного алгоритму зображення повинно бути в чорно-білому форматі, тобто кожен піксель має значення 0 або 255. Таким чином впершу чергу необхідно перевести вхідне кольорове зображення до чорно-білого формату. Для цього скористаємося наробками та дослідженнями розробників бібліотеки OpenCV. В цілому для переведення зображення із формату RGB до чорно-білого необхідно реалізувати функцію трансформації трьох каналів до одного. Для цього просто кожен канал домножається на відповідний коефіцієнт, а їх сума буде результатом. В якості коефіцієнтів бібліотека OpenCV використовує відповідно: 0.299, 0.587, 0.114. Для ознайомлення наведемо приклад відповідного коду з модулів бібліотеки, що реалізовує цей функціонал [21]:

```
template<typename _Tp> struct RGB2Gray
{
    typedef _Tp channel_type;

    RGB2Gray(
        int _srccn, int blueIdx, const float* _coeffs
    ) : srccn(_srccn)
    {
        static const float coeffs0[] = { 0.299f, 0.587f, 0.114f };
        memcpy(
            coeffs,
            _coeffs ? _coeffs : coeffs0,
            3*sizeof(coeffs[0])
        );
        if (blueIdx == 0)
            std::swap(coeffs[0], coeffs[2]);
```

```
    }

    void operator()(const _Tp* src, _Tp* dst, int n) const
    {
        int scn = srccn;
        float cb = coeffs[0], cg = coeffs[1], cr = coeffs[2];
        for(int i = 0; i < n; i++, src += scn)
            dst[i] = saturate_cast<_Tp>(
                src[0]*cb + src[1]*cg + src[2]*cr
            );
    }
    int srccn;
    float coeffs[3];
};
```

Приклад коду для ковертації зображення в форматі Gray scale з формату BGR, використовуючи оболонку Python:

```
source_img = cv2.cvtColor(rule_img, cv2.COLOR_BGR2RGB)
gray_img = cv2.cvtColor(rule_img, cv2.COLOR_BGR2GRAY)

fig = plt.figure(figsize=(30, 10))
ax = fig.add_subplot(121)
ax.imshow(source_img)

ax = fig.add_subplot(122)
ax.imshow(gray_img, cmap='gray')
```

Результатом виконання будуть два зображення. Ліворуч початкове зображення в форматі RGB, а праворуч відповідно у форматі Gray scale (рисунок 4.1). Звернемо також увагу на засвітлення та затемнення: вони передаються однаково як на RGB, так і Gray scale прикладі. Це важливий фактор, адже більшість алгоритмів дають чіткі результати лише на однорідних зображеннях, іншими словами на таких зображеннях, де різниця значень між сусідніми пікселями мінімальна. Це обумовлено структурою підходу трансформації.

Для того, щоб прийти до однорідності можна використовувати різного роду узагальнення. Наприклад, є підхід рухаючого вікна, ідея якого полягає в тому, щоб орієнтуватись не тільки на значення сусідніх пікселів, а відразу на значення пікселів у вказаному діапазоні. Таким чином, регулюючи величину вікна, можна

отримувати результати різної контрастності, проте середнє значення на результуючому зображенні буде максимально однорідним.

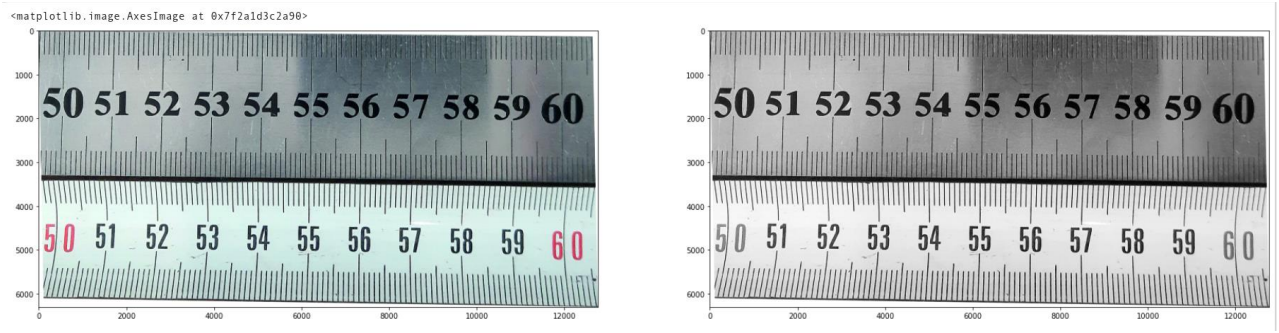

Рисунок 4.1 – Приведення до gray scale схеми

Наступним кроком зображення з Gray scale необхідно перевести в чорно-білий формат. Для цього скористаємося підходом кордонів, у якому встановимо після якого значення пікселі будуть розпізнаватися як білі. Такий підхід називається binary threshold. Наведемо приклад коду, що реалізує такий тип алгоритму, на оптимально підібраних параметрах:

```
_, thresh = cv2.threshold(gray_img, 170, 255, cv2.THRESH_BINARY)

fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(121)
ax.imshow(gray_img, cmap='gray')

ax = fig.add_subplot(122)
ax.imshow(thresh, cmap='gray')
```

Розглянемо результати роботи такого алгоритму на різних зображеннях. Якщо зображення однотипного окрасу, то алгоритм показує точні результати (рисунок 4.2). Проте за умови низької якості зображення чи нерівномірного освітлення отримуємо відповідні затемнення, що скажуться на подальшій частині алгоритму (рисунок 4.3). Також як видно із прикладів зображень, всі подряпини лінійки також впливають на результат, тому доволі важливо, щоб лінійка відповідала максимально її початковому виду. Також зазначимо, що подряпини

камери також вносять свої похибки до результату. Вважатимемо, що лінійка та камера знаходяться в ідеальному стані.

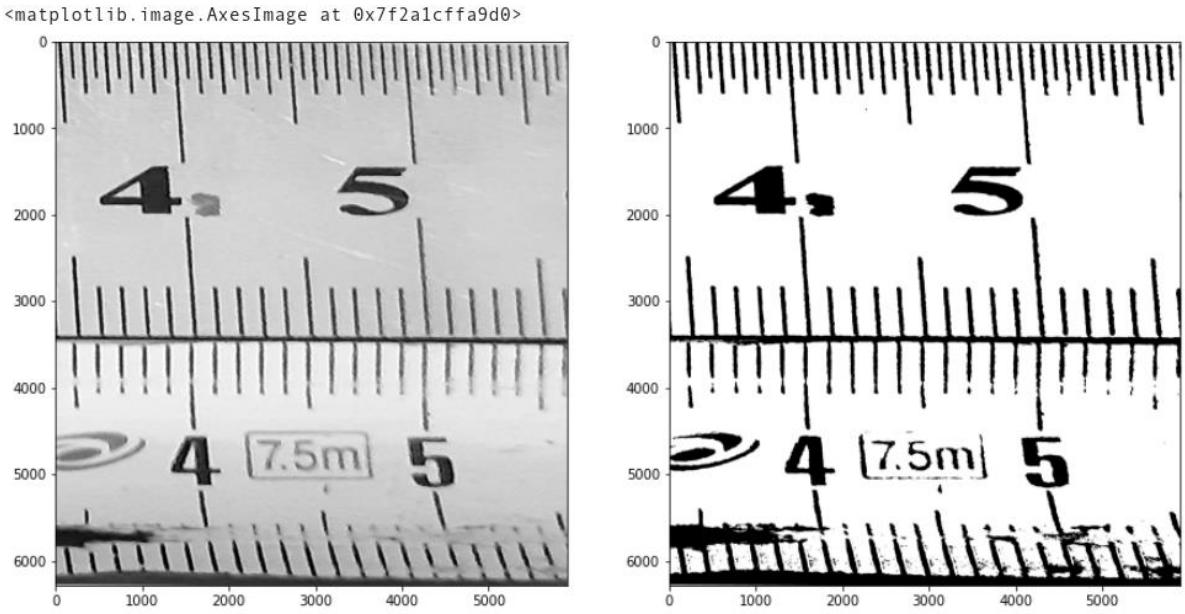

Рисунок 4.2 – Приведення до чорно-білої схеми

Для порівняння скористаємося підходом з адаптивними кордонами чи ще як його називають adaptive thresholding.

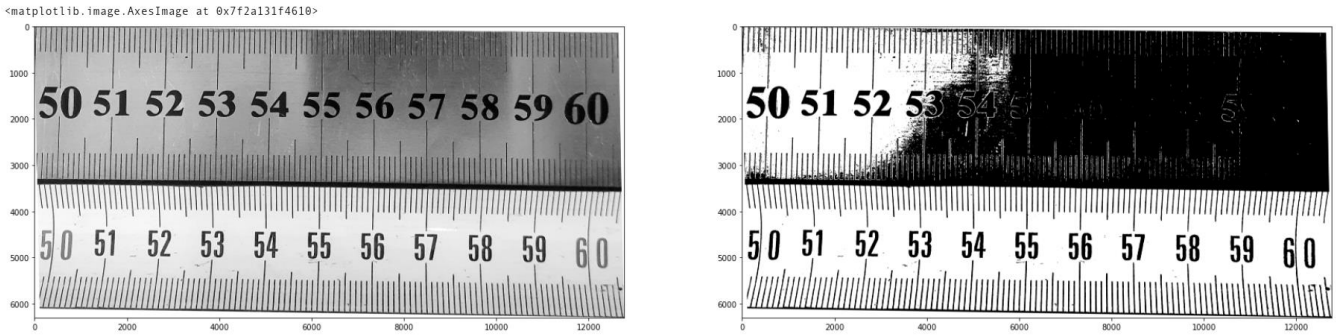

Рисунок 4.3 – Приведення до чорно-білої схеми з затемненням

У якості реалізації adaptive thresholding скористаємося гаусівською сіткою. Ідея полягає в тому, щоб робити аналіз спираючись не на значення одного пікселя, а на значеня пікселів, що знаходяться в сітці обраного розміру. Також алгоритм приймає ще один параметр: константу, що буде вираховуватись з

кожного результуючого пікселя. Наведемо приклад такого алгоритму через Python обертку:

```
thresh = cv2.adaptiveThreshold(
    gray_img,
    255,
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
    cv2.THRESH_BINARY,
    51,
    8
)

fig = plt.figure(figsize=(30, 10))
ax = fig.add_subplot(121)
ax.imshow(gray_img, cmap='gray')

ax = fig.add_subplot(122)
ax.imshow(thresh, cmap='gray')
```

Результат роботи цього алгоритму на прикладі наведеного раніше зображення (рисунок 4.4).

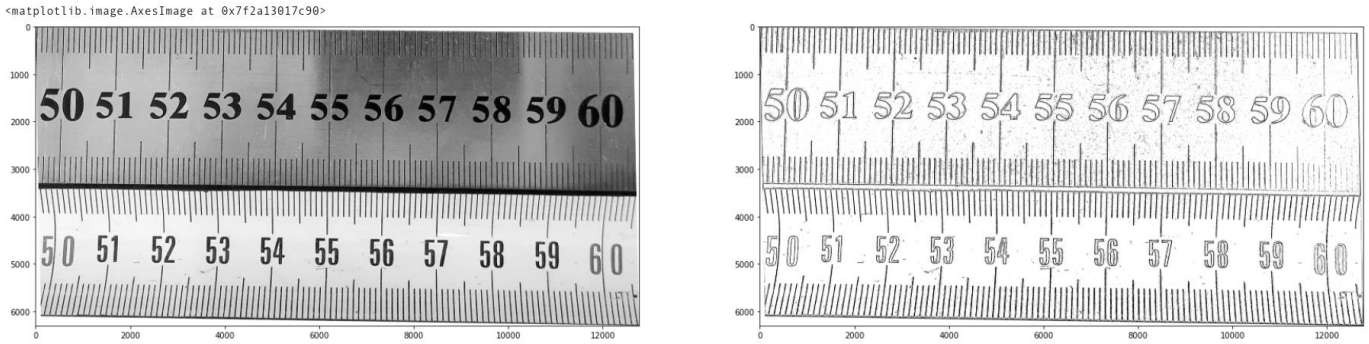

Рисунок 4.4 – Результат адаптивного алгоритму

Результат також залежить від розширення зображень. В даному випадку зображення подані з розміром 6000 x 12000, тобто сітка розміром 50 x 50 замала для обчислення чіткого результату. Наведеом приклад алгоритму адаптивних кордонів для сітки розміром 210 x 210 (рисунок 4.5). Як бачемо тепер частинами видно чорні точки, проте в той же час результуюче зображення стало набагато чіткішим.

Як відомо алгоритм приймає ще один параметр, а саме константу. Збільшивши її значення, можна зменшити кількість чорних точок, що також позитивно впливатиме на кінцевий результат.

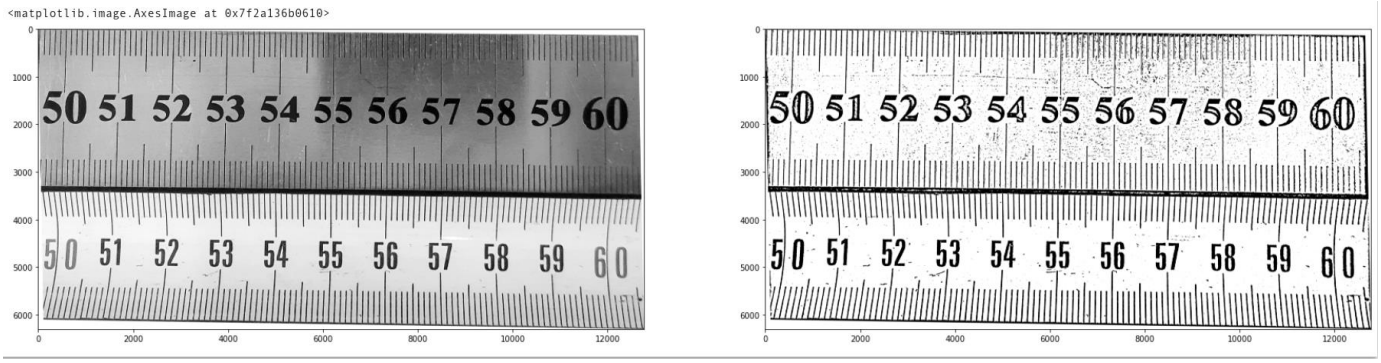

Рисунок 4.5 – Адаптивний алгоритм з більшою сіткою

Як видно, результат із більшою константою отримав набагато менше побічних чорних точок (рисунок 4.6).

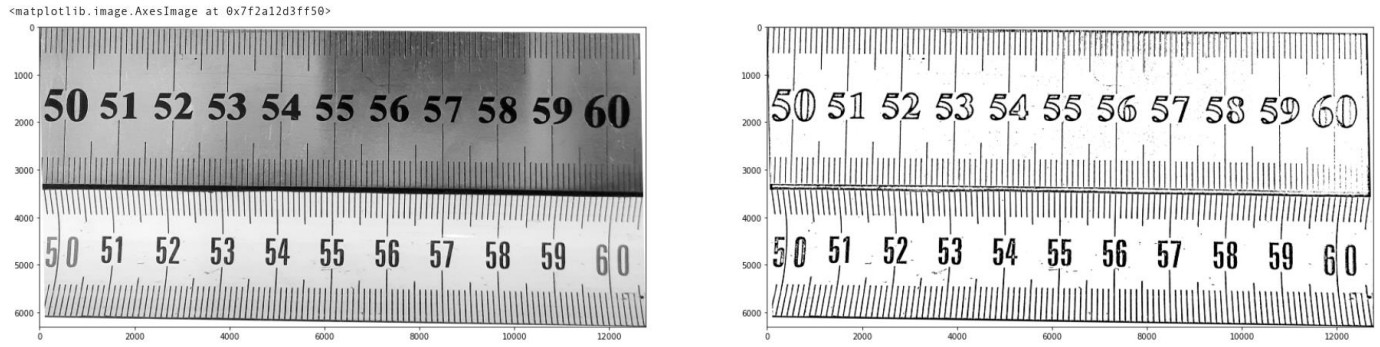

Рисунок 4.6 – Адаптивний алгоритм з більшою константою

Отримавши чорно-біле зображення тепер необхідно виділити із нього контури засічок, для подальшого порівняння засічок еталонної лінійки та цільової.

Розглянемо алгоритм border following. Його ідея дещо схожа з алгоритмом пошуку в ширину, проте з деякими модифікаціями для пошуку саме контурів. Зазначимо, що алгоритм знаходить одночасно як зовнішні, так і внутрішні

контури. Наведемо приклад реалізації алгоритму scan line з використанням Python обертки:

```
target = thresh[3440:]
contours, hierarchy = cv2.findContours(
    target, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE
)

drawing = np.zeros(target.shape)
color = 255
for ind in range(len(contours)):
  cv2.drawContours(
    drawing,
    contours,
    ind,
    color,
    2,
    cv2.LINE_8,
    hierarchy,
    0
  )

fig = plt.figure(figsize=(30, 10))
ax = fig.add_subplot(121)
ax.imshow(target, cmap='gray')

ax = fig.add_subplot(122)
drawing = abs(drawing - 255)
ax.imshow(drawing, cmap='gray')
```

Розглянемо результат виконання наведеного алгоритму на декількох зображеннях, що вже приведені до чорно-білого формату.

Для початку розглянемо на невеликому діапазоні частину лінійки від 4 до 5 сантимертів (рисунок 4.7). Як бачимо контури визначено доволі чітко, але в той же час зображення доволі світле, тобто слабка контрастність. Насправді причина у великих масштабах та товщині ліній, що відображають кордони. Іншими словами дуже велика кількість пікселів зображення стосовно 1 мм лінійки. Відповідно одним з рішень може бути збільшення товщини ліній, що нанесено на результуюче зображення, але в той же час збільшення товщини веде до збільшення похибки результуючого зображення з контурами.

Зазначимо, що на результат валідації контарстність не впливає, адже для результату важлива не контрастність, а те що чітко виділені контури значеннями відповідними числовими значеннями.

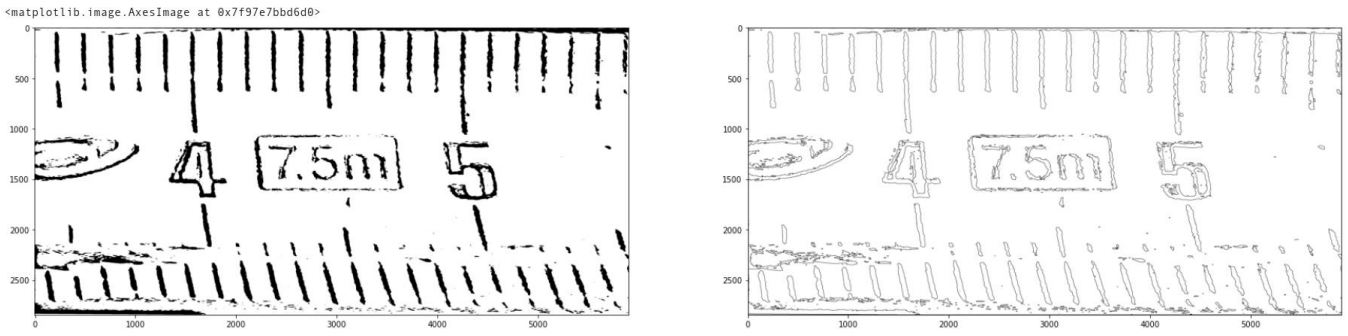

Рисунок 4.7 – Алгоритм знаходження кордонів

Також розглянемо виконання на зображенні з більшим діапазоном, а саме на 10 сантиметрів (рисунок 4.8).

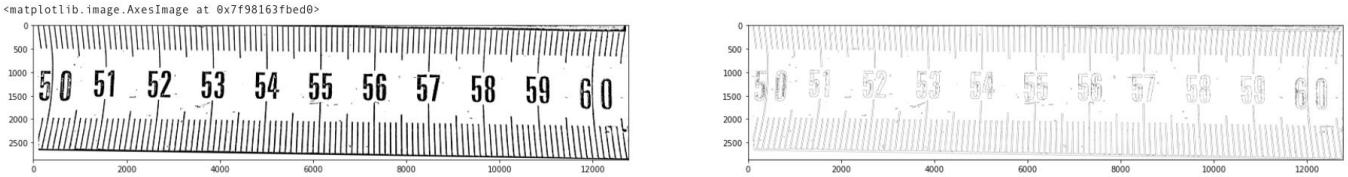

Рисунок 4.8 – Алгоритм знаходження кордонів на більшому діапазоні

Зазначемо, що на даному етапі відображаються всі кордони. Для встановлення відхилень необхідно визначити відповідно тільки верхню частину позначок.

Також в рамках даної дазачі, можна оптимізувати алгоритм пошуку кордонів, скориставшись відразу алгоритмом скануючого променя. Ідея алгоритму заключається в тому, щоб визначати відповідні міліметрові засічки, як точки перетину вертикальних засічок цільової лінійки з горизонтальною прямою.

Наведемо приклад реалізації алгоритму scan line, що проходить по зазначеній горизонтальній прямій та визначає точки, як середнє значення із послідовності чорних відрізків. Також зазначимо, що можуть бути похибки чи

неточності зображення, тому кожну знайдену таку точку, потім необхідно співвіднести до найближчої засічки:

```
x = 0
target_points = []
width = target.shape[1]
while x < width:
  if target[0][x] == 0:
      start = x
      while x < width and target[0][x] == 0:
          x += 1
      target_points.append(
          ((start + x) // 2, 0)
      )
  else:
    x += 1
```

Запустимо описаний вище алгоритм через зображення в діапазоні 10 сантимерів (рисунок 4.9). Як видно засічки визначено доволі чітко, проте головною проблемою є паралельність лінійок на зображенні до осі абцис. Критично важливо, щоб промень проходив через усі позначки лінійки, бо в іншому випадку не всі позначки будуть ідентифіковані і використані при валідаційному порівнянні.

Для оптимізації алгоритму можна використати полярну систему координат, і тоді підналаштовувати здвиг лінійки відповідно під цю систему, проте такі обчислення тою чи іншою мірою приводять до похибок, що важко виправити людині через звичайний зоровий контакт.

Ще однією проблемою може стати непаралельність лінійок на зображенні одної відносно іншої. Система базується на тому що лінійки на зображенні паралельні. Знову ж таки шляхом математичних перетворень зображення можна дійти до паралельності, проте це додаткові похибки. Тому найпростішим підходом буде послідовний спуск скануючої прямої до тих пір, доки пряма не перетне всі засічки лінійки.

Єдине зазначимо, що у випадку, коли лінійки паралельні, але камера зробила знімок під певним кутом, і відповідно осі камери не співпадатимуть з

осями лінійки, то в такому випадку можна просто зробити оберт всього зображення на визначений кут без накладних похибок.

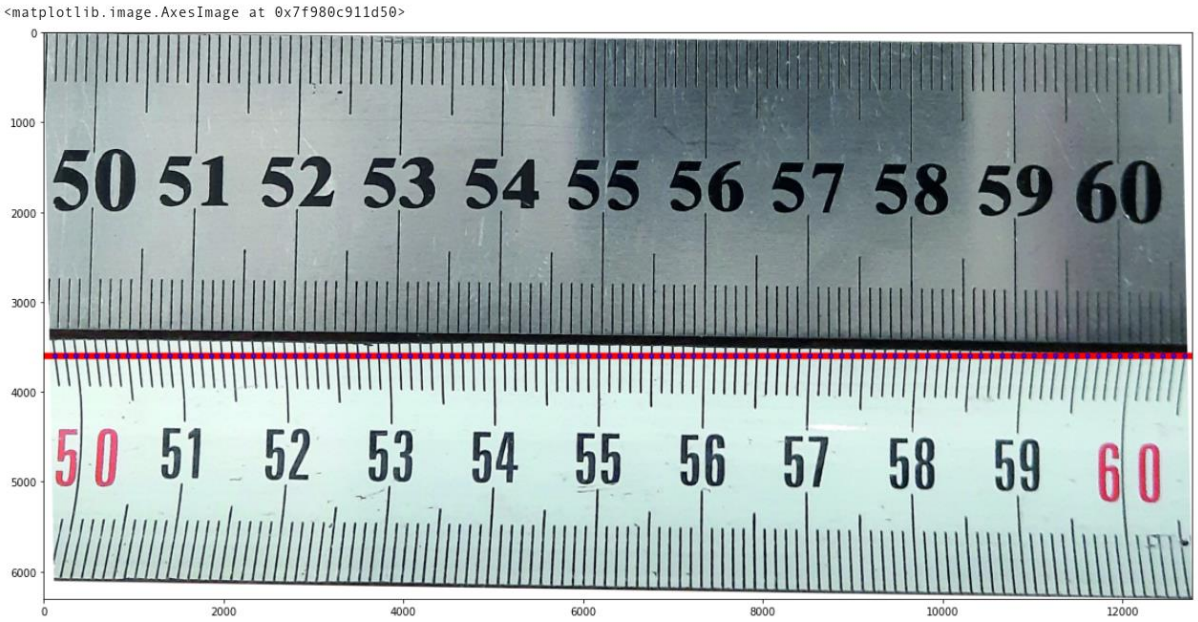

Рисунок 4.9 – Алгоритм типу scan line

Тепер необхідно перевірити точність роботи алгоритму на різних даних. Для цього розмітимо заготовлені зображення та порівняємо результати алгоритму з розміченими даними.

В першу чергу розглянемо зображення лінійки на діапазоні двох сантиметрів. Відповідна розмітка матиме наступний вигляд у пікселях:

```
markers = [
  219,
  486,
  757,
  1025,
  1297,
  1567,
  1843,
  2109,
  2379,
  2647,
  2915,
  3185,
  3459,
  3727,
  3994,
```

```
  4260,
  4533,
  4801,
  5069,
  5339,
  5613,
  5883
]
```

Відповідно знайдені точки перетину скануючої прямої прийняли наступний вид (рис. 4.10).

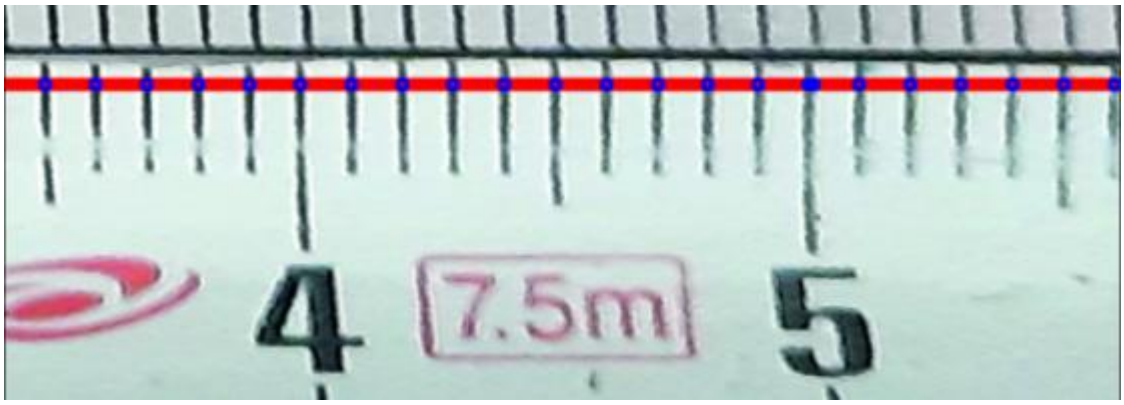

Рисунок 4.10 – Результат скануючої прямої в діапазоні двох сантиметрів

Значення найбільшої похибки зіставило 0.03 мм.

Тепер розглянемо алгоритм на зображенні частини лінійки в діапазоні десяти сантиметрів. Відповідна розмітка матиме вигляд:

```
markers = [
  110,
  225,
  343,
  459,
  574,
  690,
  807,
  926,
  1040,
  1159,
  1272,
  1391,
  1510,
  1623,
  1742,
  1859,
  1973,
```

```
  2090,
  2209,
  2325,
  2441,
  2557,
  2676,
  2793,
  2907,
  3021,
  3144,
  3255,
  3373,
  3492,
  3610,
  3723,
  3840,
  3956,
]
```

Відповідно знайдені точки скануючої прямої прийняли наступний вид (рис. 4.11).

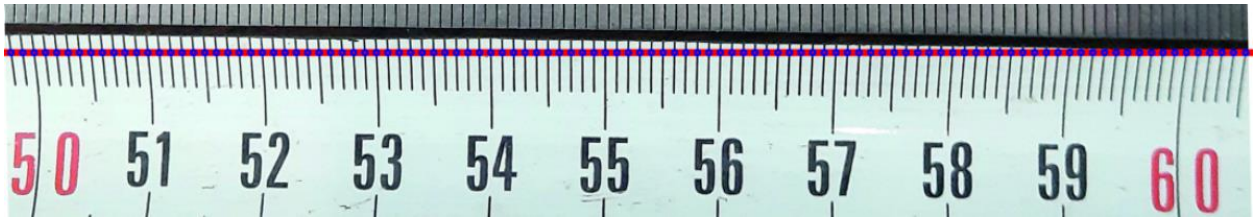

Рисунок 4.11 – Результат скануючої прямої на діапазоні десяти сантиметрів

Значення найбільшої похибки зіставило 0.2 мм. Зазначимо що в даному випадку лінійки не прилягають одна до одної, що призводить до додаткових похибок.

Для тестування ПЗ було заготовлено набір зображень відповідно еталонної лінійки та цільової (до того ж різних форматів, під різними ракурсами і з різним рівнем розмитості).

Розглянемо результати системи у відповідності до якості зображення (під якістю розуміємо чіткість зображення, точність співвідношення позначок цільової та еталонної лінійок). Виділимо певні критерії, за яких можна сказати, що умови для валідації ідеальні: лінійки знаходяться паралельно одна до одної, відсутня відстань між еталонною та цільовою лінійкою, камера знаходиться під кутом 90

градусів, відносно лінійок. Недотримання хоча б одного із цих пунктів призводить до погіршення результату та відповідних похибок.

В першу чергу розглянемо ідеальний варіант і результат роботи системи на ньому (рисунок 4.12).

Як бачимо система доволі чітко визначила позначки еталонної лінійки та встановила позначки цільової лінійки. Зазначимо що самі лінійки знаходяться доволі близько одна до одної, а кут між відповідними позначками наближений до розгорнутого. Також видно, що відповідні позначки між цільовою та еталонною лінійками не відрізняються більше ніж на 1 мм.

Розглянемо наступний тестовий випадок, коли лінійки все ще доволі паралельні одна від одної, але вже між ними більша відстань (рисунок 4.13).

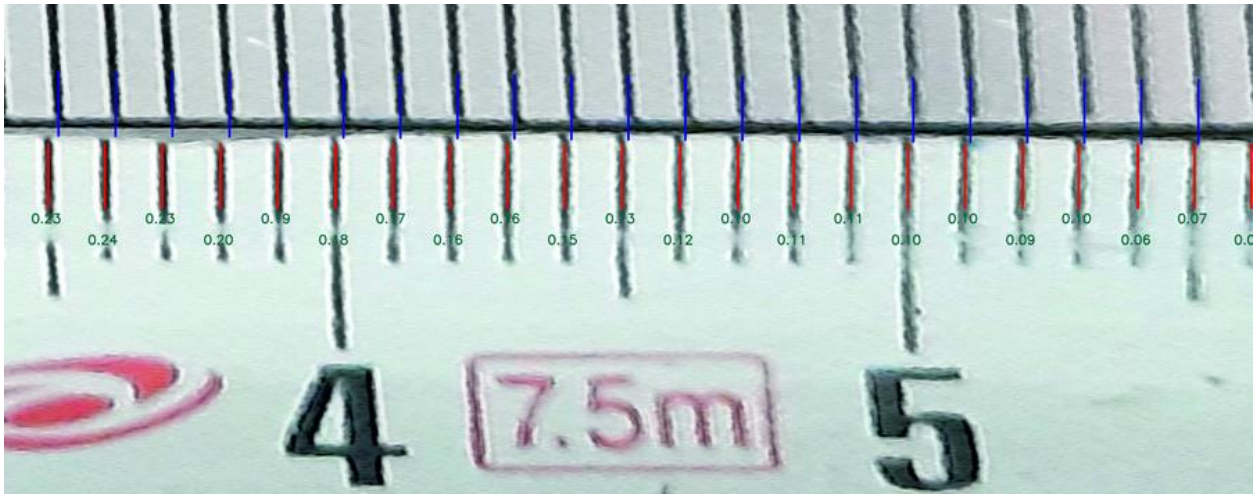

Рисунок 4.12 – Результат валідації системи

Тепер позначки трохи здвинуті, проте система все ще доволі чітко знаходить позначки віртуальної та еталонної лінійок. Зазначимо, що позначки цільової лінійки максимально підведені до віртуальних позначок еталонної лінійки для кращого візуального ефекту порівняння та додатковою перевіркою людиною коректності валідації.

Розглянемо ще один тестовий випадок, коли відповідні позначки на еталонній та цільовій лінійках відрізняються більше ніж на 1 мм (рисунок 4.14).

Як бачимо, система видає на перший погляд дивний результат, проте йому є логічне пояснення.

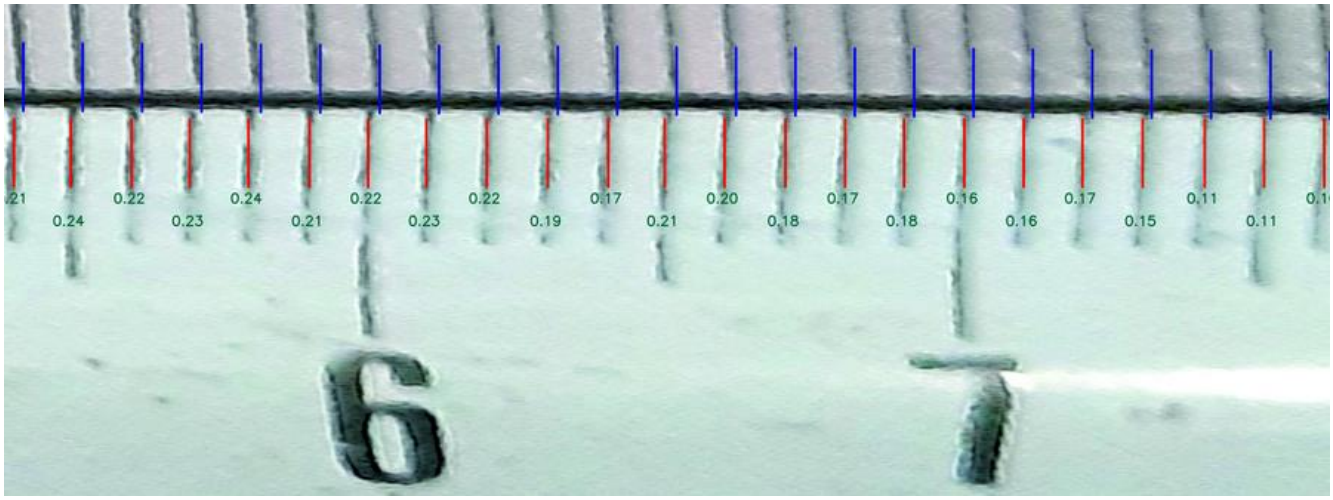

Рисунок 4.13 – Результат валідації з більшою відстанню

Причин такому ефекту дві. Перша, це похибка, що перевищує 1мм. Система розрахована на аналітику невеликих відхилень до 1 мм. В даному випадку стає невідомо, до якої метричної позначки цільової лінійки необхідно віднести позначку еталонної лінійки (алгоритм вибирає позначку до якої найменша відстань). Таким чином позначка буде обрана невірно.

Іншим важливим критерієм слугує відстань між лінійками. Вважається, що лінійки будуть знаходитись якомога ближче одна до одної, адже для порівняння значень лінійки з еталоном (у звичайній перевірці людиною), необхідно якомога ближче їх змістити один до одного.

В результаті подібні випадки, можна звести до нежиттєвих випадків системи, адже людина самостійно без допомоги сторонніх засобів може визначити, що позначки відрізняються більше ніж на 1 мм. Окрім того, прихід до цього можна побачити ще на валідаціях попередніх блоків лінійки.

Також зазначимо, що подібну проблему можна виправити шляхом ідентифікації кожному значенню еталонної лінійки одного значення з цільової лінійки. Проте знову ж таки, якщо позначки відрізняються більше ніж на 0.5 мм,

то система вичислить похибки невірно, адже спирається на підхід найближчих сусідів.

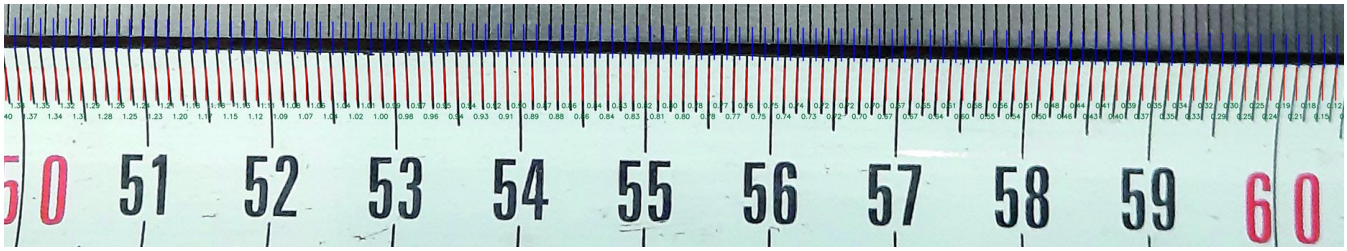

Рисунок 4.14 – Результат валідації з похибкою більше за 1 мм

В якості подальшого покращення системи можна додати спеціалізовані нейронні сітки для аналізу метрологічних засічок під кутом. На даний момент система очікує зображення лінійки, позначки на якій будуть максимально паралельні осі ординат.

ВИСНОВКИ

В ході виконання роботи, був проведений аналіз предметної області метрологічних вимірювань, зокрема перевірка лінійок відносно державних стандартів. На основі отриманої інформації було спроектовано рішення для автоматизації процесів валідації.

Було порівнянно ефективність відносно поставленої задачі алгоритмів для аналізу зображень, а саме алгоритми сегментації, трансформації, пошуку шаблону. На основі отриманих результатів було обрано найефективніше об’єднання цих рішень та запрограмовано у результуючу систему для аналізу металевої лінійки до відповідного стандарту. При цьому зазначимо, що найкращий результат показали поєднані алгоритми рухомого вікна (для трансформації зображення до чорно-білого формату) та алгоритм скануючої прямої (для відтворення позначок, що відповідають міліметровим засічкам лінійки).

Система реалізована через фреймворк Django та з використанням бібліотеки OpenCV. В якості архітектурного рішення реалізовано REST (за принципами RESTful) з використанням протоколу HTTP та форматом даних multipart/formdata. Під час реалізації було також порівнянно ефективність роботи системи з використанням формату Base64. В результаті отримали, що швидкість з використанням Base64 вища при невеликиз зображеннях (на декілька байтів), але для великих файлів куди ефективніше себе показав формат multipart/formdata.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Hramm, O., Bilous, N., Ahekian, I. Configurable Cell Segmentation Solution Using Hough Circles Transform and Watershed Algorithm // Proceedings of the International Conference on Advanced Optoelectronics and Lasers - CAOL, 2019, 2019-September, С. 602-605, 9019493.

2. Малкіна В.М., Білоус Н.В. Методика вимірювання показників вибірки насіння соняшнику на основі класифікації за ознаками геометричних показників // «Системи обробки інформації» №2 (127) 2015, С. 118-120.

3. Shcherbakova, G., Hao-Su, S., Krylov, V., Bilous, N., Antoshchuk, S. Estimation of the duration of RR-intervals of electrocardiograms by mean of multi-start optimization based on wavelet transformation // Proceedings of the 2017 IEEE 9th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications IDAACS 2017, 2017, 2, С. 752–755, 8095190.

4. Shcherbakova, G., Krylov, V., Logvinov, O., Bilous, N. Adjustement of wavelet function parameters for analysis of non-stationary periodic signals with multistart optimization // 2017 4th International Scientific-Practical Conference Problems of Infocommunications Science and Technology, PIC S and T 2017 - Proceedings, 2017, 2018-January, С. 110–112.

5. AO Rakova, NV Bilous. REFERENCE POINTS METHOD FOR HUMAN HEAD MOVEMENTS TRACKING // Radio Electronics, Computer Science, Control, 2020.

6. Залужний Д.В., Білоус Н.В. Алгоритми сегментації зображень лінійки // Тези доповідей дванадцятої міжнародної науково-технічної конференції. 2022. Том 2. Секція 5. С. 170.

7. ДСТУ 427-75 – Державний стандарт щодо металевих лінійок. URL: https://instrumentservis.ua/uploads/gosts/427-75.pdf (дата звернення: 19.03.2022).

8. Специфікація та приклади реалізації Watershed алгоритму з оптимізаціями. URL: http://www.cs.rug.nl/~roe/publications/parwshed.pdf (дата звернення: 19.03.2022).

9. Border following – Topological Structural Analysis of Digitized Binary Images by Border Following. URL: http://pdf.xuebalib.com:1262/xuebalib.com.17233.pdf (дата звернення: 19.03.2022).

10. ARuler – Мобільний додаток з Google Play, що дозволяє віртуальне вимірювання. URL: https://play.google.com/store/apps/details?id=com.grymala.aruler (дата звернення: 19.03.2022).

11. Merlic – Набір бібліотек та конструкторів для автоматичної побудови систем пов’язаних з комп’ютерним зором. URL: https://www.mvtec.com/products/merlic (дата звернення: 19.03.2022).

12. Django – Python фреймворк для побудови серверних рішень. Документація. URL: https://www.djangoproject.com/ (дата звернення: 19.03.2022).

13. OpenCV – Набір бібліотек для обробки зображень. Документація. URL: https://docs.opencv.org/4.x/ (дата звернення: 19.03.2022).

14. Docker – Official page. URL: https://www.docker.com/ (дата звернення: 19.03.2022).

15. Docker Compose – Система одночасного запуску контейнерів. URL: https://docs.docker.com/compose/ (дата звернення: 28.04.2022).

16. Nginx – HTTP сервер. Документація. URL: https://nginx.org/en/ (дата звернення: 19.03.2022).

17. React.js – JavaScript фреймворк для створення single page додатків. URL: https://en.reactjs.org/ (дата звернення: 19.03.2022).

18. Base64 – Формат кодування данних в ascii символи. Документація. URL: https://datatracker.ietf.org/doc/html/rfc4648 (дата звернення: 19.03.2022).

19. HTML encoding standard – Multipart form data як ефективний формат кодування файлів. URL: https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#multipart-form-data (дата звернення: 28.04.2022).

20. RFC 7578 – Специфікація алгоритму multipart form data. URL: https://www.rfc-editor.org/rfc/rfc7578 (дата звернення: 28.04.2022).

21. Git OpenCV – Посилання на код з перетворенням до gray scale формату. URL: https://github.com/egonSchiele/OpenCV/blob/master/modules/imgproc/src/color.cpp#L392 (дата звернення: 30.04.2022).