МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ РАДИОФИЗИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Кафедра информатики и компьютерных технологий

БАРКОВСКИЙ
Ярослав Юрьевич

РАЗРАБОТКА И СРАВНЕНИЕ АЛГОРИТМОВ РАБОТЫ ДАТЧИКА НАПРАВЛЕНИЯ НА СОЛНЦЕ НА ОСНОВЕ КМОП-МАТРИЦЫ

Дипломная работа

Научный руководитель:
старший преподаватель кафедры
ИиКС РФиКТ, С.В. Василенко

Допущен к защите
«___» _____ 2021 г.

Зав. кафедрой информатики и компьютерных систем,
доктор технических наук, профессор Мулярчик С.Г.

Минск, 2021

ОГЛАВЛЕНИЕ

РЕФЕРАТ

Дипломная работа: 73 страницы, 24 рисунка, 3 таблицы, 10 источников, 3 приложения.

АЛГОРИТМЫ, ПОИСК, МАТРИЦА, ПОИСК ЦЕНТР ПЯТНА, СВЕТОЧУВСТВИТЕЛЬНАЯ МАТРИЦА, PYTHON, MATLAB, ДАТЧИК ОРИЕНТАЦИИ ПО СОЛНЦУ, НЕСОРТИРОВАННЫЙ МАССИВ.

Объект исследования – алгоритмы определения центра освещённого пятна на светочувствительной матрице в составе датчика ориентации по Солнцу.

Цель работы – разработка и сравнение вычислительной эффективности алгоритмов по поиску центра светового пятна на изображении со светочувтсвительной матрицы в составе датчика ориентации по Солнцу.

Методы исследования – анализ, моделирование, сравнение.

В результате выполнения работы разработаны и сравнены между собой алгоритмы обнаружения и локализации светового пятна на изображении со светочувствительной матрицы. Эти методы основаны на теории поиска в несортированном массиве. Программная реализация и тестирование алгоритмов были выполнены на языке программирования Python и в системе компьютерной математики Matlab. Работа выполнялась в интересах разработки датчика ориентации по Солнцу на основе светочувствительной матрицы для использования в составе бортовой системы ориентации наноспутника.

РЭФЕРАТ

Дыпломная праца: 73 старонкі, 24 малюнка, 3 табліцы, 10 крыніц, 3 прыкладання.

АЛГАРЫТМЫ, ПОШУК, МАТРЫЦА, ПОШУК ЦЭНТРА ПЛЯМЫ, СВЯТЛОАДЧУВАЛЬНЫЯ МАТРЫЦЫ, PYTHON, MATLAB, МАЛЮНКІ, ДАТЧЫК АРЫЕНТАЦЫІ ПА СОНЦЫ, НЕ САРТАВАНЫ МАСІЎ.

Аб'ект даследавання – алгарытмы вызначэння цэнтра асвялтенай плямы на святлоадчувальнай матрыцы ў складзе датчыка арыентацыі па Сонцы.

Мэта працы – распрацоўка і параўнанне вылічальнай эфектыўнасці алгарытмаў па пошуку цэнтра светлавога плямы на малюнку са светочувтсвительной матрыцы ў складзе датчыка арыентацыі па Сонцы.

Метады даследвання – аналіз, мадэляванне, параўнанне.

У выніку выканання работы распрацаваны і параўнаць паміж сабой алгарытмы выяўлення і лакалізацыі светлавога плямы на малюнку са святлоадчувальнай матрыцы. Гэтыя метады заснаваныя на тэорыі пошуку ў малокомплектных масіве. Праграмная рэалізацыя і тэставанне алгарытмаў былі выкананы на мове праграмавання Python і ў сістэме камп'ютэрнай матэматыкі Matlab. Праца выконвалася ў інтарэсах распрацоўкі датчыка арыентацыі па Сонца на аснове святлоадчувальнай матрыцы для выкарыстання ў складзе бартавы сістэмы арыентацыі нанаспадарожніку.

ABSTRACT

Degree paper: 73 pages, 24 images, 3 tables, 10 sources, 3 apps.

ALGORITHMS, SEARCH, MATRIX, SEARCH SPOT CENTER, SENSITIVE MATRIX, PYTHON, MATLAB, SUN ORIENTATION SENSOR, UNSORTED ARRAY.

Object of research – algorithms for determining the center of the illuminated spot on the photosensitive matrix as part of the solar orientation sensor.

Objective – development and comparison of the computational efficiency of algorithms for finding the center of a light spot in an image from a photosensitive matrix as part of a solar orientation sensor.

Methods of investigation – analysis, modeling, comparison.

As a result of the work, algorithms for detecting and localizing a light spot in an image from a photosensitive matrix were developed and compared. These methods are based on unsorted array search theory. Software implementation and testing of algorithms were performed in the Python programming language and in the Matlab computer mathematics system. The work was carried out in the interests of developing a solar orientation sensor based on a photosensitive matrix for use in the onboard attitude control system of a nanosatellite.

ВВЕДЕНИЕ

В современном мире Солнце является основным навигационным ориентиром, особенно для спутников, поэтому все спутники оснащаются приборами, получившими название солнечных датчиков (СД) или по-другому датчиками солнечной ориентации. Первые упоминания о приборах ориентации по Солнцу в истории космонавтики относятся к первым запускам искусственных спутников Земли. Это оптико-электронные приборы служили для поиска Солнца и формирования электрических сигналов, пропорциональных направлению на энергетический центр диска Солнца в связанной со спутником системе координат. Эти сигналы используются бортовой системой управления либо для разворота спутника в процессе обеспечения его требуемой угловой ориентации на Солнце (например, для ориентации жестко за-крепленных на корпусе спутника солнечных батарей или антенны радиопередатчика), либо для последующего расчета места ориентации спутника в пространстве. В первом случае прибор называется датчиком угловой ориентации спутника, а во втором – датчиком углового положения Солнца. Оба типа приборов делятся по точностным характеристикам разделяются на грубые и точные датчики. Принято считать, что грубые — это датчики, с погрешностью более 5°, умеренной точности, с погрешностью от 0,5 до 5° и точные, с погрешностью менее 30 угл.мин.

Разработано множество вариантов схемотехнического исполнения СД, обеспечивающих как грубое, так и точное измерение направления на Солнце, использующих как статичные, так и подвижные составные элементы. Учитывая современные тенденции развития исследований Земли из космоса, включая задачи дистанционного зондирования Земли, стоит отметить, что все большее значение принимает концепция использования сверхмалых космических аппаратов таких как наноспутники формата CubeSat с линейными размерами порядка 10 см.

Среди датчиков определения положения солнечные датчики представляют собой простую и надежную технологию, используемую во многих космических полетах, позволяющую определять относительное положение тела относительно Солнца, измеряя угол падения солнечного излучения сквозь небольшое отверстие. Общая базовая линия для двухосных цифровых датчиков Солнца состоит в массиве активных пикселей, расположенных за небольшой апертурой: положение пятна, освещенного солнечными лучами, позволяет определить направление Солнца. С появлением более компактных транспортных средств, таких как CubeSat и наноспутники, возникла необходимость ограничить размер и вес таких

устройств: в качестве компромисса это обычно приводит к значительному снижению производительности. В настоящее время стандартные готовые компоненты для CubeSat имеют точность до 0,3 ° с полем обзора от ± 45 ° до ± 90 ° и стоят от нескольких тысяч долларов. В данной работе будет представлен недорогой миниатюрный датчик Солнца на основе коммерческой CMOS-камеры. В устройстве используется прецизионная диафрагма 20 мкм с зазором 7 мм с CMOS. Геометрия и конструкция обеспечивают максимальное разрешение менее 0,05 °, превосходя большинство доступных в настоящее время коммерческих решений. Природа технологии позволяет уменьшить размер, а также ограничить вес.

Задачей данной дипломной работы является создание и сравнение работоспособности алгоритмов поиска центра изображения светового пятна на светочувствительной матрице CMOS кратко описанной выше. Для достижения данной цели промежуточно были сгенерированы изображения падающего луча на светочувствительную матрицу.

ГЛАВА 1.
СВЕТОЧУВСТВИТЕЛЬНЫЕ МАТРИЦЫ.
ТЕОРИЯ АЛГОРИТМОВ ПОИСКА ПЯТНА НА МАТРИЦЕ

Определение: Светочувствительная матрица (фотоматрица) – специализированная цифро-аналоговая или аналоговая интегральная микросхема, состоящая из светочувствительных элементов – фотодиодов. Основная задача фото матриц — это преобразование спроецированного на них оптического изображения в аналоговый электрический сигнал или в поток цифровых данных (если в матрице присутствует АЦП). Широкое применение получили в цифровых фотоаппаратах, видеокамерах и в системах солнечной и астронавигации.

1.1. Общая характеристика светочувствительных матриц

Основными характеристиками фотоматриц являются: отношение сигнал/шум, чувствительность, разрешение, физический размер матрицы, коэффициент заполнения, отношение сторон кадра и пропорции пикселя. Рассмотрим каждую в отдельности:

- Отношение сигнал/шум в общем понятии - безмерная величина, которая отображает отношение между мощностью полезного сигнала и мощностью шума. В случае фотоматриц всякое физическое тело совершает некоторые колебания от своего среднего состояния, в науке это называется флуктуациями. Поэтому и каждое свойство всякого тела тоже изменяется, колеблясь в некоторых пределах. Это справедливо и для такого свойства, как светочувствительность фотоприемника, независимо от того, что собой представляет этот фотоприемник. Следствием этого является то, что некоторая величина не может иметь какого-то конкретного значения, а изменяется в зависимости от обстоятельств. Если, например, рассмотреть такой параметр фотоприемника, как «уровень чёрного», то есть то значение сигнала, которое будет показывать фотодатчик при отсутствии света, то и этот параметр будет каким-то образом флуктуировать.
- Чувствительность фотоматрицы характеризует степень ее реакции на условия окружающего освещения, то есть, чем меньшее количество световой энергии необходимо для получения изображения, тем выше светочувствительность матрицы. Самая частая причина получения

изображений низкого качества – плохая освещенность объекта. Вообще, чем лучше освещенность, тем лучше изображение.

- Разрешение — способность оптического прибора воспроизводить изображение близко расположенных объектов. Разрешение матриц зависит от их типа, площади и плотности фоточувствительных элементов на единицу поверхности. Оно нелинейно зависит от светочувствительности матрицы и от заданного программой уровня шума.
- У современных цифровых фотоматриц разрешающая способность определяется размером пикселя, который варьируется у разных фотоматриц в пределах от 0,0025 мм до 0,0080 мм, а у большинства современных фотоматриц он равен 0,006 мм. Поскольку две точки будут различаться, если между ними находится третья (незасвеченная) точка, то разрешающая способность соответствует расстоянию в два пикселя, то есть:

$$M = \frac{1}{2p} \tag{1.1}$$

где
M – разрешающая способность,
p – размер пикселя.

- Физические размеры фотосенсоров определяются размером отдельных пикселей матрицы, которые в современных фотосенсорах имеют величину 0,005-0,006 мм. Чем крупнее пиксель, тем больше его площадь и количество собираемого им света, поэтому тем выше его светочувствительность и лучше отношение сигнал/шум.
- Коэффициент заполнения из датчика изображения массива представляет собой отношение светочувствительной области пикселя к его общей площади. Для пикселей без микролинз коэффициент заполнения представляет собой отношение площади фотодиода к общей площади пикселя, но использование микролинз увеличивает эффективный коэффициент заполнения, часто почти до 100%, путем схождения света от всей площади пикселя в фотодиод.
- Другим случаем, который уменьшает коэффициент заполнения изображения, является добавление дополнительной памяти рядом с каждым пикселем, чтобы добиться глобального эффекта затвора на датчике CMOS.
- Существует несколько основных отношений сторон кадра которые используются во всех матрицах:
  - Формат кадра 4:3
  - Формат кадра 3:2

- Формат кадра 16:9

- Пропорции пикселя. Выпускаются светочувствительные матрицы с тремя различными пропорциями пикселя:
  - Для видеоаппаратуры выпускаются сенсоры с пропорцией пикселя 4:3
  - Так же некоторые выпускают светочувствительные матрицы с пропорцией пикселя 3:4
  - Светочувствительные матрицы для астрономического, фотографического и рентгенографического оборудования обычно имеют квадратный пиксель, то есть пропорция сторон пикселя 1:1

Так же светочувствительные матрицы различаются методами получения цветных изображений, так как пиксели сами по себе являются “черно-белым”. Рассмотрим 3 основных системы: трехматричная система, матрицы с мозаичным фильтром и матрицы с полноцветными пикселями. В трехматричных системах свет попадает на дихроидные призмы делится на три основных цвета, после чего каждый цвет направляется на отдельную матрицу. Такие системы используются в видеокамерах высокого класса.

- В трехматричных системах свет попадает на дихроидные призмы делится на три основных цвета, после чего каждый цвет направляется на отдельную матрицу. Такие системы используются в видеокамерах высокого класса.

- В матрицах с мозаичными фильтрами пиксели расположены в одной плоскости и каждый пиксель накрыт светофильтром какого-то цвета. А недостающая информация в таких матрицах восстанавливается путем интерполяции. Основные типы расположения светофильтров это RGGB, RGBW, RGEB и CGMY.

- Существуют две технологии матриц с полноцветными пикселями, позволяющими получать с каждого пикселя все три цветовые координаты. Первая это многослойные матрицы используемы фирмой Foeveon, а вторая это полноцветная RGB-матрица от Nikon.
  - Фотоматрица от Foeveon называется Foeveon X3, где X3 означает трехслойность и трехмерность. В ней цветоделение на аддитивные цвета RGB проводится послойно, по толщине полупроводникового материала, с использованием физических свойств кремния.
  - В полноцветных матрицах Nikon лучи RGB предметных точек в каждом пикселе, содержащем одну микролинзу и три фотодиода, проходят через открытую микролинзу и падают на первое дихроичное зеркало. При этом синяя составляющая пропускается первым дихроичным зеркалом на детектор синего, а зелёная и красная составляющие отражаются на второе зеркало. Второе дихроичное

зеркало отражает зелёную составляющую на детектор зелёного, и пропускает красную и инфракрасную составляющие. Третье дихроичное зеркало отражает красную составляющую на детектор и поглощает инфракрасную составляющую. Из-за сложностей в технологии этот патент не нашел достойного применения.

1.1.1. ПЗС - матрицы

ПЗС-матрица – специализированная аналоговая интегральная микросхема, состоящая из светочувствительных фотодиодов, выполненная на основе кремния, использующая технологию ПЗС — приборов с зарядовой связью.

Прибор с зарядовой связью был изобретён в 1969 году Уиллардом Болом и Джорджем Смитом в Лабораториях Белла. Лаборатории работали над видеотелефонией и развитием «полупроводниковой пузырьковой памяти». Приборы с зарядовой связью начали свою жизнь как устройства памяти, в которых можно было только поместить заряд во входной регистр устройства. Однако способность элемента памяти устройства получить заряд благодаря фотоэлектрическому эффекту сделала данное применение ПЗС устройств основным. В 1970 году исследователи Bell Labs научились снимать изображения с помощью простых линейных устройств. Название ПЗС — прибор с зарядовой связью — отражает способ считывания электрического потенциала методом сдвига заряда от элемента к элементу. ПЗС устройство состоит из поликремния, отделённого от кремниевой подложки, у которой при подаче напряжения через поликремневые затворы изменяются электрические потенциалы вблизи электродов.

В основе работы ПЗС лежит явление внутреннего фотоэффекта. Один элемент ПЗС-матрицы формируется тремя или четырьмя электродами. Положительное напряжение на одном из электродов создаёт потенциальную яму, куда устремляются электроны из соседней зоны. Последовательное переключение напряжения на электродах перемещает потенциальную яму, а, следовательно, и находящиеся в ней электроны, в определённом направлении. Заряд, накопленный под одним электродом, в любой момент может быть перенесен под соседний электрод, если его потенциал будет увеличен, в то время как потенциал первого электрода, будет уменьшен. Так происходит перемещение по одной строке матрицы.

Если речь идёт о ПЗС-линейке, то заряд в её единственной строке «перетекает» к выходным каскадам усиления и там преобразуется в уровень напряжения на выходе микросхемы.

У матрицы же, состоящей из многих видеострок, заряд из выходных элементов каждой строки оказывается в ячейке ещё одного сдвигового устройства, устроенного обычно точно таким же образом, но работающего на более высокой частоте сдвига. Для использования ПЗС в качестве светочувствительного устройства часть электродов изготавливается прозрачной.

По принципу перемещения и считывания заряда различают светочувствительные ПЗС-матрицы с межрядным переносом (Interline Transfer), с покадровым переносом (Frame) и с полнокадровым переносом (Full Frame) (см. Рисунок 1.1).

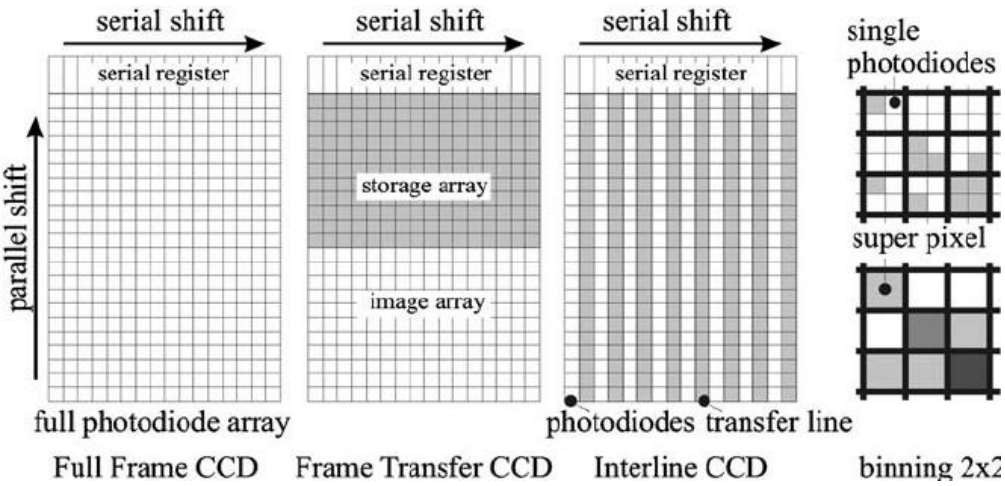

Рисунок 1.1 - – виды матриц по принципу перемещения и считывания заряда

1. Матрицы с полнокадровым переносом (Full Frame);

Данный тип сенсора является наиболее простым с конструктивной точки зрения и именуется ПЗС-матрицей с полнокадровым переносом (full frame CCD matrix). Помимо микросхем «обвязки», такой тип матриц нуждается также в механическом затворе, перекрывающем световой поток после окончания экспонирования. До полного закрытия затвора считывание зарядов начинать нельзя — при рабочем цикле параллельного регистра сдвига к фототоку каждого из его пикселов добавляются лишние электроны, вызванные попаданием фотонов на открытую поверхность ПЗС-матрицы. Данное явление называется «размазыванием» заряда в полнокадровой матрице (full frame matrix smear).

2. Матрицы с покадровым переносом (Frame Transfer);

Существует усовершенствованный вариант полнокадровой матрицы, в котором заряды параллельного регистра не поступают построчно на вход последовательного, а «складируются» в буферном параллельном регистре. Данный регистр расположен под основным параллельным регистром сдвига, фототоки построчно перемещаются в буферный регистр и уже из него поступают на вход последовательного регистра сдвига. Поверхность буферного регистра покрыта непрозрачной (чаще металлической) панелью, а вся система получила название матрицы с буферизацией кадра (frame-transfer CCD).

3. Матрицы с межрядным переносом (Interline Transfer);

Специально для видеотехники был разработан новый тип матриц, в котором интервал между экспонированием был минимизирован не для пары кадров, а для непрерывного потока. Разумеется, для обеспечения этой непрерывности пришлось предусмотреть отказ от механического затвора.
Фактически данная схема, получившая наименование матрицы с буферизацией столбцов (interline CCD matrix), в чём-то сходна с системами с буферизацией кадра — в ней также используется буферный параллельный регистр сдвига, ПЗС-элементы которого скрыты под непрозрачным покрытием. Однако буфер этот не располагается единым блоком под основным параллельным регистром — его столбцы «перетасованы» между столбцами основного регистра. В результате рядом с каждым столбцом основного регистра находится столбец буфера, а сразу же после экспонирования фототоки перемещаются не «сверху вниз», а «слева направо» (или «справа налево») и всего за один рабочий цикл попадают в буферный регистр, целиком и полностью освобождая потенциальные ямы для следующего экспонирования.

Попавшие в буферный регистр заряды в обычном порядке считываются через последовательный регистр сдвига, то есть «сверху вниз». Поскольку сброс фототоков в буферный регистр происходит всего за один цикл, даже при отсутствии механического затвора не наблюдается ничего похожего на «размазывание» заряда в полнокадровой матрице. А вот время экспонирования для каждого кадра в большинстве случаев по продолжительности соответствует интервалу, затрачиваемому на полное считывание буферного параллельного регистра. Благодаря всему этому появляется возможность создать видеосигнал с высокой частотой кадров — не менее 30 кадров секунду.

1.1.2. КМОП – матрицы

КМОП-матрица — светочувствительная матрица, выполненная на основе КМОП-технологии. КМОП- комплементарная структура металл-оксид-полупроводник, набор полупроводниковых технологий построения интегральных микросхем и соответствующая ей схемотехника микросхем. КМОП-матрицах используются полевые транзисторы с изолированным затвором с каналами разной проводимости.

В конце 1960-х гг. многие исследователи отмечали, что структуры КМОП (CMOS) обладают чувствительностью к свету. Однако приборы с зарядовой связью обеспечивали настолько более высокую светочувствительность и качество изображения, что матрицы на технологии КМОП не получили сколько-нибудь заметного развития.

В начале 1990-х характеристики КМОП-матриц, а также технология производства были значительно улучшены. Прогресс в субмикронной фотолитографии позволил применять в КМОП-сенсорах более тонкие соединения. Это привело к увеличению светочувствительности за счёт большего процента облучаемой площади матрицы.

Переворот в технологии КМОП-сенсоров произошёл, когда в лаборатории реактивного движения (Jet Propulsion Laboratory — JPL) NASA успешно реализовали Active Pixel Sensors (APS) — активно-пиксельные датчики (см. Рисунок 1.2). Теоретические исследования были выполнены ещё несколько десятков лет тому назад, но практическое использование активного сенсора отодвинулось до 1993 года. APS добавляет к каждому пикселю транзисторный усилитель для считывания, что даёт возможность преобразовывать заряд в напряжение прямо в пикселе. Это обеспечило также произвольный доступ к фотодетекторам наподобие реализованного в микросхемах ОЗУ.

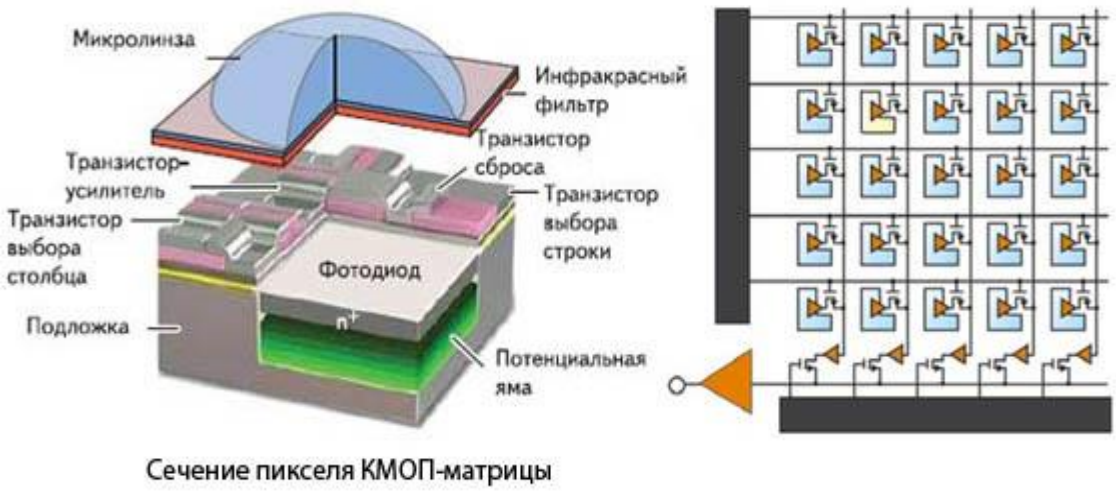

Рисунок 1.2 - Пиксель КМОП-матрицы на основе технологии ASP

В отличие от ПЗС-матриц, КМОП-матрица содержат отдельный транзистор в каждом светочувствительном элементе (пикселе) в результате чего преобразование заряда выполняется непосредственно в пикселе.
Синхронизация работы матрицы осуществляется через адресные шины столбцов и строк.
Благодаря такой схеме появляется возможность считывать заряд сразу из группы пикселей (а не последовательно ячейка за ячейкой, как в ПЗС-матрице) или даже выборочно из отдельных пикселей. В такой матрице отсутствует необходимость в регистрах сдвига столбцов и строк, что намного убыстряет процесс считывания информации с матрицы. Значительно уменьшается и энергопотребление матрицы.

КМОП-матрицы, которые производятся сейчас, делаются на основе датчиков активных пикселей ASP. Датчик активного пикселя (APS) — это датчик изображения, в котором каждая элементарная ячейка датчика пикселей имеет фотоприемник (обычно фотодиод) и один или несколько активных транзисторов. В датчике с активными пикселями металл-оксид-полупроводник (МОП) в качестве усилителей используются полевые МОП-транзисторы (см. Рисунок 1.3). Существуют различные типы APS, в том числе ранняя APS NMOS и гораздо более распространенная дополнительная MOS (CMOS) APS, также известная как датчик CMOS , которая широко используется в технологии цифровых камер, такие как камеры для мобильных телефонов , веб-камеры , большинство современных цифровых карманных камер, большинство цифровых зеркальных фотокамер с одним объективом (DSLR) и беззеркальные камеры со сменными объективами (MILC). КМОП-датчики появились в качестве альтернативы датчикам изображений с зарядовой связью (ПЗС).

Стандартный КМОП APS пиксели сегодня состоит из фотоприемника, плавающего затвора, а также так называемые 4Т клеток, состоящие из четырех КМОП транзистора, включая транзистор передачи, элемент сброса, элемент выбора и транзистор считывания источника-повторителя. Закрепленный фотодиод первоначально использовался в ПЗС с межрядным переносом из-за его низкого темнового тока и хорошего синего отклика, а при соединении с передающим затвором обеспечивает полную передачу заряда от закрепленного фотодиода к плавающему затвору, устраняя запаздывание. Использование внутрипиксельной передачи заряда может обеспечить более низкий уровень шума, позволяя использовать коррелированную двойную выборку (CDS).

Существует два типа структур с активным пиксельным сенсором (APS): боковой APS и вертикальный APS. Боковая конструкция APS описывается как

структура, которая имеет часть области пикселей, используемую для фотодетектирования и хранения сигнала, а другая часть используется для активного транзистора.

Преимущество этого подхода по сравнению с вертикально интегрированной APS состоит в том, что процесс изготовления более прост и

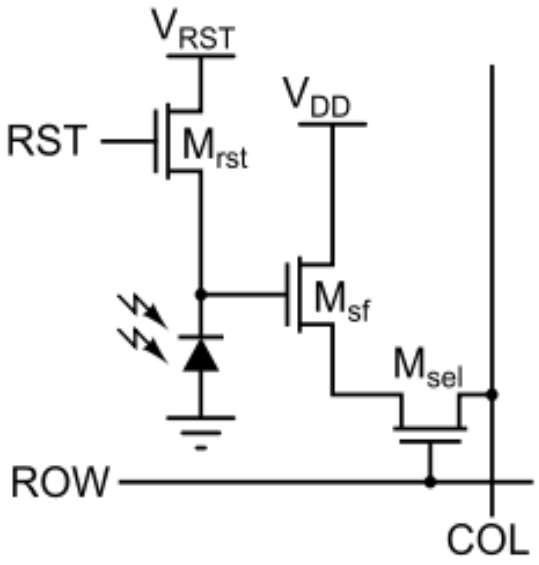

Рисунок 1.3 - Трехтранзисторная схема обработки ASP

в высокой степени совместим с современными процессами CMOS и CCD-устройств.

Вертикальная структура APS — это структура, которая увеличивает коэффициент заполнения (или уменьшает размер пикселя), сохраняя заряд сигнала на выходном транзисторе.

1.1.3. Типы датчиков на основе светочувствительных матриц

В настоящее время для научных задач исследования космоса и в задачах дистанционного зондирования Земли все большее значение принимает концепция использования малых спутников и аппаратов. Массогабаритные требования, предъявляемые к служебным системам, в том числе приборам астронавигации, становятся ключевыми при разработке таких устройств. Особенно это характерно для аппаратуры наноспутников, малых аппаратов с собственной массой порядка десятков килограммов. Применение в таких спутниках традиционных высокоточных систем ориентации массой в килограмм и более становится проблематичным. Значительное, в несколько раз, уменьшение энергопотребления и массогабаритных параметров отдельного датчика ориентации позволяет объединить такие датчики в

систему, что дает возможность по-иному взглянуть на систему ориентации космического аппарата. Установка нескольких малогабаритных датчиков на космический аппарат снимает много проблем с режимом управления и увеличивает надежность всего космического аппарата. Миниатюризация приборов астроориентации обеспечивается, прежде всего, современными технологиями и некоторым снижением требований к точности и чувствительности отдельного датчика.

Исходя из этого факта рассмотрим компактные типы датчиков с небольшой энергозатратностью. Наиболее подходящими под описание конструкции являются датчики направления на Солнце, состоящие из линейчатых ПЗС- матриц и малогабаритных КМОП-матриц.

Первыми рассмотрим конструкции из линейчатых ПЗС-матриц. Они могут быть представлены как 2 перпендикулярно расположенные ПЗС-линейки с 1 отверстием в каждой крышке или 1 ПЗС-линейки с 3 отверстиями в крышке.

1) Конструкция с одной ПЗС-линейкой и тремя отверстиями в крышке.

В оптическом элементе имеются три щели (см. Рисунок 1.4) при этом крайние щели образуют с центральной щелью угол 45°. Поверхность оптического элемента и центральная щель задают внутреннюю систему координат солнечного датчика. Солнечное излучение, проходя через оптический элемент, формирует изображение трех щелей на чувствительной поверхности ПЗС-линейки.

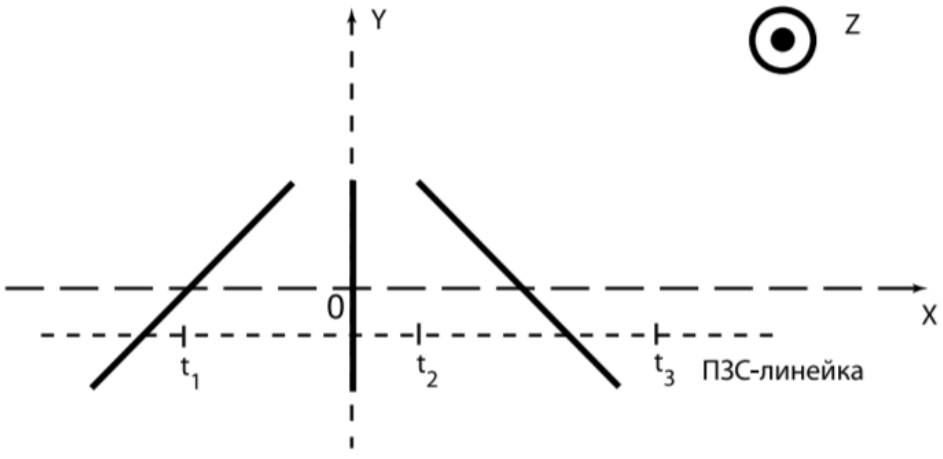

Рисунок 1.4 - Расположение щелей оптического солнечного датчика: t1, t2, t3 — координаты энергетических центров изображений трех щелей кодирующей маски

По положению центральной группы щелей на ПЗС-линейке относительно центрального пиксела линейки можно определить угол Солнца в плоскости матрицы (см. Рисунок 1.5), по расположению изображений крайних щелей относительно изображения центральной щели можно определить угол Солнца в плоскости матрицы. В принципе, для определения направления на Солнце достаточно изображений щелей одной крайней и центральной группы, но для увеличения угла поля зрения используют три щели.

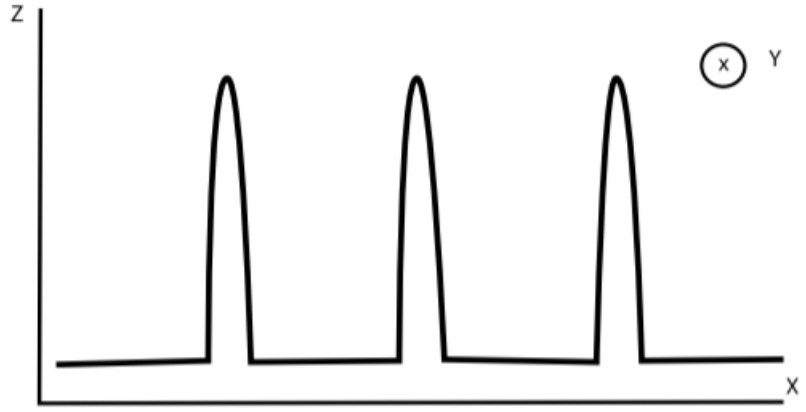

Рисунок 1.5 – Изображение щелей на ПЗС линейке

2) Две ПЗС линейки, расположенные перпендикулярно друг другу.

Каждая ПЗС линейка вычисляет угол падения лучей света относительно одной оси. Происходит это путем прохождения света через щель в крышечке, которой накрыты матрицы, и попаданием его на активную часть светочувствительной матрицы. После чего они детектируются и просчитывается их местоположение на матрице. Затем высчитывается угол падения солнечных лучей и с помощью двух таких углов в двух перпендикулярных плоскостях мы высчитываем угол наклона Солнца относительно поверхности светочувствительной матрицы. Угол обзора в таком случае составляет порядка ±60°.

Теперь рассмотрим КМОП-матрицы. Существует два основных типа конструкций — это две линейные КМОП-матрица с одним отверстием в каждой крышечке и прямоугольная КМОП-матрица так же с одним отверстием в крышечке.

1) В случае, когда мы используем две линейки КМОП-матриц они располагаются перпендикулярно друг другу (см. Рисунок 1.6). Над каждой матрицей есть крышечка с продольным отверстием, через которое попадает солнечный свет. Тем самым каждая при вычислении каждая линейка даст результат по одной оси. После этого остается высчитать угол наклона

относительно Солнца основываясь на два угла в двух плоскостях. Средний угол обзора, по одной оси которого можно добиться это ±45°, однако это еще зависит от размеров матрицы и пикселей.

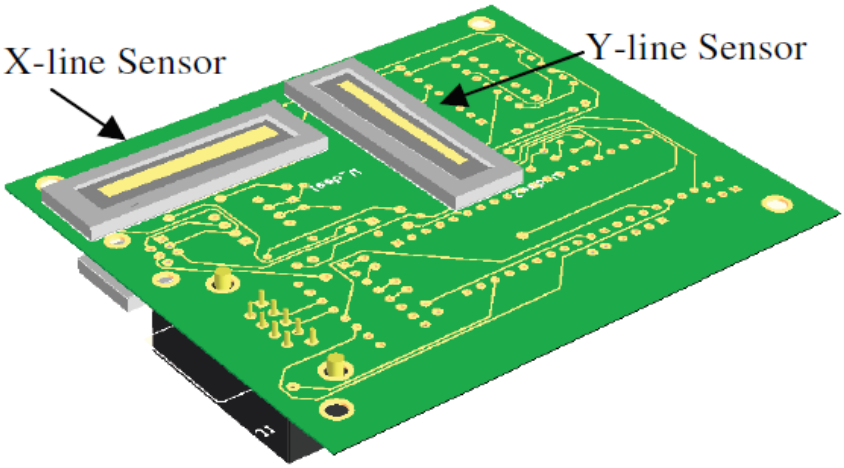

Рисунок 1.6 - Расположение двух КМОП-линеек на плате

2) Когда у нас одна КМОП-матрица она должна воспринимать световые лучи сразу по двум осям. Отсюда можно сделать вывод, что чем ближе форма матрицы к квадрату, тем более равные углы обзора по двум координатам она будет иметь. Матрица накрывается крышечкой с небольшим отверстием, через которую на нее будет попадать солнечный свет (см. Рисунок 1.7). Солнечные лучи проходят через отверстие и обнаруживаются двумерным массивом КМОП. После чего он детектирует и запоминает куда упал свет. Пиксельные позиции вычисляются, а затем устанавливается их положение по двум осям. И исходя из этих данных рассчитывается угол падения солнечного луча. В конструкциях такого типа можно легко достичь угла обзора в ±75°.

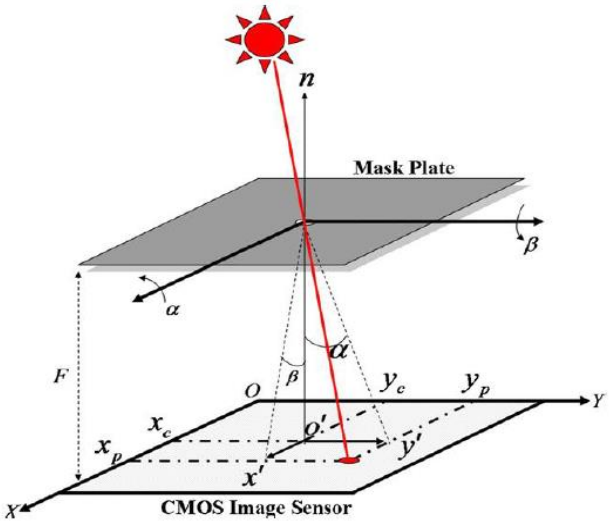

Рисунок 1.7 - Принцип работы КМОП-матрицы для датчика направления на солнце

Рассмотрев эти конструкции, мы можем сделать вывод, что структура, состоящая из одной КМОП-матрицы с одним отверстием в крышечке, выдает наибольший угол обзора

1.2. Теория алгоритмов поиска пятна на матрице

Основная задача по поиску пятна на светочувствительной матрице может быть сведена к более простым задачам. В случае получения изображения КМОП-матрицей изображение конвертируется в двумерных массив пикселей размерами КМОП-матрицы.

Для того, чтобы симулировать поведение КМОП-матрицы в плане получения изображений я создал алгоритм для генерации изображений с различными положениями и видами пятен. Так как КМОП-матрица будет передавать изображение в микроконтроллер в виде массивов пикселей мною были использованы библиотеки и модули проводящие аналогичные операции, то есть чтение изображений и их перевод в массивы пикселей.

Так как полученные массивы пикселей по своей сути являются неотсортированными массивами, то самое надёжное и лучшее решение данной проблемы это последовательное прохождение по каждому элементу массивов, что в итоге даёт нам линейное время поиска пятна. В наихудших случаях, например, когда пятно находится в самом дальнем углу светочувствительной матрицы размерами 752x752 пикселей либо вообще

отсутствует на матрице количество итераций для прохождения по всем массивам займёт O (752x752), что подразумевает под собой 565504 операций сравнения цвета пикселя, инкрементирования итераторов и т.д. Что значительно будет замедлять работоспособность алгоритма и буде влиять на качественную работу микроконтроллера.

Для того, чтобы ускорить поиск центра пятна задача разбивается на три подзадачи:

- Поиск пятна на матрице;
- Локализация пятна;
- Поиск центра пятна;

В ходе данной работы будут разработаны и протестированы алгоритмы поиска пятна, которые будут работать быстрее, чем последовательный поиск в двумерной матрице полученной из КМОП-матрицы.

ГЛАВА 2.
ГЕНЕРАЦИЯ ИЗОБРАЖЕНИЙ НА МАТРИЦЕ

2.1. Информация о используемой матрице

В данной работе использовалась модель КМОП-матрицы MT9V032, позволяющая получить изображение падающего на него луча. Общий вид КМОП матрицы представлен ниже (см. Рисунок 2.1)

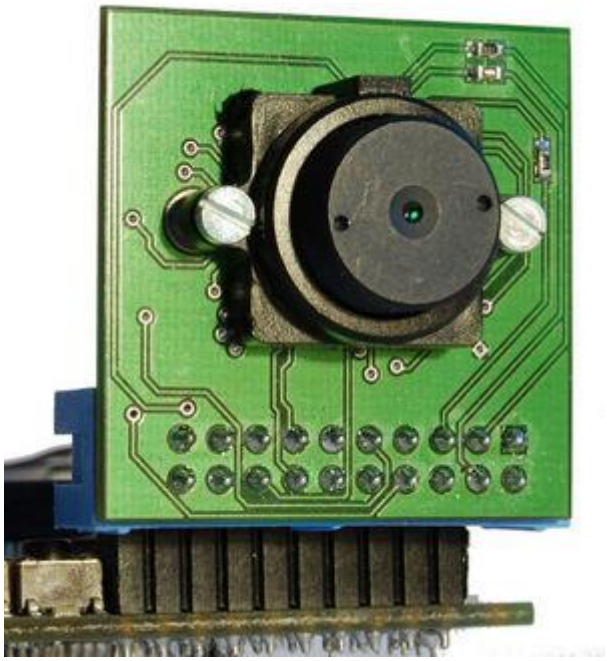

Рисунок 2.1 – Вид КМОП матрицы MT9V032

Характеристики используемого датчика:

- Оптический формат: 1/3 дюйма;
- Активный размер тепловизора: 4.51мм (В) × 2.88мм (Ш);
- Диагональ: 5.5мм;
- Активные пиксели: 752H × 480V;
- Массив цветных пикселей монохромный или цветной RGB;
- Максимальная скорость передачи данных: 26.6 MPS / 26.6 MHz;
- Разрешение: 752 × 480;
- Частота кадров в секунду (при полном разрешении): 60;
- Чувствительность 4.8 V/lux-sec (550nm);

- Динамический диапазон > 55дБ; > 80 дБ – 100 дБ в режиме HDR;
- Напряжение питания 3,3 В ± 0,3 В;
- Рабочая температура от -30 ° С до + 70 ° С;
- Потребляемая мощность <320 мВт при максимальной скорости передачи данных;
- 100 Вт в режиме ожидания.

Особенности матрицы:

- Формат массива: Wide-VGA, активный 752H x 480V (360,960 пикселей)
- Пиксели фотодиодов с глобальным затвором; Одновременная интеграция и считывание
- Монохромный или цветной: улучшенные характеристики в ближнем ИК-диапазоне для использования с невидимой подсветкой в ближнем ИК-диапазоне
- Режимы считывания: прогрессивный или чересстрочный
- Эффективность затвора:> 99%
- Простой двухпроводной последовательный интерфейс
- Возможность блокировки регистра
- Размер окна: программируется пользователем на любой меньший формат (QVGA, CIF, QCIF и т. Д.). Скорость передачи данных может поддерживаться независимо от размер окна
- АЦП: на кристалле, 10-битный параллельный столбец (опция для работы в режиме компандирования с 12-битного до 10-битного);
- Автоматическое управление: автоматическая регулировка экспозиции (AEC) и автоматическое усиление; Контроль (AGC); Переменный региональный и переменный вес AEC / AGC

2.2. Теория генерации изображения

Как видно на рисунке падения луча на матрицу (см. **Ошибка! Источник ссылки не найден.**), устройство состоит из двух частей: активного датчика КМОП и маски. Маска представляет собой круглое отверстие, через которое может фильтровать солнце: активный пиксель (x, y) (см. Рисунок 2.2).

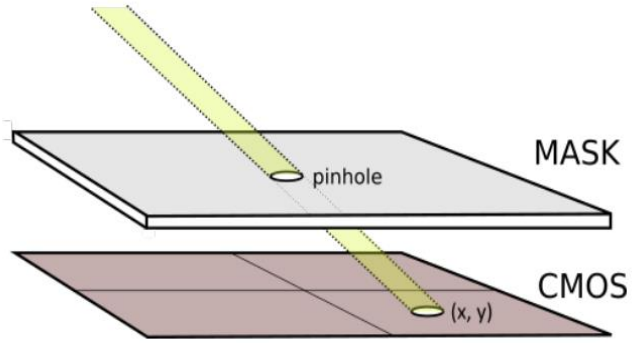

Рисунок 2.2 – Падение луча на матрицу

Датчик используется как детектор светового пятна. Зная координаты точки (x, y) на КМОП можно использовать простую тригонометрию для определения азимутального и зенитного углов. (см. Рисунок 2.3).

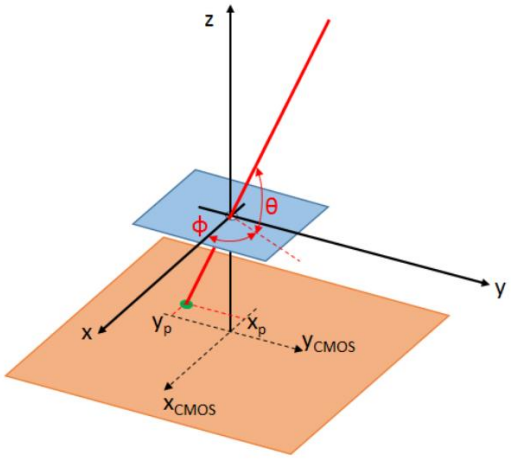

Рисунок 2.3 - Базовая геометрия датчика: координаты пятна определяются азимутальным и зенитным углами

2.3. Геометрическая модель и моделирование

Геометрическая модель предлагаемого датчика представлена на рисунке 2.3 выше, с дополнением, что синяя плоскость – маска, оранжевая – КМОП матрица. Как упоминалось ранее, датчик измеряет направление вектора

солнечных лучей, т.е. относительное положение солнца в поле зрения. Эта информация может быть представлена, как и азимутальным углом, так и зенитным или же связанным вектором:

$$v = (\cos\Phi\cos\theta, \sin\Phi\cos\theta, \sin\theta), \quad (2.1)$$

где:
Ф– азимутальный угол;
θ– зенитный угол.
Но вся эта информация может быть получена, зная положение светового пятна (x, y), поскольку расстояние h между маской и КПОМ мы знаем:

$$v = \frac{(x,y,h)}{\sqrt{(x^2+y^2+x^2}}, \quad (2.2)$$

где:
x, y – координаты пятна на датчике,
h – высота маски над датчиком.

Эта формулировка не включает никаких тригонометрических функций, а единственное и реальное решение. существует для любого положения светового пятна на КМОП.

Зная размер КМОП и расстояние установки маски, можно определить теоретическое поле зрения датчика, учитывая идеальную маску с незначительной толщиной и отсутствием дифракция. Для датчика, описанного в этой работе, два угла поля зрения θ1 и θ2 представленные на рисунке 11 составляют соответственно 64,1 ° и 49,0 °. (см. Рисунок 2.4).

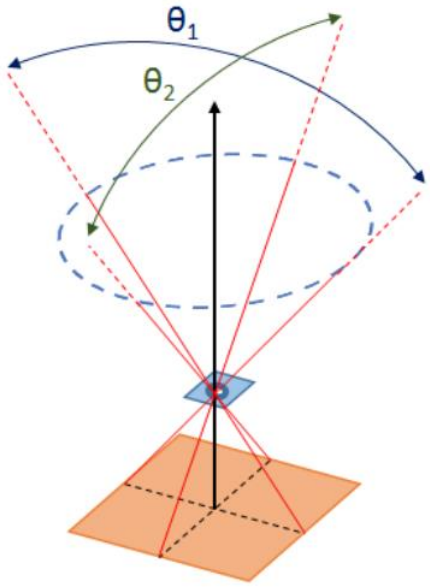

Рисунок 2.4 - Поле зрения датчика

2.4. Моделирование падающего луча

Вышеупомянутое геометрическое описание позволило разработать три различные модели возрастающей сложности, анализируя реакцию датчика солнца на поступающее излучение, как показано на рисунке 2.5 (см. Рисунок 2.5). В идеальном случае маски без толщины проецируемое световое пятно имеет точный размер и форму отверстия маски; Измеряя центр светового пятна, можно напрямую рассчитать вектор направления Солнца.

Но, пока толщиной маски можно пренебречь, такое поведение солнечного луча возможно. Маска в данном устройстве, имеет толщину, сопоставимую с диаметром отверстия, поэтому часть входящего излучения задерживается границами маски, изменяя форму кругового пятна, что можно заметить на рисунке 2.6 (см. Рисунок 2.6) при θ1 = 90, Ф1 = 90, без обрезания краёв пятна, и при θ2 = 78, Ф2 = 78. На рисунках видно, что при изменении задаваемых углов, форма круга изменятся к более эллипсоидной и смещается от центра.

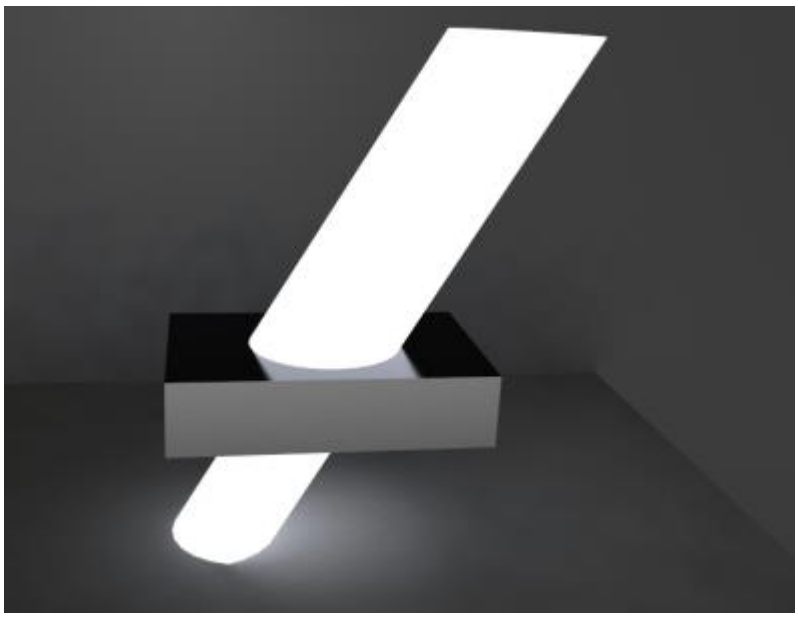

Рисунок 2.5 – Влияние толщины маски на проецируемое пятно света

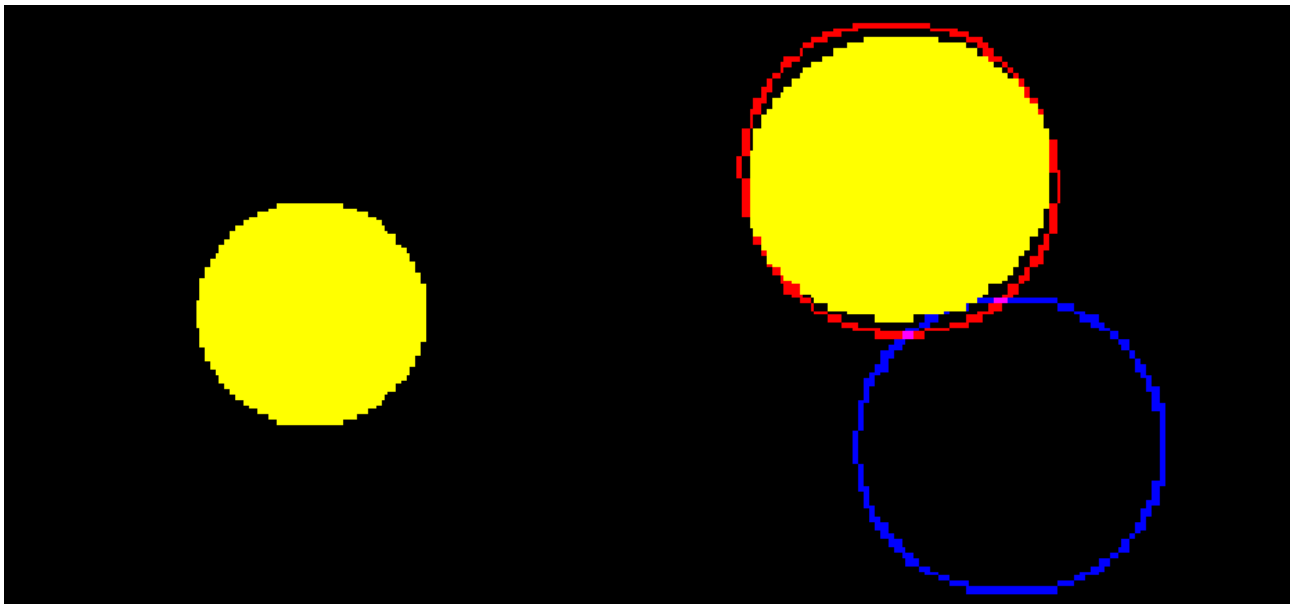

Рисунок 2.6 - Световые пятна при θ1 = 90, Ф1 = 90 (слева) и θ2 = 78, Ф2 = 78 (справа). Красные границы обозначают исходную форму пятна, а синий - позицию

Так же следует отметить, что обе модели выше были разработаны с использованием идеального источника света, то есть точечный источник с параллельными падающими лучами.

Программная реализация отображения пятна на матрице предназначена для оценки перевода и расчёта правильной ориентации пятна. Это моделирование

можно провести, отметив, что геометрия задачи допускает осесимметричные упрощения. Кусочное уравнение, описывающее форму проецируемого пятна в полярных координатах (далее выраженных в терминах азимута и зенита), будет иметь следующий вид:

$$f(\phi,\theta)\begin{cases} r(\theta)-2r(\theta)(x_0\cos\phi+y_0\sin\phi)+{x_0}^2+{y_0}^2=d,\ R_{c0}-\frac{d}{2}<r<R_{c0}+\frac{\Delta C}{2} \\ r(\theta)-2r(\theta)(x_1\cos\phi+y_1\sin\phi)+{x_1}^2+{y_1}^2=d,\ R_{c1}-\frac{\Delta C}{2}<r<R_{c1}+\frac{d}{2} \end{cases}, \quad (2.3)$$

$$r(\theta)=(t+h)*\tan(\frac{\pi}{2}-\theta) \quad (2.4)$$

$$\Delta C=h*\tan(\frac{\pi}{2}-\theta) \quad (2.5)$$

$$\begin{cases} R_{c0}=\sqrt{{x_0}^2+{y_0}^2} \\ R_{c1}=\sqrt{{x_1}^2+{y_1}^2} \end{cases} \quad (2.6)$$

$$\begin{cases} x_1=\Delta C*\cos(\phi+\pi) \\ y_1=\Delta C*sin(\phi+\pi) \end{cases} \quad (2.7)$$

$$\begin{cases} x_0=\Delta(t+h)*\tan\left(\frac{\pi}{2}-\theta\right)*\cos(\phi+\pi) \\ y_0=(t+h)*\tan(\frac{\pi}{2}-\theta)*sin(\phi+\pi) \end{cases} \quad (2.8)$$

где:
d – диаметр отверстия в маске;
t – толщина отверстия;
h –высота отверстия над матрицей;
x_1, y_1, x_0, y_0 – координаты центров двух окружностей;
формирующих финальное изображение пятна;
ΔC – расстояние между центрами окружностей.

На рисунке 2.7 (см. Рисунок 2.7) можно увидеть схематическое представление изменённого солнечного пятна на матрице.

Листинг программы генерации изображений на матрице на языке Matlab приведён в ПРИЛОЖЕНИЕ А

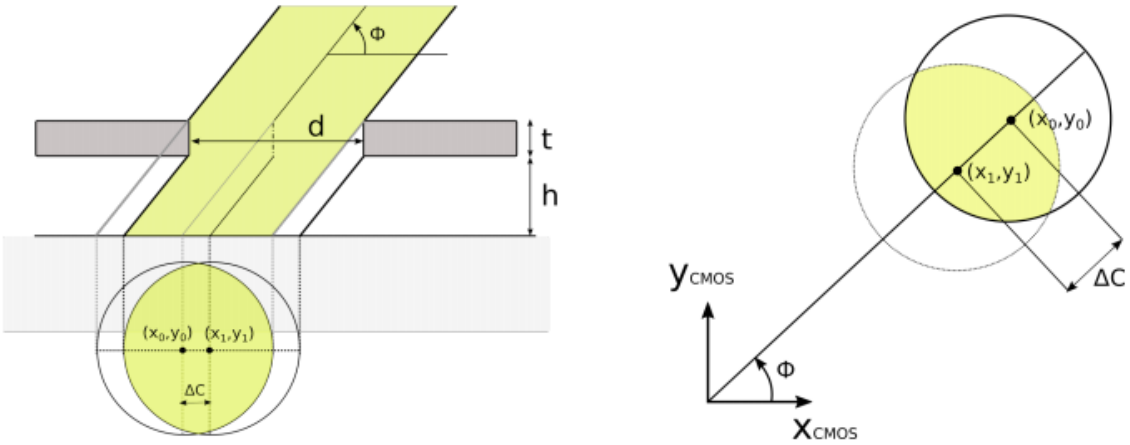

Рисунок 2.7 – Схематическое представление влияния толщины отверстия на изображение

ГЛАВА 3.
ОБРАБОТКА ИЗОБРАЖЕНИЙ

В данной главе будет рассмотрен процесс поиска пятна на матрице несколькими методами и их центра, сравнение их эффективности.

Процесс нахождения углов падения луча состоит из нескольких частей:

1. Поиск пятна на матрице;
2. Локализация пятна на матрице;
3. Поиск центра пятна;
4. Вычисление азимутального и зенитного углов.

Алгоритмы создавались и тестировались на двух языках программирования, первый – Matlab, второй – Python.

Выбор второго языка программирования стоял между языками, на которых можно программировать микроконтроллеры, такие языки как: C, Python, JavaScript. Выбор пал на язык программирования Python, т.к. этот язык имеет ряд преимуществ:

- простой и понятный синтаксис;
- язык python – логичный, лаконичный и понятный;
- имеет сильное комьюнити – можно найти уже готовые решения каких-либо задач. Имеет множество библиотек написанных на C;
- способен запускаться на микроконтроллерах;

3.1. Описание алгоритмов поиска пятна на матрице

Первым делом после прочтения картинки проводилась её бинаризация для того, чтобы убрать световые пятна отражений или другие нежелательные эффекты.

Сам процесс поиска пятна является самой затратной частью алгоритма по поиску центра пятна на матрице, о чём уже говорилось выше. Для того чтобы как-либо ускорить время поиска пятна были разработаны и протестированы несколько алгоритмов:

1. Brute spot – долгий перебор всех пикселей матрицы, начиная с верхнего левого, и сравнение цвета каждого пикселя, пока не встретится пиксель светового пятна. Как работает алгоритм показано на рисунке ниже (см. Рисунок 3.1)

Рисунок 3.1 – Поиск пятна алгоритмом Brute spot

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ Б п. Б.1. Brute spot

2. Rand spot – алгоритм поиска со случайным выбором пикселя на матрице. Выбор случайного пикселя может производиться до того момента, пока выбранный пиксель не окажется “белым” (т.е. частью светового пятна), либо пока не останется пикселей для случайного выбора, т.к. после каждой итерации алгоритма количество возможных значений пикселей уменьшается. Но во время обработки изображений количество итераций данного алгоритма было ограничено до четверти размера изображения и половины размера изображения на языках Matlab и Python соответственно, для ускорения обработки изображения. Результат работы алгоритма можно увидеть на рисунке ниже (см. Рисунок 3.2) (размеры изображения были обрезаны и увеличены для лучшей видимости), где точки красного цвета обозначают пиксели, в которые попал и обработал алгоритм, желтый цвет обозначает световое пятно, красная точна нём – первый обнаруженный пиксель светового пятна;

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ Б п. Б.2. Rand spot

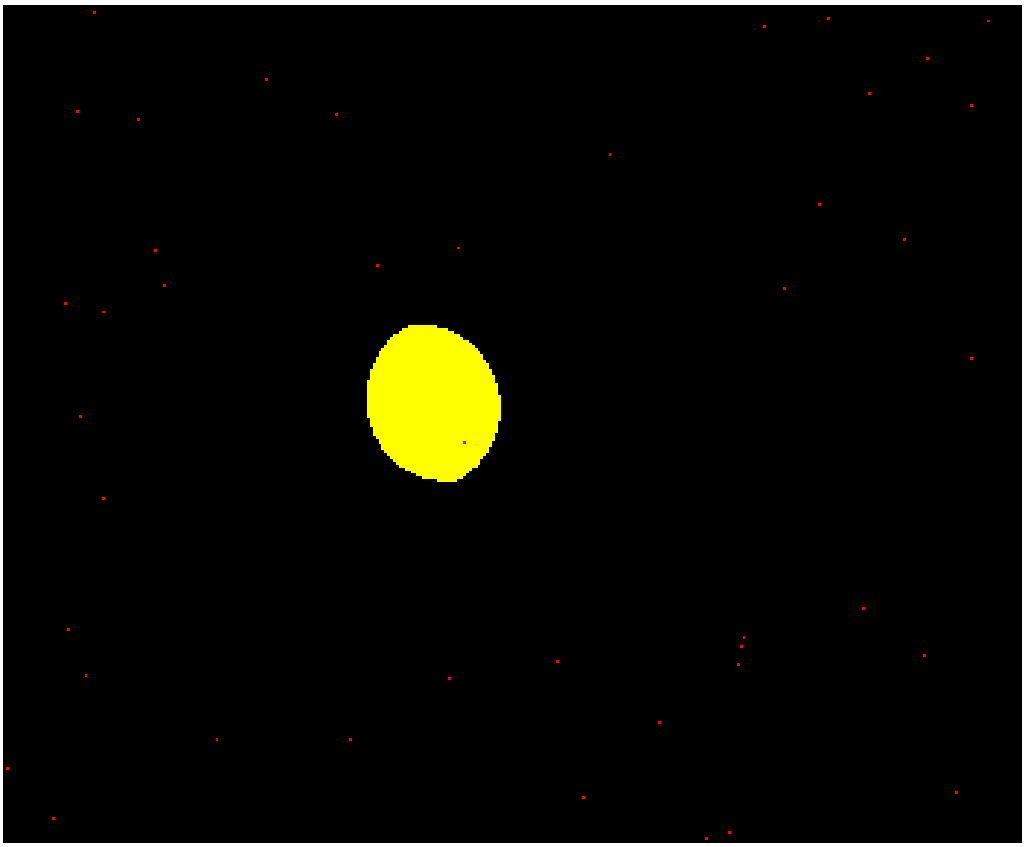

Рисунок 3.2 - Поиск пятна алгоритмом Rand spot

3. Breadth spot – поиск в ширину, является одним из методов обхода графов, так как двумерную матрицу можно представить в виде неориентированного графа был использован для поиска пятна. Для корректной работы данного алгоритма необходимо иметь структуру данных очередь или связный список, встроенные в язык или реализованные самостоятельно. Данный алгоритм не может быть корректно реализован на языке Matlab. Результат работы алгоритма можно увидеть на рисунке ниже (см. Рисунок 3.3), где зеленый цвет обозначает пиксели, обработанные алгоритмом, желтый цвет обозначает сгенерированное световое пятно на изображении;

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ Б п. Б.3. Breadth spot

4. Double spot – является вариацией алгоритма Brute force, но основное отличие заключается в том, что проверялся не каждый пиксель в матрице, а каждый второй, для того чтобы ускорить время поиска пятна в 2 раза. Результат работы алгоритма можно увидеть на рисунке 3.4 ниже (см. Рисунок 3.4), где красный цвет обозначает пиксели, обработанные алгоритмом, желтый цвет обозначает сгенерированное световое пятно на

изображении, синий цвет обозначает пиксель светового пятна, обнаруженный алгоритмом;

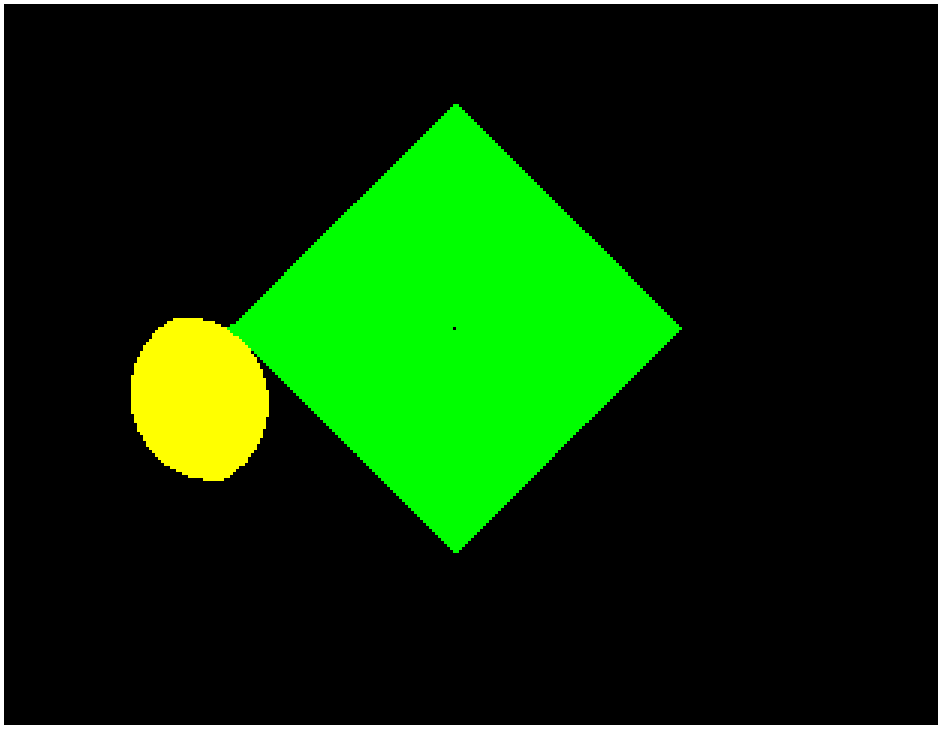

Рисунок 3.3 – Поиск пятна алгоритмом Breadth spot

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ Б п. Б.4. Double spot

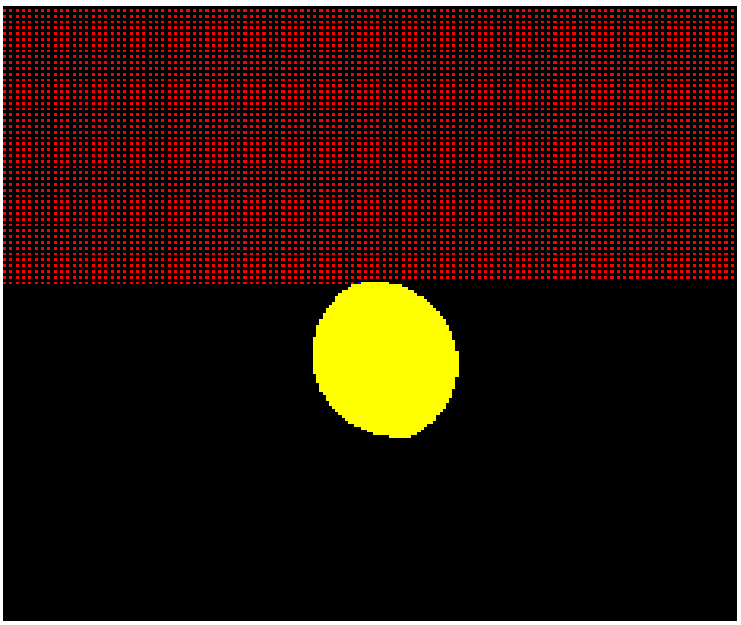

Рисунок 3.4 - Поиск пятна алгоритмом Double spot

5. Round spot – очередная вариация алгоритма Brute force, но основное отличие заключается в том, что поиск пятна начинается из центра матрицы и начинается разрастаться до границ матрицы. Аналогичен алгоритму Breadth spot, но имеет другую реализацию, без очереди или связного списка. Результат работы алгоритма можно увидеть на рисунке 3.5 ниже (см. Рисунок 3.5), где зеленый и синий цвет обозначает нечетные и четные пиксели, соответственно обработанные алгоритмом, желтый цвет обозначает сгенерированное световое пятно на изображении, красный цвет обозначает пиксель светового пятна, обнаруженный алгоритмом;

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ Б п. Б.5. Round spot

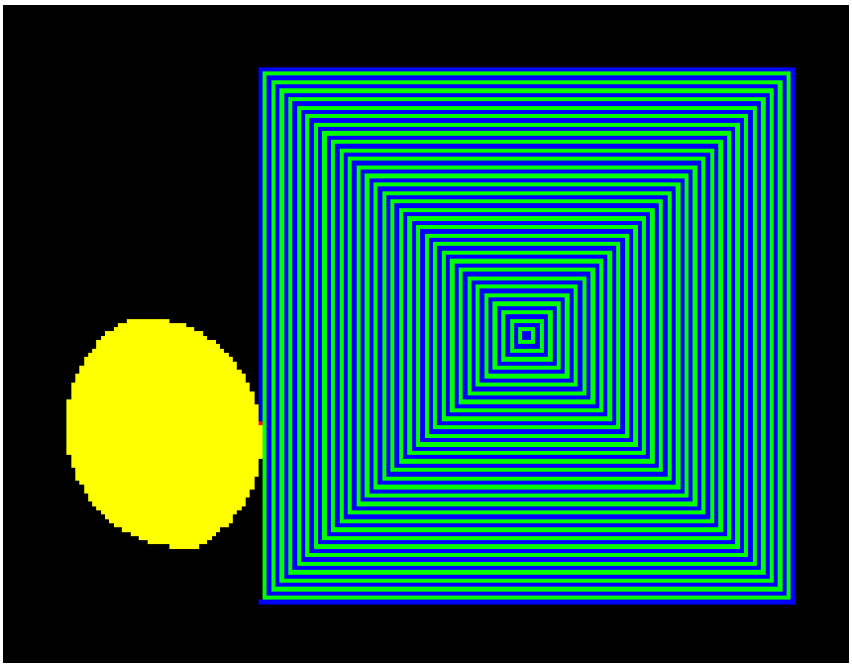

Рисунок 3.5 - Поиск пятна алгоритмом Round spot

6. Round double spot – является комбинацией алгоритмов “Double spot” и “Round spot”, начинает поиск из центра матрицы при этом проверяя каждый второй пиксель значительно ускоряя работоспособность. Результат работы алгоритма можно увидеть на рисунке 3.6 ниже (см. Рисунок 3.6), где зеленый и синий цвет обозначает нечетные и четные пиксели, соответственно обработанные алгоритмом, желтый цвет обозначает сгенерированное световое пятно на изображении, красный цвет обозначает пиксель светового пятна, обнаруженный алгоритмом;

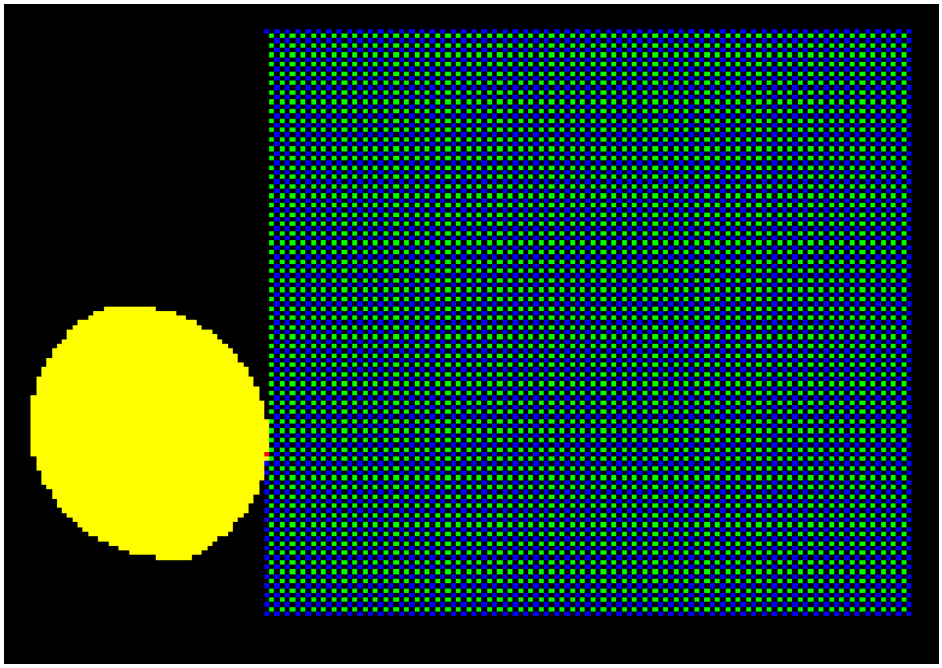

Рисунок 3.6 – Поиск пятна алгоритмом Round double spot

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ Б п. Б.6. Round double spot

3.2. Описание алгоритмов локализации пятна

Процесс локализации пятна необходим для того, чтобы получить все пиксели пятна на матрице используя за начальное значение пиксель, полученный на этапе поиска пятна на матрице (поиска первого пикселя пятна).

На данном этапе использовалось несколько алгоритмов:

1. Brute force – данный алгоритм является продолжением аналогичного алгоритма поиска пятна, но после нахождения “первого” пикселя светового пятна алгоритм поиска не останавливает своё выполнение и продолжает идти по матрице сравнивая значение в каждом пикселе матрицы. Но после нахождения “первого” пикселя светового пятна ускоряется, то есть после того, как он обнаружит первый пиксель пятна в строке матрицы алгоритм запомнит это и, если после этого алгоритм опять встретит пиксель не являющийся частью пятна, он перейдёт на следующую строку, если за всю строку он не найдёт ни одного пикселя пятна, то алгоритм остановит своё выполнение, т.к. поймёт, что уже обработал всё пятно, таким образом ускоряя значительно свою работу. Как это выглядит можно увидеть на рисунке ниже

(см. Рисунок 3.7). Закрашенная желтым часть – пиксели, которые обработал алгоритм и не нашёл там светового пятна, красный цвет обозначает световое пятно, которое обнаружил алгоритм, черный цвет – все необработанные пиксели.

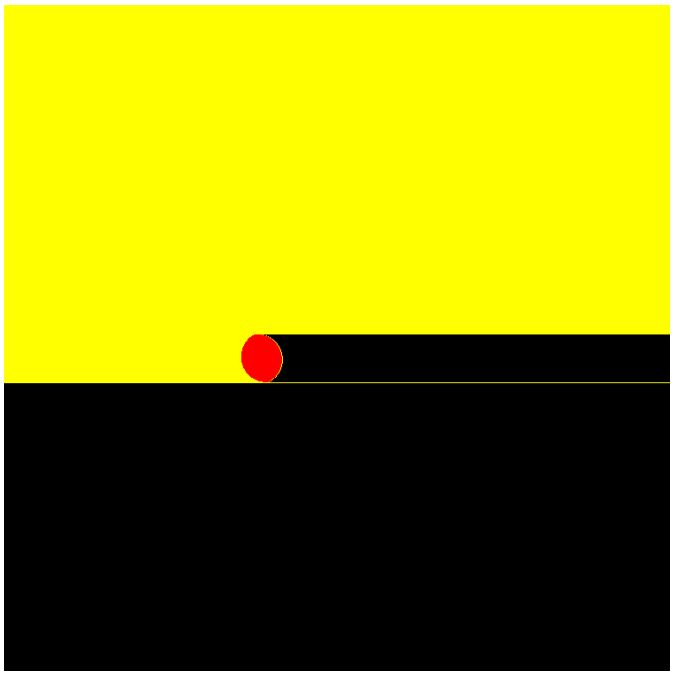

Рисунок 3.7 – поиск и локализация пятна алгоритмом Brute force

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ В п. В.1. Brute force

2. Quadro search – по своей сути является обычным алгоритмом последовательного перебора пикселей, т.е. аналогом Brute force, но от Brute force алгоритма он отличается количеством обрабатываемых пикселей. Так как данный алгоритм применяется уже после того, как пятно обнаружили, алгоритму достаточно изучить пиксели в квадрате размерами [d x d], где d – диаметр отверстия маски. Это обусловлено тем, что при прямом падении луча, $\theta = 0$, $\Phi = 0$, изображение пятна не искажается т.к. не перекрывается стенками отверстия, и мы получаем пятно максимально возможного размера. Т.к. пятна по размерам больше, чем этого быть не может достаточно изучить 4 квадрата размерами [d/2 x d/2], размеры изучаемого квадрата продемонстрированы на рисунке ниже (см. Рисунок 3.8), где зеленый цвет обозначает границы изучаемого квадрата.

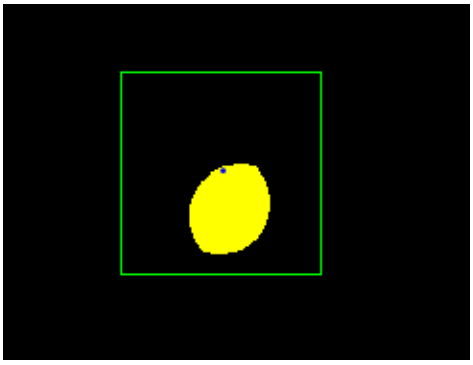

Рисунок 3.8 – Локализация пятна в квадрате размерами [d x d]

Во время изучения алгоритмом данного квадрата, аналогично методу Brute force, обработка квадрата заканчивалась при выходе за границы пятна, что изображено на рисунке ниже (см. Рисунок 3.9), где жёлтый цвет обозначает обработанные алгоритмом пиксели, а красный обнаруженные алгоритмом пиксели светового пятна;

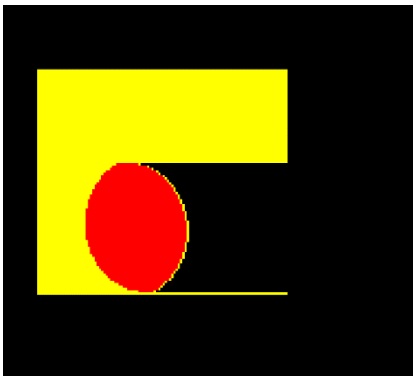

Рисунок 3.9 - Результат локализации пятна методом Quadro search

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ В п. В.2. Quadro search

3. Breadth search – полный аналог алгоритма Breadth spot для поиска пятна, делает тоже самое, но уже внутри самого светового пятна, алгоритм пошагово обрабатывает пиксели, которые являются соседними к ранее обнаруженному, любым из методов, “первому” пикселю пятна. После обработки соседних пикселей алгоритм начинает обрабатывать соседей соседей и так далее, алгоритм обрабатывает только пиксели светового пятна, если алгоритм встречает пиксель, не являющийся частью светового пятна, он

не добавляет его в очередь проверки пикселей. Как отработал данный алгоритм можно увидеть на рисунке ниже (см. Рисунок 3.10), где красный цвет обозначает обнаруженные пиксели светового пятна, а желтый цвет обозначает “первый” полученный поиском пиксель.

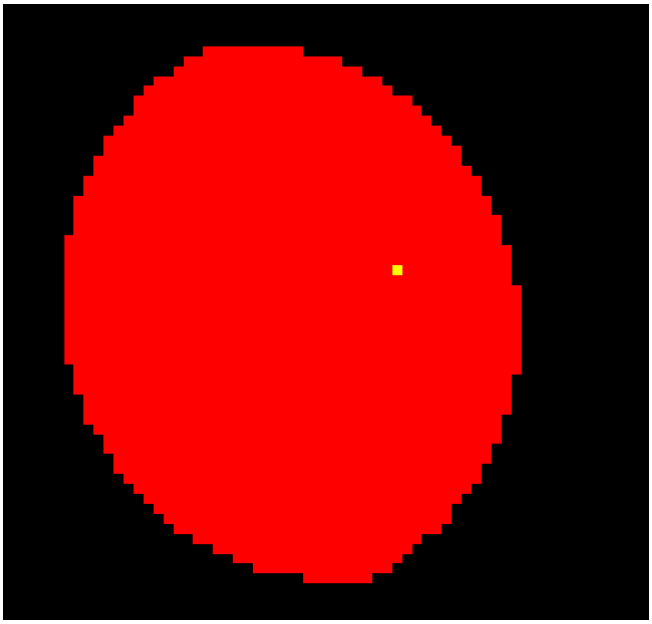

Рисунок 3.10 – Локализация пятна алгоритмом Breadth Search

Листинг реализации алгоритма на языках Python и Matlab приведён в ПРИЛОЖЕНИЕ В п. В.3. Breadth search

3.3. Алгоритм определения центра пятна

После нахождения и последующей локализации пятна для того, чтобы определить центр светового пятна на матрице использоваться один и тот же метод – координаты, полученные после локализации пятна, усредняются по формулам (3.1) и (3.2) представленным ниже

$$x_c = \frac{\sum_{ellipce} x_i}{n_x}, \quad (3.1)$$

$$y_c = \frac{\sum_{ellipce} y_i}{n_y}, \quad (3.2)$$

где:

x_c, y_c – координаты центра пятна на датчике;

n_x, n_y - количество найденных точек пятна;

x_i, y_i – точка пятна.

Данный метод достаточно точный для определения центра пятна, так как его среднее отличие вычисления координат центра пятна от теоретического находится в пределах одного пикселя. И далее данное отклонение уже варьируется от точности локализации пятна.

3.4. Определение положения Солнца

Определением положения Солнца является определение азимутального и зенитного углов по двум тригонометрическим формулам (3.3) и (3.3) соответственно:

$$\varphi = arctg(\frac{y_s - y_c}{x_s - x_c}), \quad (3.1)$$

$$\theta = arctg(\frac{\sqrt{(x_s - x_c)^2 + (y_s - y_c)^2} * pixelSize}{h}), \quad (3.2)$$

где:

φ – азимутальный угол;

θ – зенитный угол;

x_c, y_c – координаты центра матрицы;

x_s, y_s – координаты центра светового пятна;

h – расстояние матрицы до отверстия;

pixelSize – размер пикселя матрицы.

ГЛАВА 4.
СРАВНЕНИЕ АЛГОРИТМОВ

4.1. Сравнение эффективности алгоритмов

Для сравнения алгоритмов программой генерировалось 500 картинок со случайными параметрами азимутальных и зенитных углов, а после каждое из изображений обрабатывалось алгоритмами поиска и локализации пятна.

Основным критерием для сравнения алгоритмов будет средняя скорость и среднее количество базовых операций таких как сравнение цвета пикселя.

Сравнение алгоритмов в таблице можно посмотреть ниже (см. Таблица 4.1), где красным выделен оптимальный алгоритм по критериям скорости и количеству операций сравнения.

Таблица 4.1 – Сравнение работы алгоритмов поиска пятна.

	Matlab		**Python**		**Оба языка**
	Среднее время, с	**Худшее время, с**	**Среднее время, с**	**Худшее время, с**	**Количество операций сравнения**
Brute spot	0,03	0,055	0,716	1,36	300000
Rand spot	0,009	0,07	0,2	1,5	70000
Breadth spot	-	-	0,075	3,5	300000
Double spot	0,0075	0,015	0,21	0,4	300000
Round spot	0,017	0,06	0,35	1,5	160000
Round double spot	0,009	0,03	0,178	0,85	80000

Сравнение алгоритмов локализации пятна представлено на таблице ниже (см. Таблица 4.2)

Таблица 4.2 – Сравнение алгоритмов локализации пятна.

	Matlab	**Python**	**Количество операций сравнения**
	Среднее время, с	**Среднее время, с**	**N**
Quadro search	0,004	0,018	14800
Breadth search	0,03	0,009	19500

После приведённых алгоритмов остаётся ещё один не приведённый алгоритм локализации пятна, который является частью “Brute force” алгоритма. По своей сути “Brute force” является продолжением алгоритма “Brute spot” по обнаружению пятна, но в данном случае, после обнаружения пятна, алгоритм не прекращает свою работу и продолжает постепенно локализовывать пятно, учитывая при этом то, что он уже нашёл первую точку пятна и может ускорять свою работу прекращая обработку строки, если выйдет за пределы пятна или прекращать свою работу вовсе, если после обнаружения пятна он не найдёт ни одного пикселя пятна в текущей строке, что ускоряет алгоритм многократно. Очевидно, что данный алгоритм будет медленнее алгоритмов локализации “Quadro search” и “Breadth search”, так как данный алгоритм начинает локализовывать пятно проверяя все пиксели в строке матрицы начиная с самого первого, то есть, если пятно окажется на правой части светочувствительной матрицы, алгоритм, после обнаружения первой точки пятна и при прекращении обработки строки, где находилась первая точка, начнёт обработку следующей строки с самого начала и будет проходить всю длину матрицы, что значительно замедляет алгоритм по сравнению с другими.

4.2. Сравнение точности алгоритмов определения центра светового пятна и алгоритмов определения ориентационных углов.

Все алгоритмы, представленные выше, дают одинаковый результат при определении центра пятна, по формулам (3.1) (3.2) указанным выше. Ошибка вычисления центра светового пятна алгоритмом в среднем при обработке 500 изображений была: 0.46 пикселя, различие может быть обусловлено ошибками округления в алгоритмах вычисления центра пятна.

Ошибка при определении азимутального и зенитного углов при обработке 500 изображений пятна составила: для азимутального угла: 1,896°, а для зенитного угла: 1,674°. Ошибка может быть обусловлена многими округлениями во время вычислений углов, а также ошибкой определения центра светового пятна.

Однако, не стоит забывать, что в случае поиска пятна алгоритмом “Double spot” или “Round double spot”, если пятно окажется размерами меньше 2 пикселей и окажется на пикселе, который не обработает алгоритм, т.к. он обрабатывает только каждый второй пиксель, центр пятна не определится, т.к. пятно вовсе не будет обнаружено. Данную проблему можно исправить, введя некоторую функцию, которая будет в зависимости отдалённости текущего пикселя от центра матрицы рассчитывать минимально возможный размер светового пятна в некоторой области возле положения текущего пикселя и начинать обрабатывать не каждый второй пиксель, а абсолютно каждый, что исключит возможную ошибку, приведённую выше. Используя эту же функцию, можно также ускорить работу алгоритмов “Double spot” и “Round double spot” обрабатывая, в зависимости от удаления текущего пикселя от центра матрицы, каждый n-ый пиксель, постепенно уменьшая пробел между обрабатываемыми пикселями.

4.3. Плюсы и минусы алгоритмов

Описание плюсов и минусов разработанных алгоритмов приведено в таблице ниже (см. Таблица 4.3)

Таблица 4.3 – Сравнение алгоритмов

Алгоритм	**Плюсы**	**Минусы**
Brute force & spot	- надёжность; - точность.	- скорость работы; - количество операций сравнения.
Rand spot	- скорость.	- необходимо где-либо запоминать уже полученные значения, чтобы избегать повторений; при отсутствии пятна на матрице скорость будет меньше, чем у других алгоритмов.
Double spot	- скорость; - быстрый во всех реализациях.	- относительно большое количество операций сравнения; есть возможность пропустить пятно размером в 1 пиксель.
Round spot	- скорость; - быстрый во всех реализациях; - относительно малое количество операций сравнения.	- при отсутствии пятна сильно замедляется; - большое количество операций сравнения при большом удалении пятна от центра матрицы, т.к. алгоритм является аналогом “Brute spot”, но с другой начальной точкой
Round double spot	- скорость; - относительно малое количество операций сравнения.	- большое количество арифметических операций; - есть возможность пропустить пятно размером в 1 пиксель, немного модифицировав алгоритм данной проблемы можно избежать.
Breadth spot	- точность;	- скорость; - медленно работает без наличия ссылок/указателей в языке программирования; - большие вычислительные затраты (при отсутствии ссылок/указателей);
Quadro search	- надёжность; - точность; - скорость работы;	- возможна лишняя работа при постоянном размере исследуемого квадрата.
Breadth search	- скорость; - точность;	- медленно работает без наличия ссылок/указателей в языке программирования;

Чтобы определить самый эффективный алгоритм поиска светового пятна необходимо знать на каком языке программирования будет программироваться микропроцессоров. Если раньше в основном выбор стоял между языками С и Ассемблер, то в наше время появляется всё больше языков программирования доступных для микропроцессоров такие как C++, MicroPython, JavaScript, Blockly и т.д.

Одной из ключевых особенностей языков Python и JavaScript является динамическая типизация, в отличии от таких языков как C, C++ в Python и JavaScript нет необходимости определять тип переменных в коде, что прилично замедляет выполнения кода, так как, например, даже для базовой операции сложения двух переменных a + b программа должна сначала определить тип объектов a и b, ведь он неизвестен при компиляции. Затем программа запрашивает подходящую операцию сложения, которая может быть перезагружена методом, определённым пользователем. Вероятно, эта самая особенность значительно замедляла скорость всех выше представленных алгоритмов на Python. При реализации алгоритмов на языке программирования C данная проблема будет отсутствовать поэтому скорость работы алгоритмов значительно ускорится.

ЗАКЛЮЧЕНИЕ

В рамках выполнения дипломной работы было совершено следующее:

1. Изучены виды и характеристики светочувствительных матриц;
2. Изучено устройство датчика направления на Солнце на основании КМОП- матрицы;
3. Разработана модель для генерации изображений с матрицы освещённого датчика. Входными параметрами модели являются: размеры и разрешение матрицы, высота бленды над матрицей, диаметр отверстия бленды, толщина бленды, зенитный и азимутальный углы падения луча. Разработанная модель использовалась для генерации набора изображений для тестирования алгоритмов;
4. Рассмотрены современные методы обработки изображений на различных языках программирования, а также способы нахождения светового пятна на матрице;
5. Разработаны алгоритмы поиска пятна, локализации пятна и определения центра пятна: “Brute force”, “Brute spot”, “Breadth spot”, “Breadth search”, “Double spot”, “Rand spot”, “Round spot”, “Round Double spot”, “Quadro search”.
6. Проведено сравнение разработанных алгоритмов по затраченному времени выполнения и количеству базовых операций сравнения.

Результаты проделанной работы показывают, что для обработки изображения с матрицы датчика и определения центра светового пятна могут использоваться различные алгоритмы поиска и локализации пятна. Самым эффективным алгоритмом, среди исследуемых, по поиску светового пятная оказался алгоритм “Round double spot”. Среди алгоритмов по локализации пятна наиболее эффективным оказался алгоритм “Breadth search”. Стоит заметить, что данный алгоритм по сути является алгоритмом поиска в ширину и требует использования таких структур данных как очередь или связанный список, если использование таких структур данных затруднительно, например, при отсутствии в языке программирования ссылок и указателей, то в данном случае предпочтительнее использовать алгоритм “Quadro search, который близок по эффективности.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Andrea Antonello, Lorenzo Olivieri, Alessandro Francesconi, Low-Cost, High-Resolution, Self-Powered, Miniaturized Sun Sensor For Space Applications, CISAS G. Colombo, University of Padova, Italy; DII, University of Padova, Italy.
2. Marcello BUONOCORE, Michele GRASSI and Giancarlo RUFINO, Aps-based miniature sun sensor for Earth observation nanosatellites.
3. Jorge Prado-Monila, Hector Arriaga-Arrollo and Julio-Cesar Balanza-Ramagnoli, Low-cost CMOS active pixel Sun Sensor for nanosatellite's two-axis attitude determination. Instituto de Geografia, Universidad Nacional Autonoma de Mexico, Mexico City, Mexico.
4. Ahad Ali, Fahad Tanveer, Low-Cost Design and Development of 2-Axis Digital Sun Sensor, AOCS Section, Satellite Research & Development Centre-Karachi, (SUPARCO) Pakistan, Journal of Space Technology, 2011. – С.82-87. С.
5. Rufino, G., and Grassi, M., "Digital sun sensor multi-spot operation." Sensors 12.12 (2012): 16451-16465.
6. MT9V032 1/3-Inch Wide VGA CMOS Digital Image Sensor, © Semiconductor Components Industries, LLC, 2006 May, 2017 − Rev. 7
7. C Vs Python [Электронный ресурс]: Особенности и различия C и Python – Режим доступа: https://www.geeksforgeeks.org/c-vs-python/#:~:text=The%20difference%20between%20C%20and,web%20development%20and%20many%20more. – Дата доступа: 20.05.2021.
8. Introduction to the A* algorithm [Электронный ресурс]: Алгоритм поиска A* – Режим доступа: https://www.redblobgames.com/pathfinding/a-star/introduction.html. – Дата доступа: 01.05.2021.
9. Grids and Graphs [Электронный ресурс]: Поиск в сетках и графах – Режим доступа: https://www.redblobgames.com/pathfinding/grids/graphs.html. – Дата доступа: 02.05.2021.
10. Circle fill on a grid [Электронный ресурс]: Заполнение окружностей на сетках – Режим доступа: https://www.redblobgames.com/grids/circle-drawing. – Дата доступа: 02.05.2021.

ПРИЛОЖЕНИЕ А

Генерация изображений пятна на матрице

```
close all;
clear all;
clc;

d = 0.0003;     % диаметр отверстия
t = 100e-6;     % толщина отверстия
h = 0.0007;     % высота отверстия
format long
H = 4e-3;    % Размеры матрицы
W = 4e-3;
x = zeros(1, 360);
y = zeros(1, 360);
pxH = 752;
pxW = 752;
pxSize = W / pxW;
randMinTh = 0;
randMaxTh = 80;
randMinPhi = -180;
randMaxPhi = 180;

% ЗАПИСЬ В ТАБЛИЦУ
filename = './tables/compare_new.xlsx';
fileData = {"Theta", "Phi", "Центр пятна X", "Центр пятна Y"};
writecell(fileData, filename, 'sheet', 1, 'Range', "B1");
dataArray = [];
xArray = [];
yArray = [];
thetaArray = [];
phiArray = [];

for iii = 1 : 500
%     theta_src = 55;     % исходный зенитный угол
%     phi_src = 58;       % исходный азимутальный угол
    theta_src = randMinTh + rand(1, 1) * (randMaxTh - randMinTh);     % СЛУЧАЙНЫЙ зенитный угол
    phi_src = randMinPhi + rand(1, 1) * (randMaxPhi - randMinPhi);       % СЛУЧАЙНЫЙ азимутальный угол

    theta = deg2rad(90) - deg2rad(theta_src);      % зенитный угол падения солнечных лучей
    phi = deg2rad(phi_src);      % азимутальный угол падения солнечных лучей
    dC = h * tan(pi/2 - theta);
    angle = deg2rad(0 : 359);          % вспомогательный массив углов
    x = (d/2) * cos(angle);             % координата x контура пятна
    y = (d/2) * sin(angle);             % координата y контура пятна
    xPx = ceil((x + W/2) / pxSize);  % x - контур пятна в пикселях
    yPx = ceil((y + H/2) / pxSize);  % y - контур пятна в пикселях
    x1 = dC * cos(phi + pi); % x1
    y1 = dC * sin(phi + pi); % y1
    x1Px = ceil((x1 + W/2 + x) / pxSize); % x1 px
    y1Px = ceil((y1 + H/2 + y) / pxSize); % y1 px
```

```
x0 = (t + h) * tan(pi/2 - theta) * cos(phi + pi); % контур смещённого пятна (без учёта обрезания круга)
y0 = (t + h) * tan(pi/2 - theta) * sin(phi + pi); % контур смещённого пятна (без учёта обрезания круга)

psY = ceil((y1/2 + y0/2 + H/2) / pxSize);
psX = ceil((x1/2 + x0/2 + W/2) / pxSize);

x0Px = ceil((x0 + W/2 + x) / pxSize); % x0Px - контур смещённого пятна (без учёта обрезания круга) в пикселях
y0Px = ceil((y0 + H/2 + y) / pxSize); % y0Px - контур смещённого пятна (без учёта обрезания круга) в пикселях

% рисование смешённого и повёрнутого пятна
image = zeros(pxW, pxH, 3);

% круги вокруг пятна и в центре матрицы.
for i = 1 : length(x0Px)
%     image(xPx(i), yPx(i), [3 3 3]) = 1;
%     image(x1Px(i), y1Px(i), [1 2 1]) = 1;
%     image(x0Px(i), y0Px(i), [1 1 1]) = 1;
end

% найти максимумы / минимумы из двух смещённых окружностей
minX = min(x0Px);
minY = min(y0Px);
maxX = max(x1Px);
maxY = max(x1Px);

if (minX > min(x1Px))
  minX = min(x1Px);
end
if (minY > min(x1Px))
  minY = min(x1Px);
end
if (maxX < max(x0Px))
  maxX = max(x0Px);
end
if (maxY < max(y0Px))
  maxY = max(y0Px);
end

% можно оптимизировать
x_ = []; % координата усечённого и перемещённого и повёрнутого круга
y_ = []; % координата усечённого и перемещённого и повёрнутого круга

% найти пересечение двух окружностей
for i = minX : maxX
   for j = minY : maxY

      if (i <= 0 || i > pxW)
         continue;
      end
      if (j <= 0 || j > pxH)
         continue;
      end
```

```
            in = inpolygon(i, j, x0Px, y0Px);
            in2 = inpolygon(i, j, x1Px, y1Px);
            if (in ~= 0 && in2 ~= 0)
                x_(end + 1) = i;
                y_(end + 1) = j;
            end
        end
    end

    % Закрашивание общей области
    for i = 1 : length(x_)
     image(x_(i), y_(i), 1) = 1;    %Red
     image(x_(i), y_(i), 2) = 1;  %Green
     image(x_(i), y_(i), 3) = 0;  %Blue
    end

    psX = ceil(sum(x_) / length(x_));
    psY = ceil(sum(y_) / length(y_));
    imwrite(image, strcat("./images/circle_", num2str(iii), ".png"));

    fileData = {rad2deg(theta), rad2deg(phi), psY, psX};
    writecell(fileData, filename, 'sheet', 1, 'Range', strcat("B" + num2str(iii + 1)));
end
```

ПРИЛОЖЕНИЕ Б

Алгоритмы поиска пятна на матрице

Б.1. Brute spot

Б.1.1 Matlab

```
function [spotX, spotY] = bruteSpot(imageGS)
    pxH = length(imageGS);
    pxW = length(imageGS(1, :));

    columns = pxH;
    rows = pxW;

    spotX = 0;
    spotY = 0;

    isFlag = false;

    for  y = 1 : rows
        for x = 1 : columns
            if(imageGS(y, x) ~= 0)
                spotX = x;
                spotY = y;

                isFlag = true;
            end
        end
        if (isFlag)
            break;
        end
    end
end
```

Б.1.2 Python

```
def bruteSpot(arrGS, arr, folder_path, iii, generateImage):
  y = 0
  x = 0
  spotX = -1
  spotY = -1
  flag = False

  # if (generateImage):
    # arrCopy = arr.copy()
```

```
 for y, rows in enumerate(arrGS):
   for x, cols in enumerate(rows):
     # if (generateImage):
       # arrCopy[y][x] = [255, 255, 0]

     if (cols[0] > 10 or cols[1] < 255):

       # if (generateImage):
         # arrCopy[y][x] = [255, 0, 0]

       spotX = x
       spotY = y
       flag = True
       break

   if (flag):
     break

 # if (generateImage):
   # img = Image.fromarray(arrCopy)
   # img.save(os.path.join(folder_path, 'spot_brute_image_' + str(iii) + '.png'))

 return [spotX, spotY]
```

Б.2. Rand spot

Б.2.1 Matlab

```
function [spotX, spotY] = randSpot(imageGS)
    % анонимная функция для последовательного вызова (){}
    subindex = @(A, r) A{r};

    pxW = 752;
    pxH = 752;

    columns = 752;
    rows = 752;

    flag = false;

    spotX = 0;
    spotY = 0;

    randRows = randperm(rows, rows);
    randCols = randperm(columns, columns);

    for y = randRows
        for x = randCols

            if (imageGS(y, x) ~= 0)
```

```
            spotX = x;
            spotY = y;
            flag = true;
            break;
          end
        end
        if (flag)
          break;
        end
    end
end
```

Б.2.2 Python

```
  pointX = -1
  pointY = -1
  find = False
  rows = len(arrGS)
  cols = len(arrGS[0])
  if (generateImage):
    arrCopy = arr.copy()

  randRows = list(range(rows));
  random.shuffle(randRows);
  randCols = list(range(cols));
  random.shuffle(randCols);

  for y in randRows:
    for x in randCols:
      if (generateImage):
        arrCopy[y][x] = [255, 0, 0]
      if (arrGS[y][x][0] > 10 or arrGS[y][x][1] < 255):
        find = True
        pointX = x
        pointY = y
        break
    if (find):
      break

  if (generateImage):
    img = Image.fromarray(arrCopy)
    img.save(os.path.join(folder_path, '2spot_rand_image_' + str(iii) + '.png'))

  return [pointX, pointY]
```

Б.3. Breadth spot

Б.3.1 Matlab

```
function [pointX, pointY] = breadthSpot(imageGS, spotX, spotY)
    if (spotX == 0 || spotY == 0)
        breadthX = 0;
        breadthY = 0;
        return;
    end

    pxH = length(imageGS);
    pxW = length(imageGS(1, :));

    breadthX = spotX;
    breadthY = spotY;
    pointX = 0;
    pointY = 0;

    flag = false;

    frontier = CQueue();
    frontier.push([breadthX, breadthY]);
    maxReached = [imageGS];
    maxReached(breadthY, breadthX) = 100;

    while (frontier.size() ~= 0)
        if (flag)
            break;
        end
        isOkay = boolean([1 1 1 1]);
        current = frontier.pop();

        % определение соседей текущей клетки
        neighboor_r = [current(1) + 1, current(2)];
        neighboor_l = [current(1) - 1, current(2)];
        neighboor_t = [current(1), current(2) + 1];
        neighboor_b = [current(1), current(2) - 1];
        neighboors = [neighboor_r; neighboor_l; neighboor_t; neighboor_b];

        % проверка, чтобы соседи не выходили за матрицу
        for i = 1 : length(neighboors)
            if (neighboors(i, 1) > pxW || neighboors(i, 1) <= 0)
                isOkay(i) = boolean(0);
                neighboors(i, 1) = -1;
            end

            if (neighboors(i, 2) > pxH || neighboors(i, 2) <= 0)
                isOkay(i) = boolean(0);
                neighboors(i, 1) = -1;
            end
        end

        for i = 1 : length(neighboors)
            if (~isOkay(i))
              continue;
            end

            if (maxReached(neighboors(i, 2), neighboors(i, 1)) ~= 100)
```

```
            if (imageGS(neighboors(i, 2), neighboors(i, 1)) == 0)
                frontier.push(neighboors(i, :));
                maxReached(neighboors(i, 2), neighboors(i, 1)) = 0;
            end
            if (imageGS(neighboors(i, 2), neighboors(i, 1)) == 226)
                pointY = neighboors(i, 2);
                pointX = neighboors(i, 1);
                flag = true;
                break;
            end
        end
    end
  end
end
```

Б.3.2 Python

```
def breadthSpot(arrGS, point, arr, folder_path, iii, generateImage, algoType):
  # if (generateImage):
    # arrCopy = arr.copy()
  if (point[0] == -1 or point[1] == -1):
    # if (generateImage):
      # fileName = algoType + 'breadth_image_' + str(iii) + '.png'

      # img = Image.fromarray(arrCopy)
      # img.save(os.path.join(folder_path, fileName))
    return [0, 0]

  # if (generateImage):
    # arrCopy = arr.copy()
  spotX = -1
  spotY = -1
  flag = False

  point = tuple(point)
  frontier = deque()
  frontier.append(point)
  reached = set()
  reached.add(point)

  while (not len(frontier) == 0):
    if (flag):
      break

    current = frontier.popleft()
    neighboors = [
      [current[0] + 1, current[1]],
      [current[0] - 1, current[1]],
      [current[0], current[1] + 1],
      [current[0], current[1] - 1],
    ]

    for next in neighboors:
      if (next[0] < 0):
```

```
      next[0] = 0
    if (next[1] < 0):
      next[1] = 0
    if (next[0] >= pxW):
      next[0] = pxW - 1
    if (next[1] >= pxH):
      next[1] = pxH - 1
    next = tuple(next)

    if (next not in reached):

      # if (generateImage):
        # arrCopy[next[1], next[0]] = [0, 255, 0]

      frontier.append(next)
      reached.add(next)

      if (arrGS[next[1]][next[0]][0] > 10 or arrGS[next[1]][next[0]][1] < 255):
        spotX = next[0]
        spotY = next[1]
        flag = True
        break

  # if (generateImage):
    # fileName = algoType + 'breadth_image_' + str(iii) + '.png'

    # img = Image.fromarray(arrCopy)
    # img.save(os.path.join(folder_path, fileName))

  return [spotX, spotY]
```

Б.4. Double spot

Б.4.1 Matlab

```
function [pointY, pointX] = doubleSpot(imageGS)
    pxH = length(imageGS);
    pxW = length(imageGS(1, :));

    columns = pxW;
    rows = pxH;

    pointX = 0;
    pointY = 0;

    for y = 0 : floor(rows / 2) - 1
        y = y * 2 + 1;
        for x = 0 : floor(columns / 2) - 1
            if(imageGS(y, x * 2 + 1) ~= 0)
```

```
            pointX = x * 2 + 1;
            pointY = y;

            return;
          end
        end
      end

      rows = [1, pxH];
      cols = [1, pxW];

      for y = rows
        for x = 1 : floor(length(imageGS(y, :)) / 2)
          x = x * 2;

          if(imageGS(y, x) ~= 0)
            pointX = x;
            pointY = y;
            return;
          end
        end
      end

      for x = cols
        for y = 1 : floor(length(imageGS) / 2)
          y = y * 2;

          if(imageGS(y, x) ~= 0)
            pointX = x;
            pointY = y;
            return;
          end
        end
      end
    end
```

Б.4.2 Python

```
def doubleSearch(arrGS, arr, folder_path, iii, generateImage):
  y = 0
  x = 0
  pointX = -1
  pointY = -1
  flag = False

  # if (generateImage):
    # arrCopy = arr.copy()

  for y in range(0, math.floor(len(arrGS) / 2), 2):
    # y = y * 2
    for x in range(0, math.floor(len(arrGS[y]) / 2), 2):

      # if (generateImage):
        # arrCopy[y][x * 2] = [255, 0, 0]
```

```
    if (arrGS[y][x][0] > 10 or arrGS[y][x][1] < 255):

      # if (generateImage):
        # arrCopy[y][x * 2] = [0, 0, 255]

      pointX = x;
      pointY = y;
      flag = True
      break

  if (flag):
    break

if (pointX == -1 or pointY == -1):
  rows = [0, len(arrGS) - 1]
  cols = [0, len(arrGS[0]) - 1]

  for y in rows:
    for x in range(math.floor(len(arrGS[y]) / 2)):

      # if (generateImage):
        # arrCopy[y][x] = [255, 255, 0]

      if (arrGS[y][x * 2 + 1][0] > 10 or arrGS[y][x * 2 + 1][1] < 255):

        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        pointX = x;
        pointY = y;
        break

  for x in cols:
    for y in range(math.floor(len(arrGS) / 2)):

      # if (generateImage):
        # arrCopy[y][x] = [255, 255, 0]

      if (arrGS[y * 2 + 1][x][0] > 10 or arrGS[y * 2 + 1][x][1] < 255):

        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        pointX = x;
        pointY = y;
        break

# if (generateImage):
  # img = Image.fromarray(arrCopy)
  # img.save(os.path.join(folder_path, 'spot_double_image_' + str(iii) + '.png'))

return [pointX, pointY]
```

Б.5. Round spot

Б.5.1 Matlab

```
function [spotX, spotY] = roundSpot(imageGS)
   pxH = length(imageGS);
   pxW = length(imageGS(1, :));

   imgCenterX = pxW / 2;
   imgCenterY = pxH / 2;

   spotX = 0;
   spotY = 0;

   isFlag = false;

   for  rad = 0 : imgCenterX
      topLeftX = imgCenterX - rad + 1;
      topLeftY = imgCenterY - rad + 1;
      topRightX = imgCenterX + rad;
      topRightY = imgCenterY - rad + 1;
      bottomRightX = imgCenterX + rad;
      bottomRightY = imgCenterY + rad;
      bottomLeftX = imgCenterX - rad + 1;
      bottomLeftY = imgCenterY + rad;

      for x = topLeftX : topRightX
         if(imageGS(topLeftY, x) ~= 0)
            spotX = x;
            spotY = topLeftY;
            isFlag = true;
            break;
         end
      end

      for x = bottomLeftX : bottomRightX
         if(imageGS(bottomLeftY, x) ~= 0)
            spotX = x;
            spotY = bottomLeftY;
            isFlag = true;
            break;
         end
      end

      for y = topLeftY : bottomLeftY
         if(imageGS(y, topLeftX) ~= 0)
            spotX = topLeftX;
            spotY = y;
            isFlag = true;
            break;
         end
      end
```

```
        for x = topRightY : bottomRightY
            if(imageGS(topRightX, x) ~= 0)
                spotX = x;
                spotY = topRightX;
                isFlag = true;
                break;
            end
        end

        if (isFlag)
            break;
        end
    end
end
```

Б.5.2 Python

```
def roundSearch(arrGS, arr, folder_path, iii, generateImage):
  # if (generateImage):
    # arrCopy = arr.copy()

  width = len(arrGS[0])
  height = len(arrGS)
  pointX = -1
  pointY = -1
  imgCenterX = math.floor(width / 2)
  imgCenterY = math.floor(height / 2)
  flag = False

  for rad in range(math.floor(width / 2)):
    topLeftX = imgCenterX - rad - 1
    topLeftY = imgCenterY - rad - 1
    topRightX = imgCenterX + rad
    topRightY = imgCenterY - rad - 1
    bottomRightX = imgCenterX + rad
    bottomRightY = imgCenterY + rad + 1
    bottomLeftX = imgCenterX - rad - 1
    bottomLeftY = imgCenterY + rad

    color = [0, 255, 0]
    if (rad % 2 == 0):
      color = [0, 0, 255]

    for x in range(topLeftX, topRightX):
      # if (generateImage):
        # arrCopy[y][x] = color

      if (arrGS[topLeftY][x][0] > 10 or arrGS[topLeftY][x][1] < 255):
        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        pointX = x
        pointY = topLeftY
```

```
      flag = True
      break

  for x in range(bottomLeftX, bottomRightX):
    # if (generateImage):
      # arrCopy[y][x] = color

    if (arrGS[bottomLeftY][x][0] > 10 or arrGS[bottomLeftY][x][1] < 255):
      # if (generateImage):
        # arrCopy[y][x] = [255, 0, 0]

      pointX = x
      pointY = bottomLeftY
      flag = True
      break

  for y in range(topLeftY, bottomLeftY):
    # if (generateImage):
      # arrCopy[y][x] = color

    if (arrGS[y][topLeftX][0] > 10 or arrGS[y][topLeftX][1] < 255):
      # if (generateImage):
        # arrCopy[y][x] = [255, 0, 0]

      pointX = topLeftX
      pointY = y
      flag = True
      break

  for y in range(topRightY, bottomRightY):
    # if (generateImage):
      # arrCopy[y][x] = color

    if (arrGS[y][topRightX][0] > 10 or arrGS[y][topRightX][1] < 255):
      # if (generateImage):
        # arrCopy[y][x] = [255, 0, 0]

      pointX = topRightX
      pointY = y
      flag = True
      break

  if (flag):
    break;

# if (generateImage):
  # img = Image.fromarray(arrCopy)
  # img.save(os.path.join(folder_path, 'spot_round_image_' + str(iii) + '.png'))

return [pointX, pointY]
```

Б.6. Round double spot

Б.6.1 Matlab

```
function [spotX, spotY] = roundDoubleSpot(imageGS)
   subindex = @(A, r) A(r);

   pxH = length(imageGS);
   pxW = length(imageGS(1, :));

   imgCenterX = pxW / 2;
   imgCenterY = pxH / 2;

   spotX = 0;
   spotY = 0;

   isFlag = false;

   for  rad = 0 : imgCenterX
      topLeftX = imgCenterX - rad + 1;
      topLeftY = imgCenterY - rad + 1;
      topRightX = imgCenterX + rad;
      topRightY = imgCenterY - rad + 1;
      bottomRightX = imgCenterX + rad;
      bottomRightY = imgCenterY + rad;
      bottomLeftX = imgCenterX - rad + 1;
      bottomLeftY = imgCenterY + rad;

      for x = (topLeftX : 2 : topRightX)
         if(imageGS(topLeftY, x) ~= 0)
            spotX = x;
            spotY = topLeftY;
            isFlag = true;
            break;
         end
      end

      for x = (bottomLeftX : 2 : bottomRightX)
         if(imageGS(bottomLeftY, x) ~= 0)
            spotX = x;
            spotY = bottomLeftY;
            isFlag = true;
            break;
         end
      end

      for y = (topLeftY : 2 : bottomLeftY)
         if(imageGS(y, topLeftX) ~= 0)
            spotX = topLeftX;
            spotY = y;
            isFlag = true;
            break;
```

```
            end
        end

        for x = (topRightY : 2 : bottomRightY)
            if(imageGS(topRightX, x) ~= 0)
                spotX = x;
                spotY = topRightX;
                isFlag = true;
                break;
            end
        end

        if (isFlag)
            break;
        end
    end
end
```

Б.6.2 Python

```
def roundDoubleSearch(arrGS, arr, folder_path, iii, generateImage):
  # if (generateImage):
    # arrCopy = arr.copy()

  width = len(arrGS[0]) - 1
  height = len(arrGS) - 1
  pointX = -1
  pointY = -1
  imgCenterX = math.floor(width / 2)
  imgCenterY = math.floor(height / 2)
  flag = False

  for rad in range(math.floor(width / 2) + 1):
    topLeftX = imgCenterX - rad
    topLeftY = imgCenterY - rad
    topRightX = imgCenterX + rad
    topRightY = imgCenterY - rad
    bottomRightX = imgCenterX + rad + 1
    bottomRightY = imgCenterY + rad + 1
    bottomLeftX = imgCenterX - rad
    bottomLeftY = imgCenterY + rad

    color = [0, 255, 0]
    if (rad % 2 == 0):
      color = [0, 0, 255]

    for x in range(topLeftX, topRightX, 2):
      # if (generateImage):
        # arrCopy[y][x] = color

      if (arrGS[topLeftY][x][0] > 10 or arrGS[topLeftY][x][1] < 255):
        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]
```

```
      pointX = x
      pointY = topLeftY
      flag = True
      break

    for x in range(bottomLeftX, bottomRightX, 2):
      # if (generateImage):
        # arrCopy[y][x] = color

      if (arrGS[bottomLeftY][x][0] > 10 or arrGS[bottomLeftY][x][1] < 255):
        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        pointX = x
        pointY = bottomLeftY
        flag = True
        break

    for y in reversed(range(topLeftY, bottomLeftY, 2)):
      # if (generateImage):
        # arrCopy[y][x] = color

      if (arrGS[y][topLeftX][0] > 10 or arrGS[y][topLeftX][1] < 255):
        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        pointX = topLeftX
        pointY = y
        flag = True
        break

    for y in reversed(range(topRightY, bottomRightY, 2)):
      # if (generateImage):
        # arrCopy[y][x] = color

      if (arrGS[y][topRightX][0] > 10 or arrGS[y][topRightX][1] < 255):
        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        pointX = topRightX
        pointY = y
        flag = True
        break

    if (flag):
       break;

  if (pointX == -1 or pointY == -1):
    rows = [0, len(arrGS) - 1]
    cols = [0, len(arrGS[0]) - 1]

    for y in rows:
      for x in range(len(arrGS[y])):

        # if (generateImage):
```

```
        # arrCopy[y][x] = [255, 255, 0]

      if (arrGS[y][x][0] > 10 or arrGS[y][x][1] < 255):

        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        pointX = x;
        pointY = y;
        break

  for x in cols:
    for y in range(len(arrGS)):

      # if (generateImage):
        # arrCopy[y][x] = [255, 255, 0]

      if (arrGS[y][x][0] > 10 or arrGS[y][x][1] < 255):

        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        pointX = x;
        pointY = y;
        break

# if (generateImage):
  # img = Image.fromarray(arrCopy)
  # img.save(os.path.join(folder_path, 'spot_round_double_image_' + str(iii) + '.png'))

return [pointX, pointY]
```

ПРИЛОЖЕНИЕ В

Алгоритмы локализации пятна на матрице

B.1. Brute force

B.1.1 Matlab

```
function [bruteX, bruteY] = bruteForce(imageGS)
   pxH = length(imageGS);
   pxW = length(imageGS(1, :));

   columns = pxH;
   rows = pxW;

   coloredX = [];
   coloredY = [];

   isFlag = false;
   isFlag2 = false;

   for  y = 1 : rows
      columnFlag = false;
      for x = 1 : columns
         if(imageGS(y, x) ~= 0)
            coloredX(end + 1) = x;
            coloredY(end + 1) = y;

            columnFlag = true;
            isFlag2 = true;

            if (length(coloredX) == 1)
              isFlag = true;
            end
         else
            if (columnFlag)
              break;
            end
         end
      end
      if (isFlag)
         if (~isFlag2)
           break;
         end
      end
      isFlag2 = false;

   end
```

```
    if (length(coloredX) == 0 || length(coloredY) == 0)
        bruteX = 0;
        bruteY = 0;
        return;
    end

    bruteX = ceil(sum(coloredX) / length(coloredX));
    bruteY = ceil(sum(coloredY) / length(coloredY));
end
```

B.1.2 Python

```
def bruteForce(arrGS, arr, folder_path, iii, generateImage):
  dtotalX = []
  dtotalY = []
  rowFlag = False
  rowFlag_2 = False

  # if (generateImage):
    # arrCopy = arr.copy()

  for y, rows in enumerate(arrGS):
    colFlag = False

    for x, cols in enumerate(rows):
      # if (generateImage):
        # arrCopy[y][x] = [255, 255, 0]

      if (cols[0] > 10 or cols[1] < 255):

        # if (generateImage):
          # arrCopy[y][x] = [255, 0, 0]

        dtotalX.append(x + 1)
        dtotalY.append(y + 1)

        rowFlag_2 = True
        colFlag = True
        if (len(dtotalX) == 1):
          rowFlag = True
      else:
        if (colFlag):
          break


    if (rowFlag and not(rowFlag_2)):
      break

    rowFlag_2 = False

  # if (generateImage):
    # img = Image.fromarray(arrCopy)
    # img.save(os.path.join(folder_path, 'brute_image_' + str(iii) + '.png'))
```

```
if (len(dtotalX) == 0 or len(dtotalY) == 0):
  return [0, 0]

centerX = math.ceil(sum(dtotalX) / len(dtotalX))
centerY = math.ceil(sum(dtotalY) / len(dtotalY))

return [centerX, centerY]
```

B.2. Quadro search

B.2.1 Matlab

```
function [randX, randY] = quadroSearch(imageGS, spotX, spotY)

    % параметры устройства
    d = 0.0003;     % диаметр отверстия

    format long
    height = 4e-3;
    width = 4e-3;

    pxH = length(imageGS);
    pxW = length(imageGS(1, :));
    pxSize = width / pxW;

    angle = deg2rad(0 : 359);           % вспомогательный массив углов
    defCircleX = (d/2) * cos(angle);            % координата x контура пятна
    defCircleY = (d/2) * sin(angle);            % координата y контура пятна
    xPx = ceil((defCircleX + width/2) / pxSize);  % x - контур пятна в пикселях
    yPx = ceil((defCircleY + height/2) / pxSize);  % y - контур пятна в пикселях
    defPxRadius = ceil((max(xPx) - min(xPx)) / 2);

    coloredX = [spotX];
    coloredY = [spotY];

    % Т.К. ПОЛУЧЕННОЕ ПЯТНО 100% МЕНЬШЕ ЧЕМ ТО, ЧТО ПОЛУЧАЕТСЯ БЕЗ НАКЛОНА
    % МАТРИЦЫ МОЖНО ПОИСКАТЬ ПИКСЕЛИ В ПРЕДЕЛАХ ДИАМЕТРА НЕИЗМЕННЁНОГО КРУГА
    % В 4 КВАДРАНТАХ

    % левый верхний квадрат граница
    leftTopX = coloredX(1) - defPxRadius *  2;
    leftTopY = coloredY(1) - defPxRadius * 2;
    % левый нижний квадрат граница
    leftBottomX = coloredX(1) - defPxRadius * 2;
    leftBottomY = coloredY(1) + defPxRadius * 2;
    % правый верхний квадрат граница
    rightTopX = coloredX(1) + defPxRadius * 2;
    rightTopY = coloredY(1) - defPxRadius * 2;
    % правый нижний квадрат
```

```
rightBottomX = coloredX(1) + defPxRadius * 2;
rightBottomY = coloredY(1) + defPxRadius * 2;

quadroPositionsX = [leftTopX, rightTopX, rightBottomX, leftBottomX];
quadroPositionsY = [leftTopY, rightTopY, rightBottomY, leftBottomY];

% проверка не выходят ли координаты за границы матрицы, если выходят ставим
% = размер матрицы или 0
for i = 1 : 4
  if (quadroPositionsX(i) <= 0)
    quadroPositionsX(i) = 1;
  else
     if (quadroPositionsX(i) > pxW)
        quadroPositionsX(i) = pxW;
     end
  end
  if (quadroPositionsY(i) <= 0)
    quadroPositionsY(i) = 1;
  else
     if (quadroPositionsY(i) > pxH)
        quadroPositionsY(i) = pxH;
     end
  end
end

isFlag = false;
isFlag2 = false;
coloredX = [];
coloredY = [];

for y = quadroPositionsY(1) : quadroPositionsY(3)
   columnFlag = false;
  for x = quadroPositionsX(1) : quadroPositionsX(3)
     if (y <= 0 || x <= 0)
       continue;
     end
      if(imageGS(y, x) ~= 0)
         coloredX(end + 1) = x;
         coloredY(end + 1) = y;

         columnFlag = true;
         isFlag2 = true;

         if (length(coloredX) == 2)
           isFlag = true;
         end
      else
         if (columnFlag)
           break;
         end
      end
  end
  if (isFlag)
    if (~isFlag2)
      break;
```

```
        end
      end
    end

    randX = ceil(sum(coloredX) / length(coloredX));
    randY = ceil(sum(coloredY) / length(coloredY));
end
```

B.2.2 Python

```
def quadroSearch(arrGS, point, arr, folder_path, iii, generateImage):
  # if (generateImage):
    # arrCopy = arr.copy()

  if (point[0] == -1 or point[1] == -1):
    # if (generateImage):
      # img = Image.fromarray(arrCopy)
      # img.save(os.path.join(folder_path, 'random_image_' + str(iii) + '.png'))
    return [0, 0]

  leftTopX = point[0] - defPxDiameter
  leftTopY = point[1] - defPxDiameter
  leftBottomX = point[0] - defPxDiameter;
  leftBottomY = point[1] + defPxDiameter;
  rightTopX = point[0] + defPxDiameter;
  rightTopY = point[1] - defPxDiameter;
  rightBottomX = point[0] + defPxDiameter;
  rightBottomY = point[1] + defPxDiameter;

  quadroX = [leftTopX, rightTopX, rightBottomX, leftBottomX]
  quadroY = [leftTopY, rightTopY, rightBottomY, leftBottomY]

  for i in range(4):
    if (quadroX[i] < 0):
      quadroX[i] = 0
    else :
      if (quadroX[i] >= pxW):
        quadroX[i] = pxW - 1

    if (quadroY[i] < 0):
      quadroY[i] = 0
    else :
      if (quadroY[i] >= pxH):
        quadroY[i] = pxH - 1

  y = 0
  x = 0
  dtotalX = []
  dtotalY = []
  rowFlag = False
  rowFlag_2 = False


  for y in range(quadroY[0], quadroY[2]):
```

```
  colFlag = False
  for x in range(quadroX[0], quadroX[2]):
    # if (generateImage):
      # arrCopy[y][x] = [255, 255, 0]

    if (arrGS[y][x][0] > 10 or arrGS[y][x][1] < 255):
      # if (generateImage):
        # arrCopy[y][x] = [255, 0, 0]

      dtotalX.append(x + 1)
      dtotalY.append(y + 1)

      colFlag = True
      rowFlag_2 = True

      if (len(dtotalX) == 1):
        rowFlag = True
    else:
      if (colFlag):
        break

  if (rowFlag and not(rowFlag_2)):
    break

  rowFlag_2 = False

 centerX = math.ceil(sum(dtotalX) / len(dtotalX))
 centerY = math.ceil(sum(dtotalY) / len(dtotalY))

 # if (generateImage):
   # img = Image.fromarray(arrCopy)
   # img.save(os.path.join(folder_path, 'c_rand_image_' + str(iii) + '.png'))

 return [centerX, centerY]
```

B.3. Breadth search

B.3.1 Matlab

```
function [breadthX, breadthY] = breadthSearch(imageGS, spotX, spotY)
   if (spotX == 0 || spotY == 0)
      breadthX = 0;
      breadthY = 0;
      return;
   end

   pxH = length(imageGS);
   pxW = length(imageGS(1, :));
```

```
    breadthX = spotX;
    breadthY = spotY;

    sumY = breadthX;
    sumX = breadthY;
    sizeX = 1;
    sizeY = 1;

    frontier = CQueue();
    frontier.push([breadthX, breadthY]);
    maxReached = [imageGS];
    maxReached(breadthY, breadthX) = 0;

    while (frontier.size() ~= 0)
        isOkay = boolean([1 1 1 1]);
        current = frontier.pop();

        % определение соседей текущей клетки
        neighboor_r = [current(1) + 1, current(2)];
        neighboor_l = [current(1) - 1, current(2)];
        neighboor_t = [current(1), current(2) + 1];
        neighboor_b = [current(1), current(2) - 1];
        neighboors = [neighboor_r; neighboor_l; neighboor_t; neighboor_b];

        % проверка, чтобы соседи не выходили за матрицу
        for i = 1 : length(neighboors)
%           neighboors(i, :)
            if (neighboors(i, 1) > pxW || neighboors(i, 1) <= 0)
                isOkay(i) = boolean(0);
                neighboors(i, 1) = -1;
            end

            if (neighboors(i, 2) > pxH || neighboors(i, 2) <= 0)
                isOkay(i) = boolean(0);
                neighboors(i, 1) = -1;
            end
%           neighboors(i, :)
%           isOkay(i)
        end

        for i = 1 : length(neighboors)
            if (~isOkay(i))
                continue;
            end

            if (maxReached(neighboors(i, 2), neighboors(i, 1)) ~= 0)
                if (imageGS(neighboors(i, 2), neighboors(i, 1)) == 0)
                    continue;
                end

                sumY = sumY + neighboors(i, 2);
                sumX = sumX + neighboors(i, 1);

                sizeX = sizeX + 1;
                sizeY = sizeY + 1;
```

```
            frontier.push(neighboors(i, :));
            maxReached(neighboors(i, 2), neighboors(i, 1)) = 0;
          end
        end
    end

    breadthX = ceil(sumX / sizeX);
    breadthY = ceil(sumY / sizeY);

end
```

B.3.2 Python

```
def breadthSearch(arrGS, point, arr, folder_path, iii, generateImage, algoType):
  # if (generateImage):
    # arrCopy = arr.copy()
  if (point[0] == -1 or point[1] == -1):
    # if (generateImage):
      # fileName = 'c_breadth_' + algoType + 'image_' + str(iii) + '.png'

      # img = Image.fromarray(arrCopy)
      # img.save(os.path.join(folder_path, fileName))
    return [0, 0]

  # if (generateImage):
    # arrCopy = arr.copy()

  dtotalX = []
  dtotalY = []
  point = tuple(point)
  frontier = deque()
  frontier.append(point)
  reached = set()
  reached.add(point)

  while not len(frontier) == 0:
    current = frontier.popleft()
    neighboors = [
      [current[0] + 1, current[1]],
      [current[0] - 1, current[1]],
      [current[0], current[1] + 1],
      [current[0], current[1] - 1],
    ]

    for next in neighboors:
      if (next[0] < 0):
        next[0] = 0
      if (next[1] < 0):
        next[1] = 0
      if (next[0] >= pxW):
        next[0] = pxW - 1
      if (next[1] >= pxH):
        next[1] = pxH - 1
```

```
    next = tuple(next)

    if (next not in reached):
      if (not(arrGS[next[1]][next[0]][0] > 10 or arrGS[next[1]][next[0]][1] < 255)):
        continue

      # if (generateImage):
        arrCopy[next[1], next[0]] = [255, 0, 0]

      frontier.append(next)
      reached.add(next)
      dtotalX.append(next[0] + 1)
      dtotalY.append(next[1] + 1)

# if (generateImage):
  fileName = 'c_breadth_' + algoType + 'image_' + str(iii) + '.png'

  # img = Image.fromarray(arrCopy)
  # img.save(os.path.join(folder_path, fileName))

if (len(dtotalX) == 0 or len(dtotalY) == 0):
  return [0, 0]

centerX = math.ceil(sum(dtotalX) / len(dtotalX))
centerY = math.ceil(sum(dtotalY) / len(dtotalY))

return [centerX, centerY]
```