|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**Лабораторная работа № 3**

|  |  |
| --- | --- |
| **Тема** Реализация и исследование алгоритмов построения отрезков  **Студент** Якуба Д. В.  **Группа** ИУ7-43  **Оценка (баллы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Преподаватель** Куров А. В. |  |

Москва

2020 г.

Оглавление

[Цель работы 3](#_Toc37206710)

[Техническое задание 3](#_Toc37206711)

[Теоретическая часть 3](#_Toc37206712)

[Описание и реализация алгоритмов 4](#_Toc37206713)

[Алгоритм ЦДА 4](#_Toc37206714)

[Алгоритм, записанный на ЯП Python: 5](#_Toc37206715)

[Алгоритм Брезенхема с действительными коэффициентами 5](#_Toc37206716)

[Алгоритм, записанный на ЯП Python: 6](#_Toc37206717)

[Алгоритм Брезенхема с целыми коэффициентами 7](#_Toc37206718)

[Алгоритм, записанный на ЯП Python: 8](#_Toc37206719)

[Алгоритм Брезенхема с устранением ступенчатости 8](#_Toc37206720)

[Алгоритм, записанный на ЯП Python: 9](#_Toc37206721)

[Алгоритм Ву 10](#_Toc37206722)

[Алгоритм, записанный на ЯП Python: 10](#_Toc37206723)

[Пользовательский интерфейс 11](#_Toc37206724)

[Сравнение визуальных характеристик 19](#_Toc37206725)

[Алгоритм ЦДА 19](#_Toc37206726)

[Алгоритм Брезенхема с действительными коэффициентами 26](#_Toc37206727)

[Алгоритм Брезенхема с целыми коэффициентами 28](#_Toc37206728)

[Алгоритм Брезенхема с устранением ступенчатости 30](#_Toc37206729)

[Алгоритм Ву 32](#_Toc37206730)

[Алгоритм Tkinter.canvas.create\_line 34](#_Toc37206731)

[Все алгоритмы на единой плоскости 36](#_Toc37206732)

[Исследование временных характеристик 37](#_Toc37206733)

# Цель работы

Реализация алгоритмов построения отрезков по методу цифрового дифференциального анализатора (ЦДА), алгоритмов Брезенхема (действительного, целочисленного и с устранением ступенчатости) и алгоритма Ву и исследование их характеристик и сравнение полученных результатов.

# Техническое задание

1.Реализовать алгоритмы ЦДА, Брезенхема (действительный, целочисленный, с устранением ступенчатости) и Ву. Реализовать возможность построения задаваемых пользователем одиночных отрезков выбранным алгоритмом.

2.Сравнить визуально отрезки, построенные в соответствии с каждым алгоритмом, а также с отрезком, построенным процедурой языка высокого уровня. Проверить попадание отрезка в заданную конечную точку.

3.Определить время, затрачиваемое на построение отрезка по каждому из алгоритмов.

# Теоретическая часть

Необходимость разработки специальных алгоритмов построения геометрических объектов объясняется тем, что сами эти объекты имеют непрерывную (аналоговую) природу, а изображение строится, как правило, на экране растрового дисплея, то есть получаемое изображение носит дискретный характер. Это означает, например, что нельзя построить непосредственно отрезок, соединяющий две заданные точки экрана.

Процесс нахождения пикселей, наилучшим образом аппроксимирующих заданный отрезок, называется разложением отрезка в растр.

Решение поставленной задачи очевидно лишь для трех типов отрезков: горизонтальных, вертикальных и наклоненных под углом в 45.

Перед рассмотрением алгоритмов построения отрезков необходимо сформулировать наиболее общие и простые требования, предъявляемые к таким алгоритмам:

1. Отрезки должны выглядеть прямыми. Отрезки должны начинаться и заканчиваться в заданных точках.

2. Яркость вдоль отрезка должна быть постоянной и не зависеть от длины и наклона.

3. Алгоритмы должны работать быстро.

Первое требование в силу дискретной природы растрового дисплея выполнено всегда быть не может. Можно лишь добиться того, что визуально отрезок будет восприниматься прямым. Решение этой задачи может достигаться путем увеличения разрешающей способности экрана дисплея и применения методов устранения ступенчатости.

Второму требованию удовлетворяют также только горизонтальные, вертикальные и наклоненные под углом в 45 отрезки. Однако вертикальные и горизонтальные отрезки по сравнению с отрезками, расположенными под 45, будут выглядеть ярче, так как расстояние между соседними пикселями у них меньше, чем у наклонных отрезков. Обеспечение постоянной яркости вдоль отрезка требует высвечивания очередного пикселя яркостью, зависящей от расстояния между пикселями, вычисление которого производится с использованием операций извлечения квадратного корня и умножения (возводить в квадрат лучше путем умножения числа самого на себя). Использование этих операций существенно замедляет работу алгоритма, поэтому второе требование остается, как правило, невыполненным.

Удовлетворение третьего требования достигается путем сведения к минимуму вычислительных операций, использования операций над целочисленными данными, а также реализацией алгоритмов на аппаратном или микропрограммном уровне.

Многие алгоритмы вычерчивания отрезков и кривых используют пошаговый принцип, суть которого состоит в том, что координаты высвечиваемого пикселя определяются каждый раз на очередном шаге вычислений, а не вычисляются заранее для всех пикселей. Результат вычислений на текущем шаге зависит от результатов, полученных на предыдущем шаге.

# Описание и реализация алгоритмов

## Алгоритм ЦДА

Данный метод использует достаточно общий принцип, известный в математике: изучение какого-либо явления на основе дифференциального уравнения или системы таких уравнений, описывающей это явление. В данном случае разложение отрезка в растр состоит в решении дифференциального уравнения, описывающего этот процесс.

Для прямой линии имеем:

Решение представляется в следующем виде:

Где *x1, y1, x2, y2* – концы разлагаемого отрезка, и *y­I –* начальное значение для очередного шага вдоль отрезка. Фактически такое уравнение представляет собой рекуррентное соотношение для последовательных значений *y* вдоль нужного отрезка. В простом ЦДА либо , либо (большее из приращений) выбирается в качестве единицы растра.

### Алгоритм, записанный на ЯП Python:

def DDAline(xStart, xEnd, yStart, yEnd, color):  
 pointsArray = []  
 if xStart == xEnd and yStart == yEnd:  
 pointsArray.append((color, (xStart, yStart)))  
 return pointsArray  
  
 deltaX = xEnd - xStart  
 deltaY = yEnd - yStart  
  
 trX = abs(deltaX)  
 trY = abs(deltaY)  
  
 length = trX if trX > trY else trY  
  
 deltaX /= length  
 deltaY /= length  
  
 curX = xStart  
 curY = yStart  
  
 for i in range(length):  
 pointsArray.append((color, (niceRound(curX), niceRound(curY))))  
 curX += deltaX  
 curY += deltaY  
 return pointsArray

## Алгоритм Брезенхема с действительными коэффициентами

Работа алгоритма Брезенхема основывается на использовании понятия ошибка. Ошибкой здесь называется расстояние между действительным положением отрезка и ближайшим пикселем сетки растра, который аппроксимирует отрезок на очередном шаге.

На каждом шаге вычисляется величина ошибки и в зависимости от полученного значения выбирается пиксель, ближе расположенный к идеальному отрезку. Поскольку при реализации алгоритма на ЭВМ удобнее анализировать не само значение ошибки, а ее знак, то истинное значение ошибки смещается на -0,5.

Поскольку на первом шаге высвечивается пиксел с начальными координатами, то для него ошибка равняется 0, поэтому задаваемое предварительно значение этой ошибки

Фактически является ошибкой для следующего шага.

В общем алгоритме Брезенхема большее по модулю из приращений принимается равным шагу растра, то есть единице, причем знак приращения совпадает со знаком разности конечной и начальной координат отрезка:

Где *xe, ye –* координаты конца отрезка; *xs, ys –* координаты начала отрезка, а sign - кусочно-постоянная функция действительного аргумента.

Значение другой координаты идеального отрезка для следующего шага определяется как , поскольку приращение ординаты совпадает с величиной одного катета прямоугольного треугольника, а другой катет равен шагу сетки растра, то есть единице.

Ошибка на очередном шаге вычисляется как

В зависимости от полученного значения ошибки выбирается пиксел с той же ординатой (при ) или пиксел с ординатой, на единицу большей, чем у предыдущего пиксела (при ).

Поскольку предварительное значение ошибки вычисляется заранее, то есть вычислено на предыдущем шаге, то во втором случае останется только вычесть единицу из значения ошибки: , так как в этом случае , что не учитывалось при расчете.

### Алгоритм, записанный на ЯП Python:

def realBresenham(xStart, xEnd, yStart, yEnd, color):  
 pointsArray = []  
 if xStart == xEnd and yStart == yEnd:  
 pointsArray.append((color, (xStart, yStart)))  
 return pointsArray  
  
 deltaX = xEnd - xStart  
 deltaY = yEnd - yStart  
  
 stepX = int(sign(deltaX))  
 stepY = int(sign(deltaY))  
  
 deltaX = abs(deltaX)  
 deltaY = abs(deltaY)  
  
 if deltaX < deltaY:  
 deltaX, deltaY = deltaY, deltaX  
 flag = True  
 else:  
 flag = False  
  
 tngModule = deltaY / deltaX  
  
 acc = tngModule - 0.5  
 curX = xStart  
 curY = yStart  
  
 for i in range(deltaX):  
 pointsArray.append((color, (curX, curY)))  
 if flag:  
 if acc >= 0:  
 curX += stepX  
 acc -= 1  
 curY += stepY  
 acc += tngModule  
 else:  
 if acc >= 0:  
 curY += stepY  
 acc -= 1  
 curX += stepX  
 acc += tngModule  
 return pointsArray

## Алгоритм Брезенхема с целыми коэффициентами

Для оптимизации скорости работы алгоритма Брезенхема с действительными коэффициентами хотелось бы избавиться в нём от дробей, ведь в таком случае мы сможем отказаться от операции математического округления, что должно дать ощутимую прибавку к скорости работы алгоритма. Приведенный алгоритм легко преобразуется к целочисленному варианту. Для этого значение ошибки запишем в виде: и, умножив обе части этого равенства на , получим:

Обозначив , окончательно получим:

(В соответствии с этим выражением должно вычисляться теперь

начальное значение ошибки).

Тогда подсчет нового значения ошибки в «действительном» алгоритме будет производиться по формулам:

### Алгоритм, записанный на ЯП Python:

def digitBresenham(xStart, xEnd, yStart, yEnd, color):  
 pointsArray = []  
  
 deltaX = xEnd - xStart  
 deltaY = yEnd - yStart  
  
 stepX = int(sign(deltaX))  
 stepY = int(sign(deltaY))  
  
 deltaX = abs(deltaX)  
 deltaY = abs(deltaY)  
  
 if deltaX < deltaY:  
 deltaX, deltaY = deltaY, deltaX  
 flag = True  
 else:  
 flag = False  
  
 acc = deltaY + deltaY - deltaX  
 curX = xStart  
 curY = yStart  
  
 for i in range(deltaX):  
 pointsArray.append((curColorLines, (curX, curY)))  
  
 if flag:  
 if acc >= 0:  
 curX += stepX  
 acc -= (deltaX + deltaX)  
 curY += stepY  
 acc += deltaY + deltaY  
 else:  
 if acc >= 0:  
 curY += stepY  
 acc -= (deltaX + deltaX)  
 curX += stepX  
 acc += deltaY + deltaY

## Алгоритм Брезенхема с устранением ступенчатости

Алгоритм Брезенхема с устранением ступенчатости отлично применим к геометрическим фигурам (многоугольникам) с однородной заливкой. Заранее стоит отметить, что принципиально устранить ступенчатость невозможно. Но можно несколько сгладить данный эффект, используя интенсивности цвета отображаемых пикселов. Идея состоит в сглаживании резких переходов от ступени к ступени. Сглаживание основывается на том, что при наличии нескольких оттенков цвета внешний вид отрезка улучшается путем размытия его краев. Для этого интенсивность пикселя устанавливается в зависимости от площади части пиксела, находящейся под отрезком. Уровень выбирается пропорционально площади части пикселя, находящейся под «идеальной» прямой. При рассмотрении площади пиксель считается квадратом со стороной 1, прямая делит пиксель на 2 части и площадь, находящаяся под отрезком, будет лежать в интервале (1; 0), что показывает интенсивность отображаемого цвета. Приращение площади можно связать с приращением функции. Таким образом начальная площадь равняется 1/2, а следующее значение определяется по формуле:

Если значение вычисленной площади больше единицы, то вычитаем из полученного выражения единицу и переходим к рассмотрению площади пикселя выше.

### Алгоритм, записанный на ЯП Python:

def stepRemovalBresenham(xStart, xEnd, yStart, yEnd, color):  
 pointsArray = []  
 if xStart == xEnd and yStart == yEnd:  
 pointsArray.append(color, (xStart, yStart)))  
 return pointsArray  
  
 deltaX = xEnd - xStart  
 deltaY = yEnd - yStart  
  
 stepX = int(sign(deltaX))  
 stepY = int(sign(deltaY))  
  
 deltaX = abs(deltaX)  
 deltaY = abs(deltaY)  
  
 if deltaX < deltaY:  
 deltaX, deltaY = deltaY, deltaX  
 flag = True  
 else:  
 flag = False  
  
 tngModule = deltaY / deltaX  
  
 acc = 1 / 2  
 correction = 1 - tngModule  
 curX = xStart  
 curY = yStart  
  
 for i in range(deltaX):   
 pointsArray.append((setItentity(color, acc), (curX, curY)))  
  
 if flag:  
 if acc >= correction:  
 curX += stepX  
 acc -= correction + tngModule  
 curY += stepY  
 acc += tngModule  
 else:  
 if acc >= correction:  
 curY += stepY  
 acc -= correction + tngModule  
 curX += stepX  
 acc += tngModule  
 return pointsArray

## Алгоритм Ву

Алгоритм Ву — это алгоритм разложения отрезка в растр со сглаживанием

Алгоритм основа на следующих принципах:

1. Отрезок высвечивается толщиной в два пикселя

2. Суммарная интенсивность двух высвечиваемых на каждом шаге пикселей постоянна:

3. Сглаживание осуществляется за счёт перераспределения на каждом шаге интенсивности между двумя пикселями

4. Интенсивность пикселя выбирается или определяется в зависимости от расстояния между пикселем и точкой, расположенном на идеальном отрезке: чем ближе пиксель находится к идеальному отрезку, тем выше интенсивность его цвета.

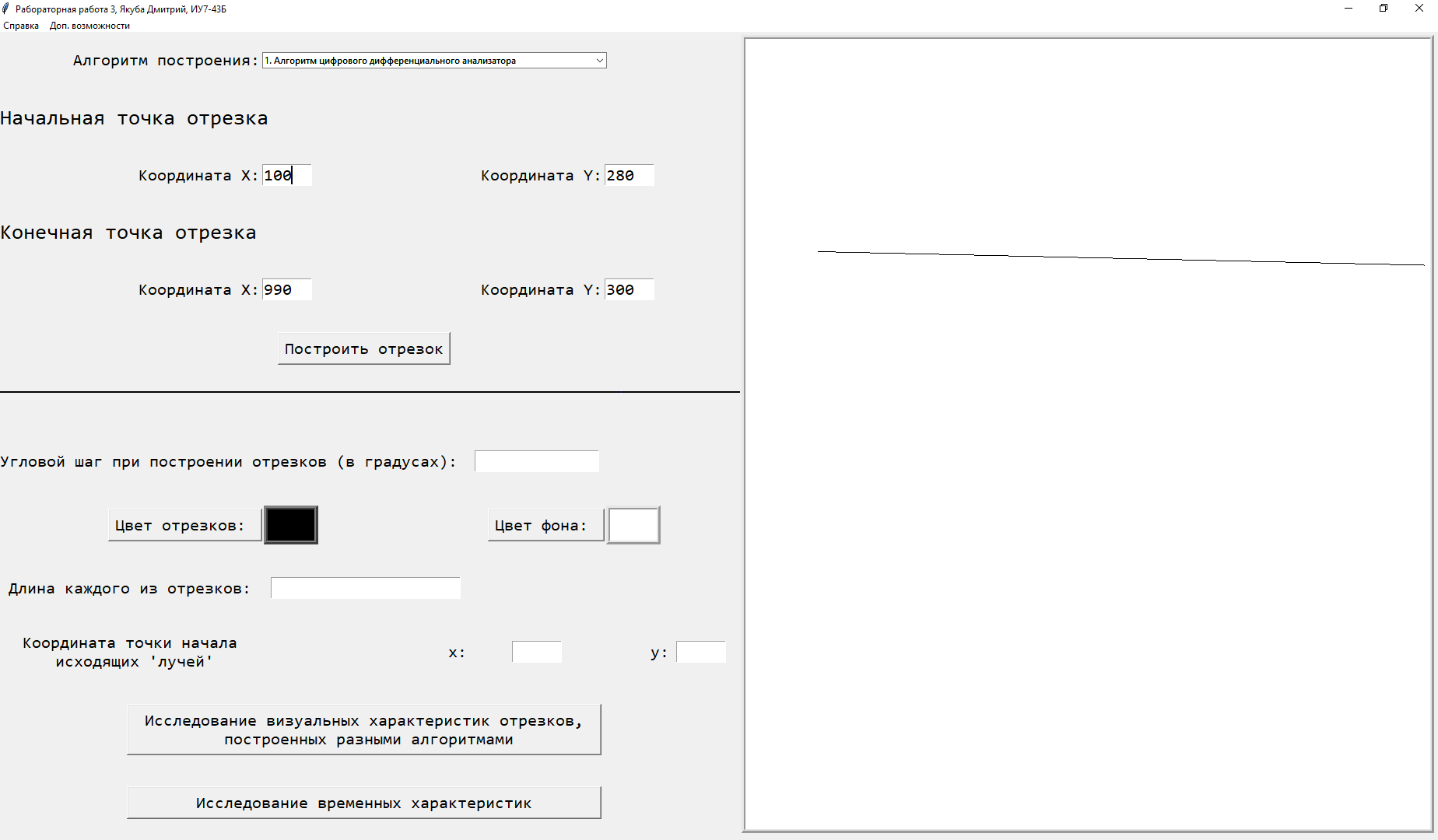
### Алгоритм, записанный на ЯП Python:

def WuAlg(xStart, xEnd, yStart, yEnd, color):  
 pointsArray = []  
 if xStart == xEnd and yStart == yEnd:  
 pointsArray.append((color, (xStart, yStart)))  
 return pointsArray  
  
 deltaX = xEnd - xStart  
 deltaY = yEnd - yStart  
  
 stepX = int(sign(deltaX))  
 stepY = int(sign(deltaY))  
  
 deltaX = abs(deltaX)  
 deltaY = abs(deltaY)  
  
 if deltaX < deltaY:  
 deltaX, deltaY = deltaY, deltaX  
 flag = True  
 else:  
 flag = False  
  
 tngModule = deltaY / deltaX  
  
 acc = -1  
 curX = xStart  
 curY = yStart  
  
 for i in range(deltaX):  
 pointsArray.append((setItentity(color, 1 + acc), (curX, curY)))  
  
 pointsArray.append((setItentity(color, -acc), (curX, curY + stepY)))  
 if flag:  
 if acc >= 0:  
 curX += stepX  
 acc -= 1  
 curY += stepY  
 acc += tngModule  
 else:  
 if acc >= 0:  
 curY += stepY  
 acc -= 1  
 curX += stepX  
 acc += tngModule  
 return pointsArray

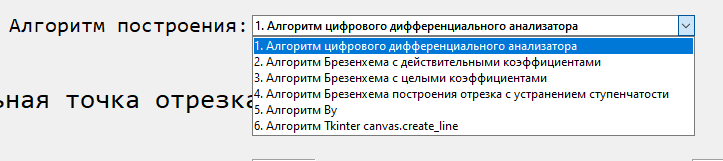
# Пользовательский интерфейс

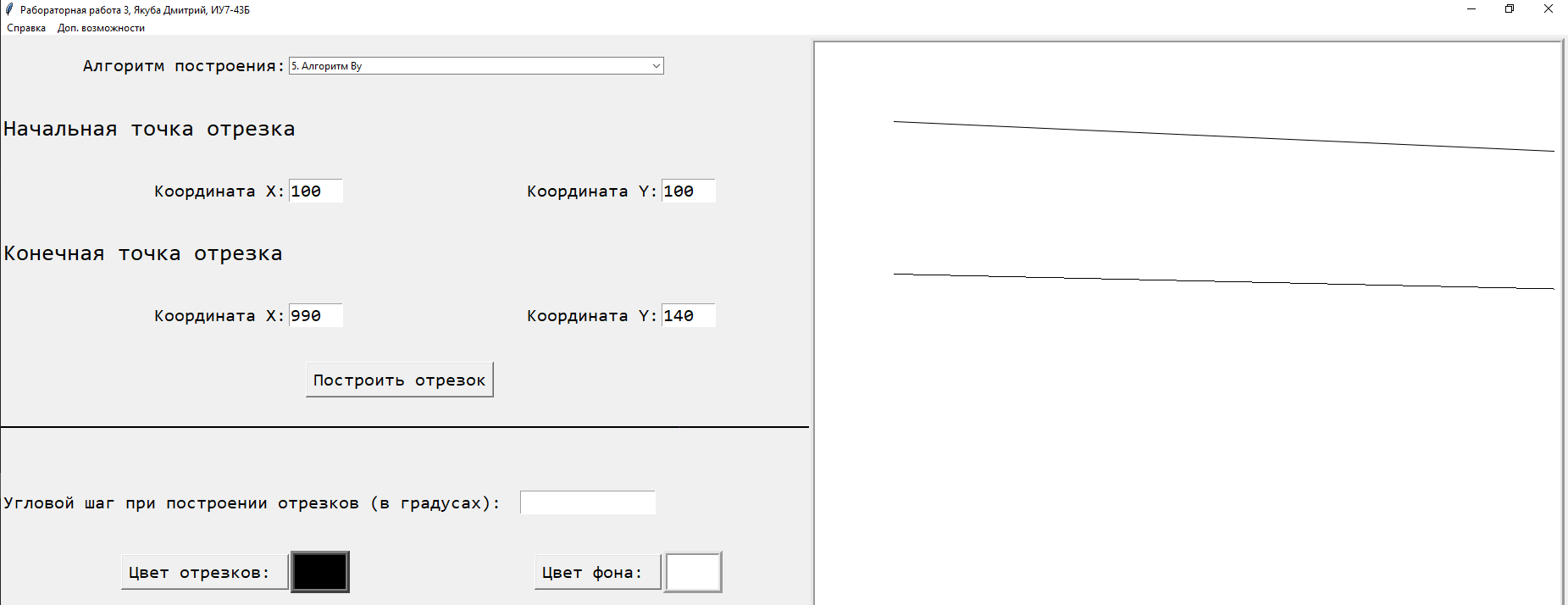


Пользователю достаточно просто координаты для построения отрезков в поля «Начальная точка» - «Координата Х», «Начальная точка» - «Координата Y» и так далее и нажать на кнопку «построить отрезок».

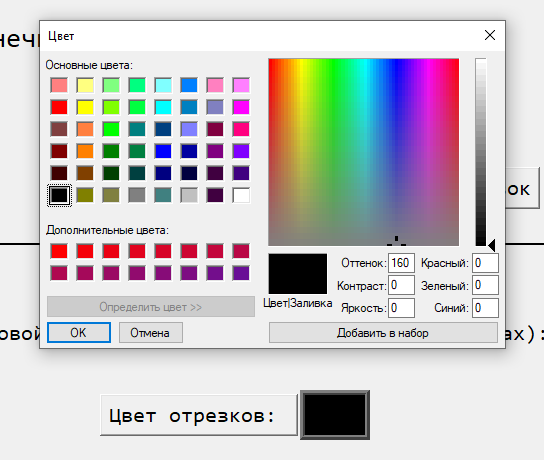


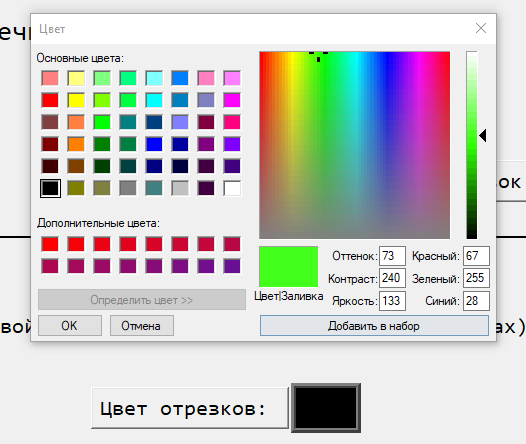
Для того, чтобы изменить алгоритм построения отрезков, достаточно воспользоваться «шторкой» «Алгоритм построения»:

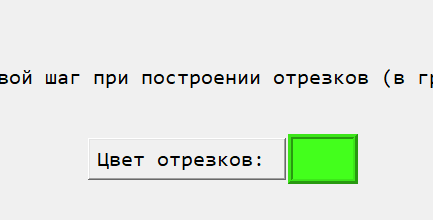




Чтобы выбрать цвет фона или цвет рисования достаточно воспользоваться кнопками «Цвет отрезков» или «Цвет фона»:

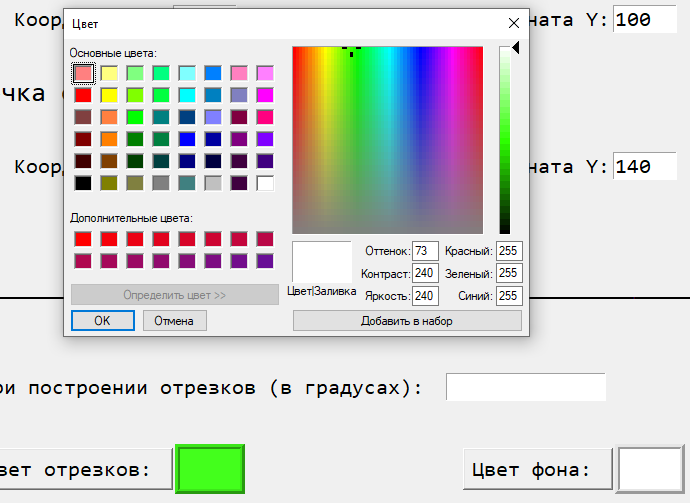


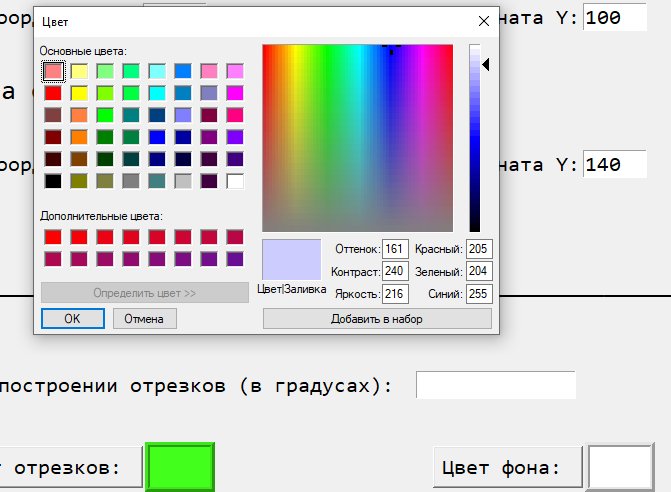


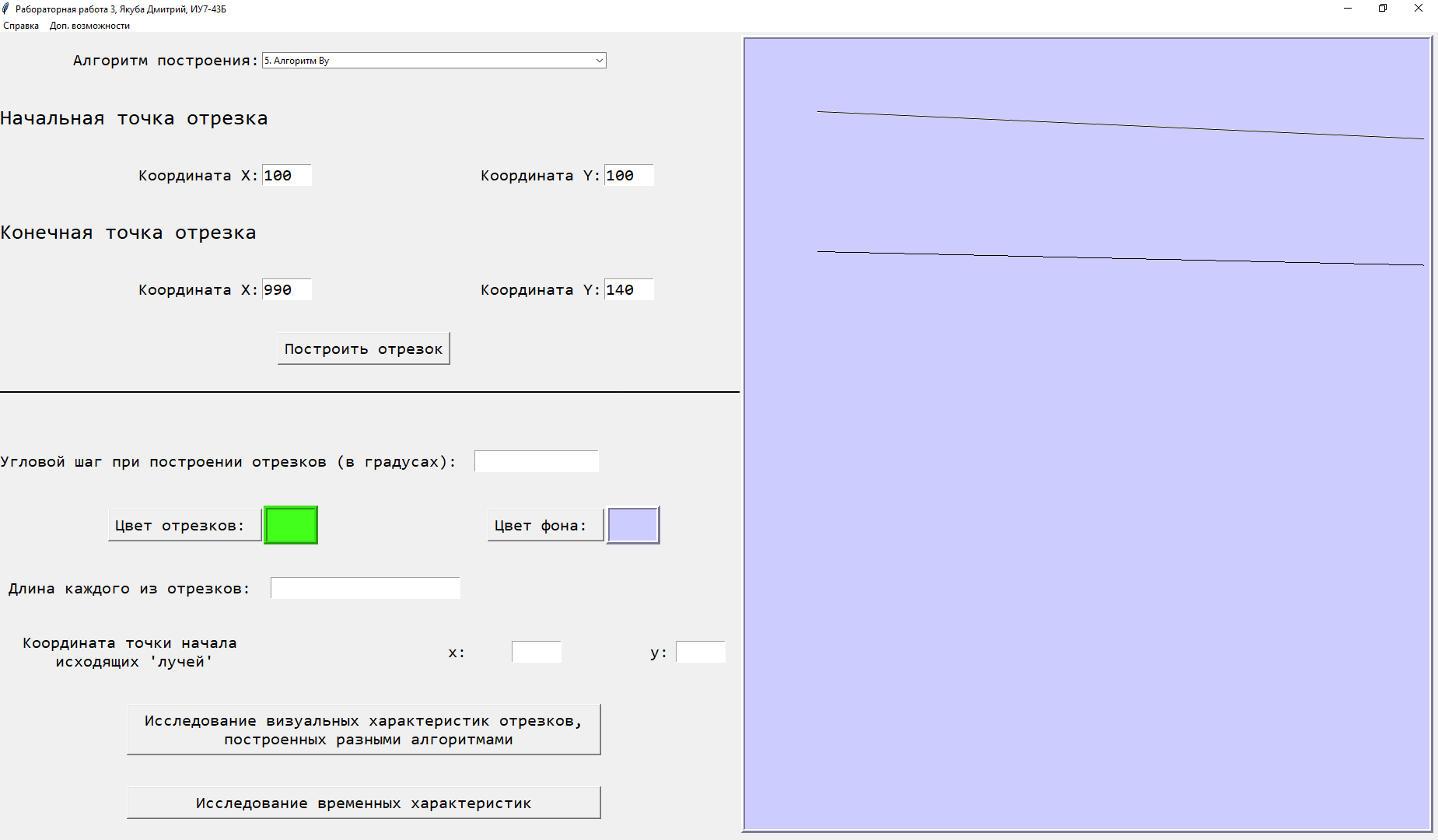


После чего можно строить отрезок любым из заданных цветов.

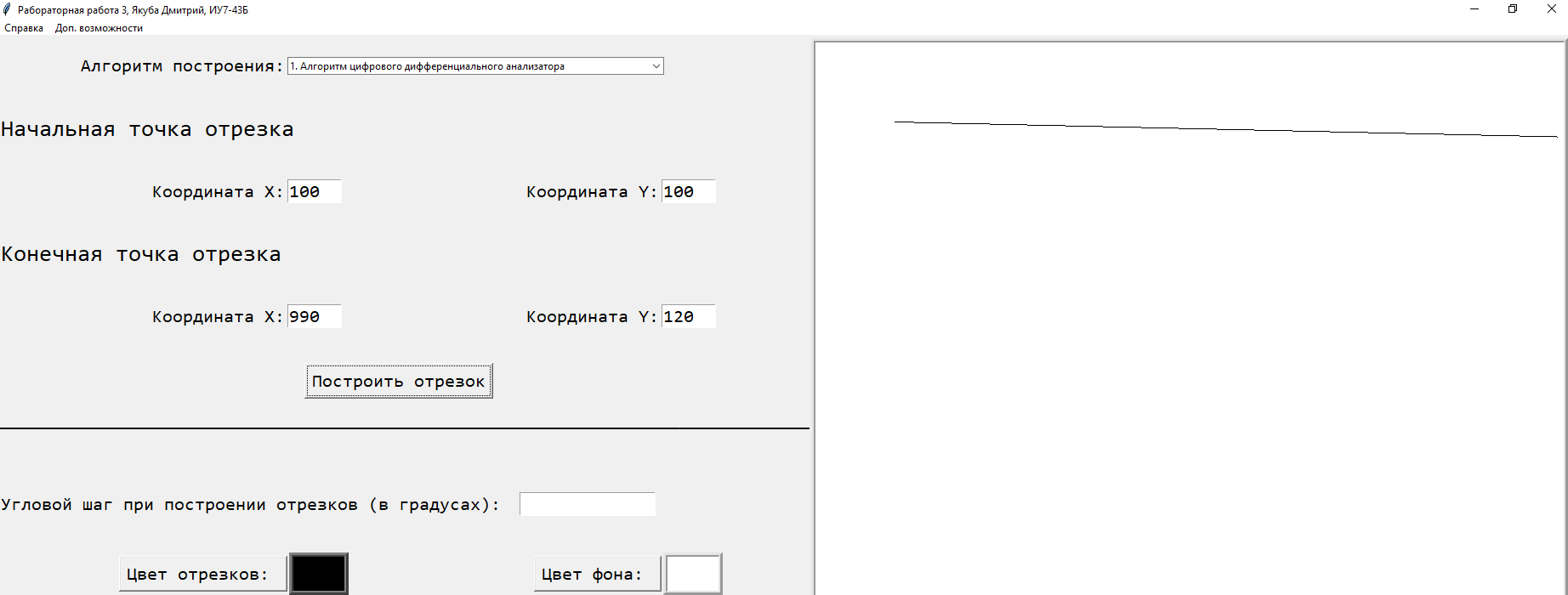
Также стоит продемонстрировать изменение цвета фона:

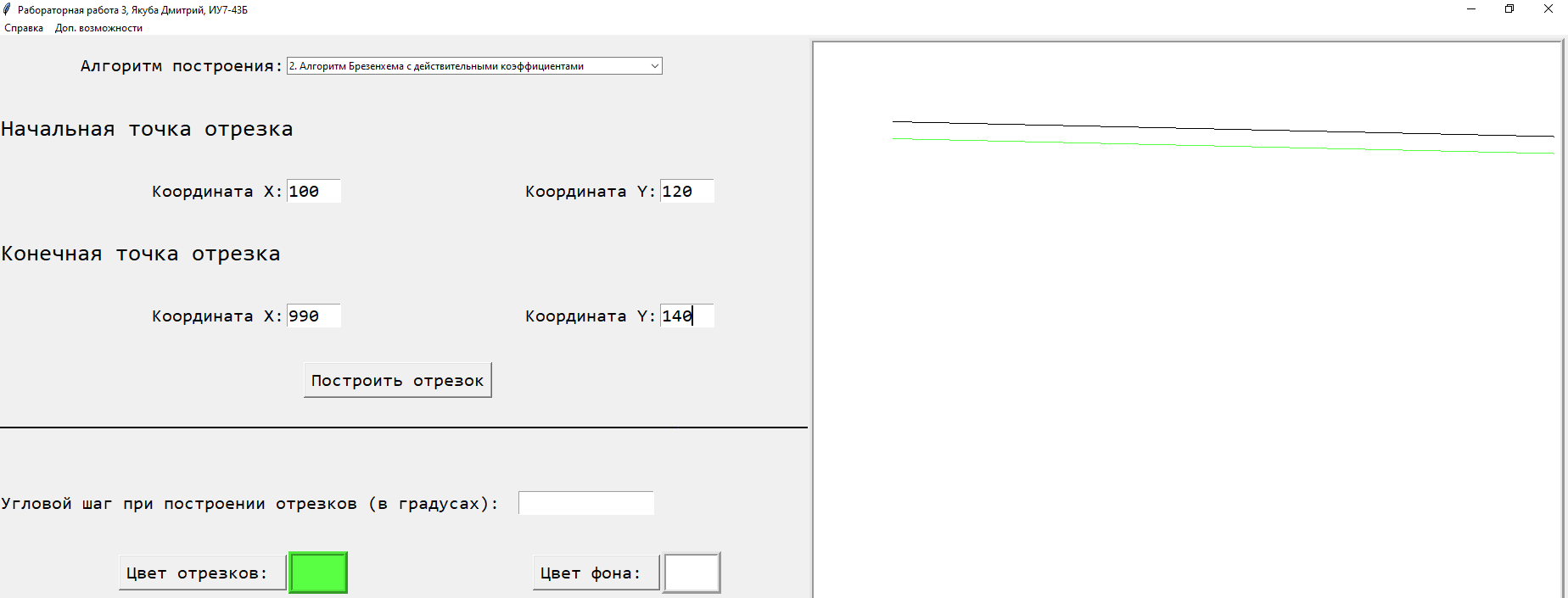


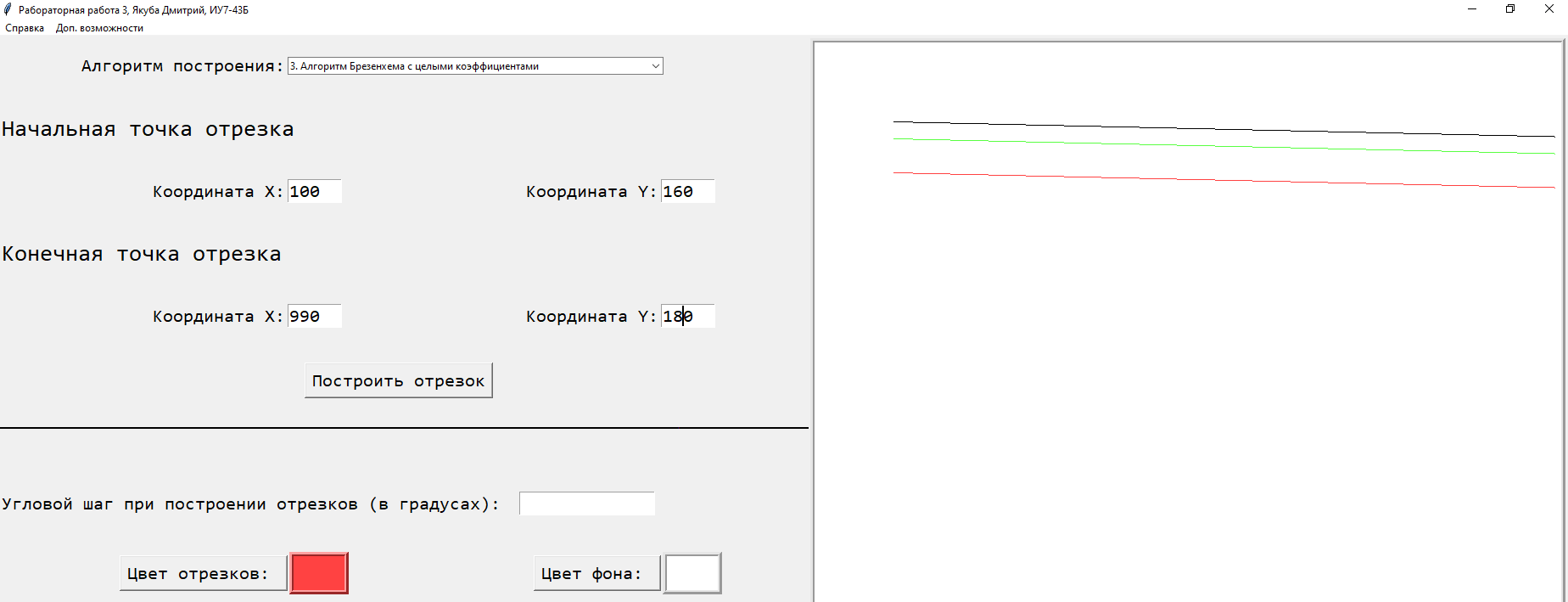


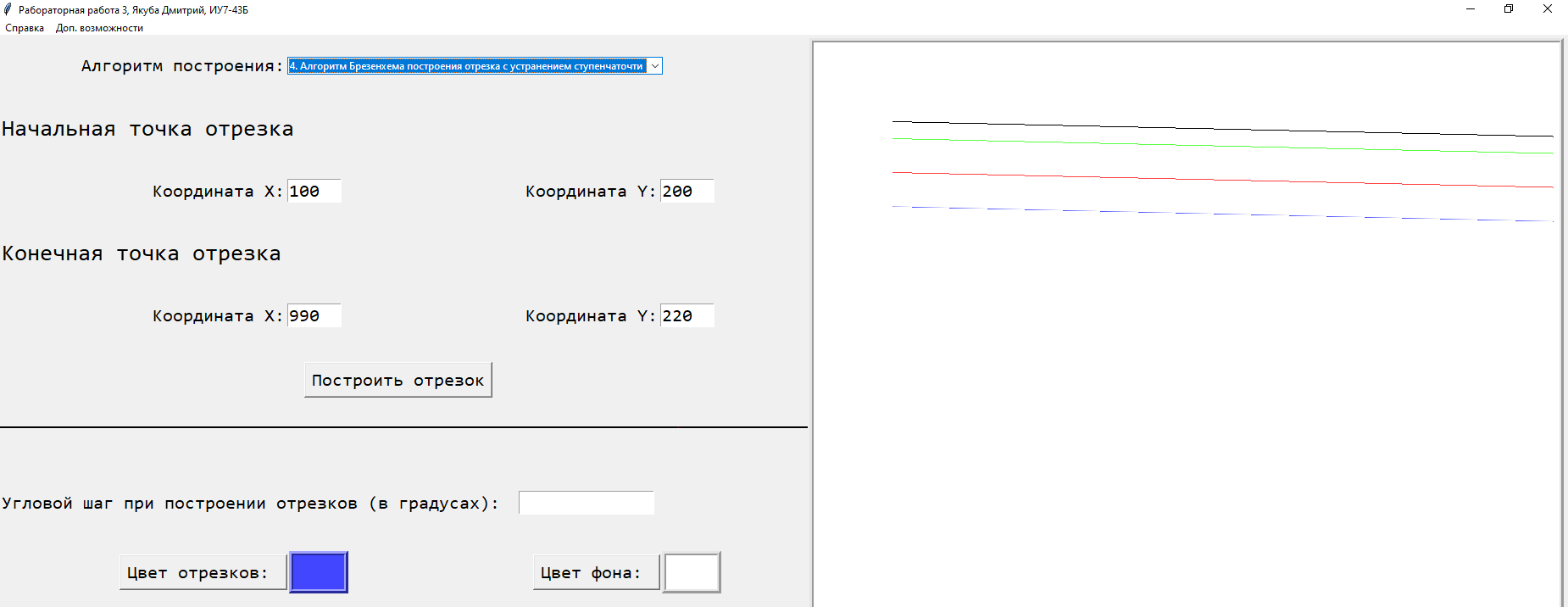


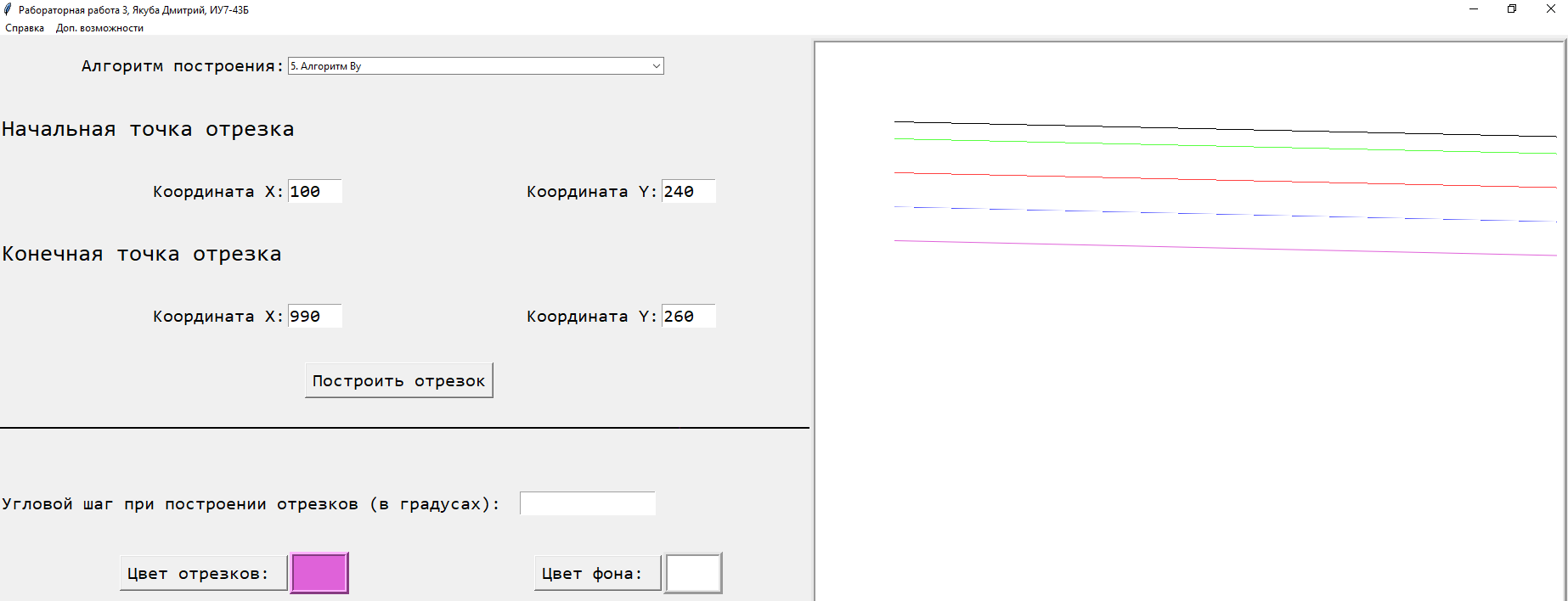
Продемонстрируем работу каждого из алгоритма построения отрезка различными цветами:

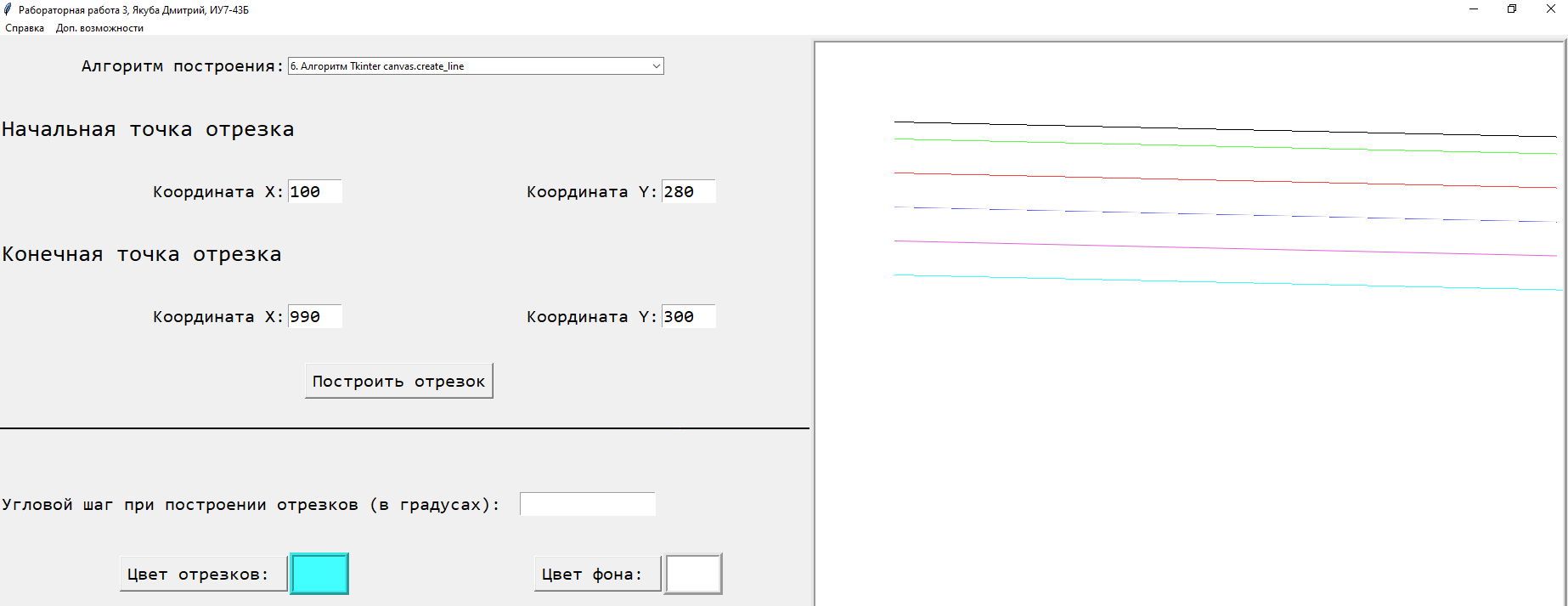




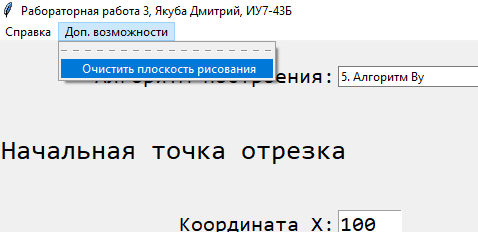




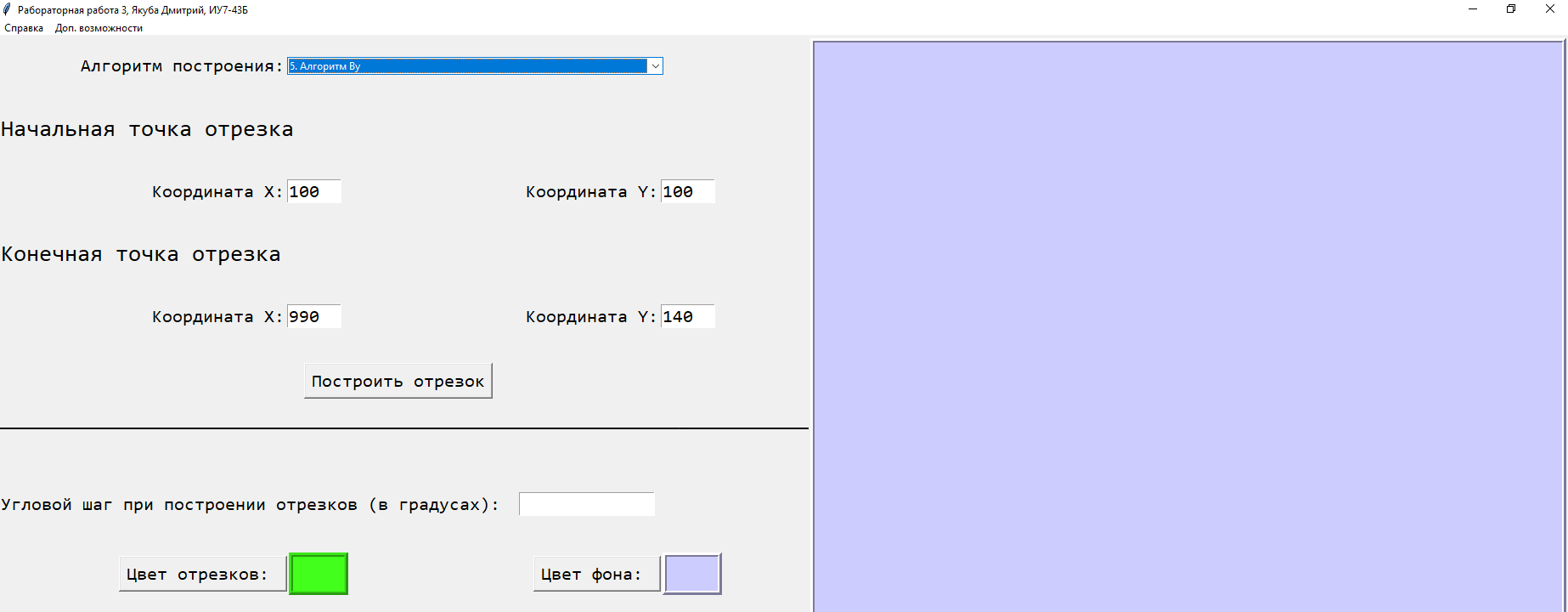




Во вкладке «Доп. Возможности» также имеется возможность очистить поле вывода:

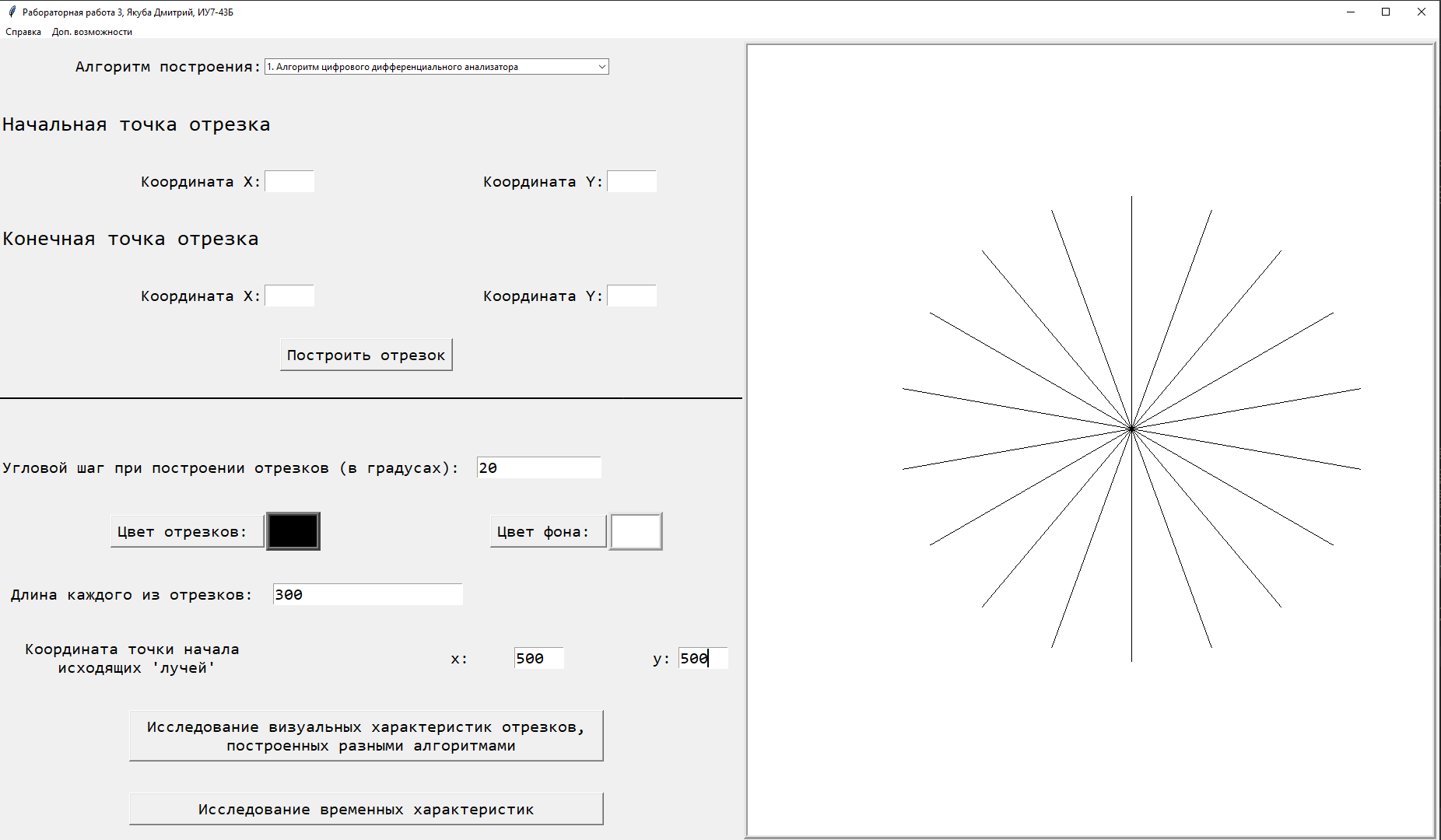


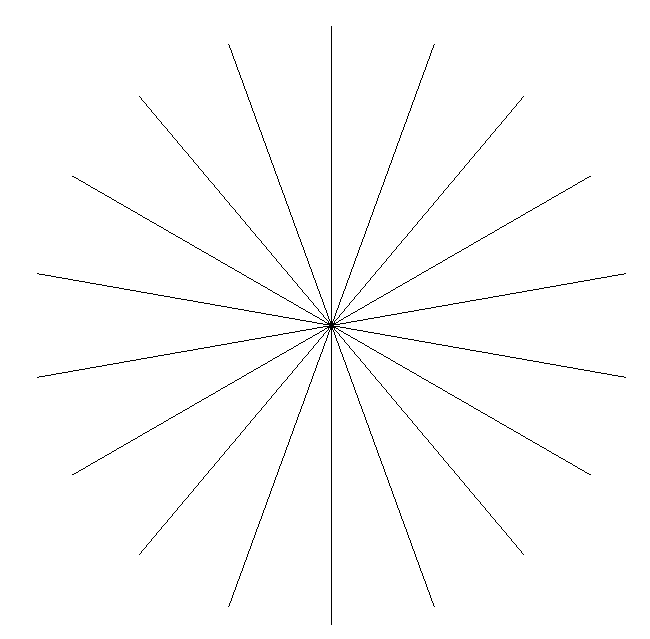


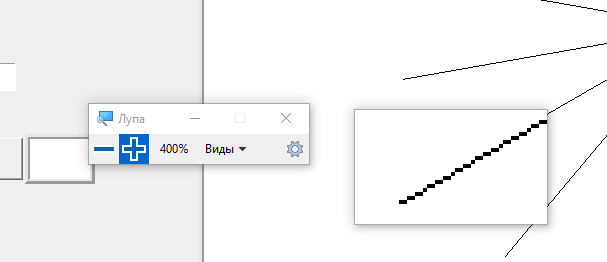
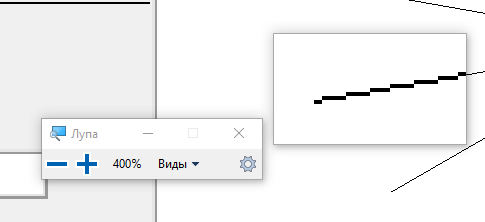


# Сравнение визуальных характеристик

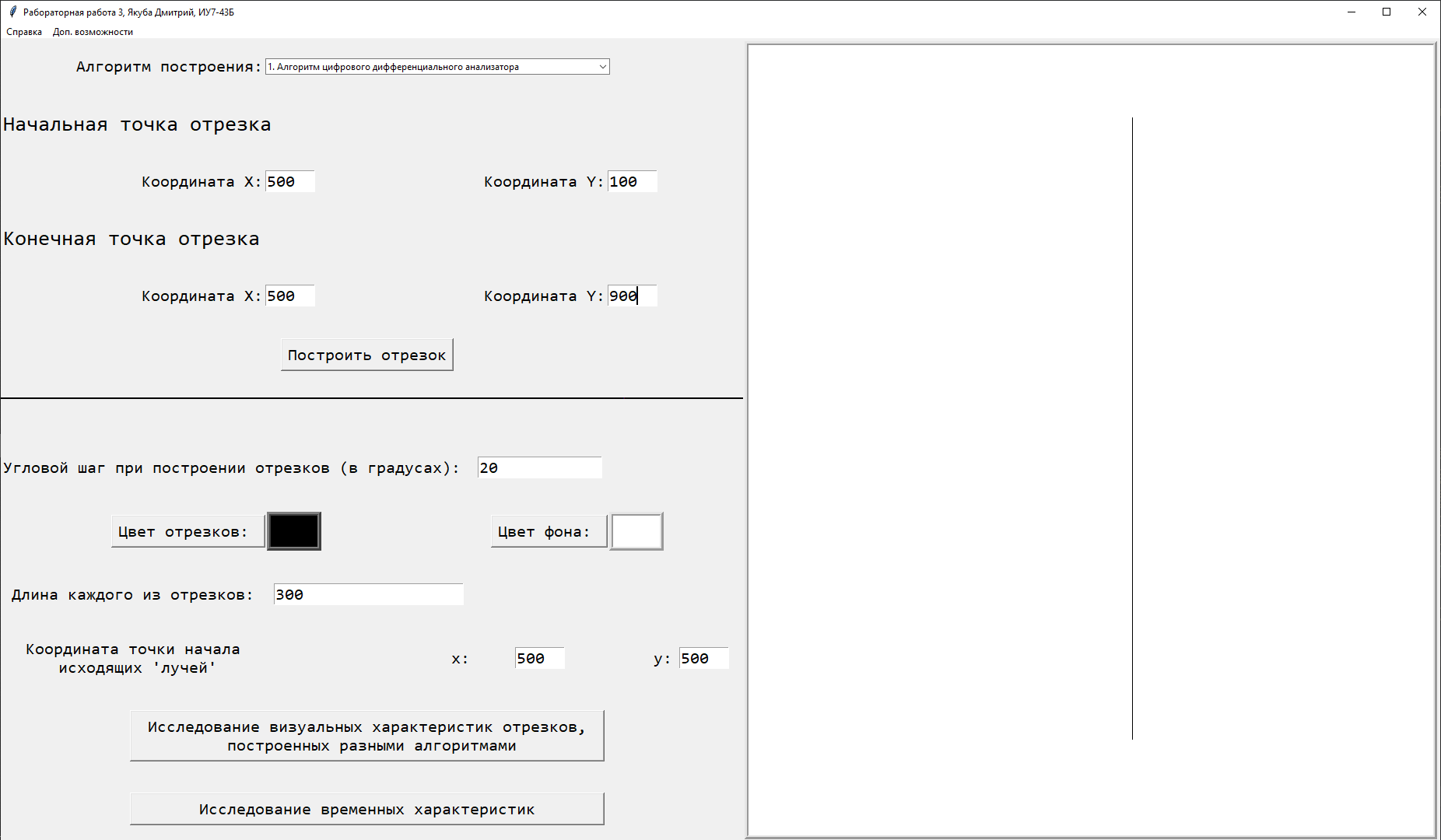
## Алгоритм ЦДА



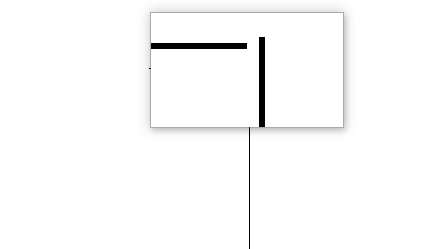


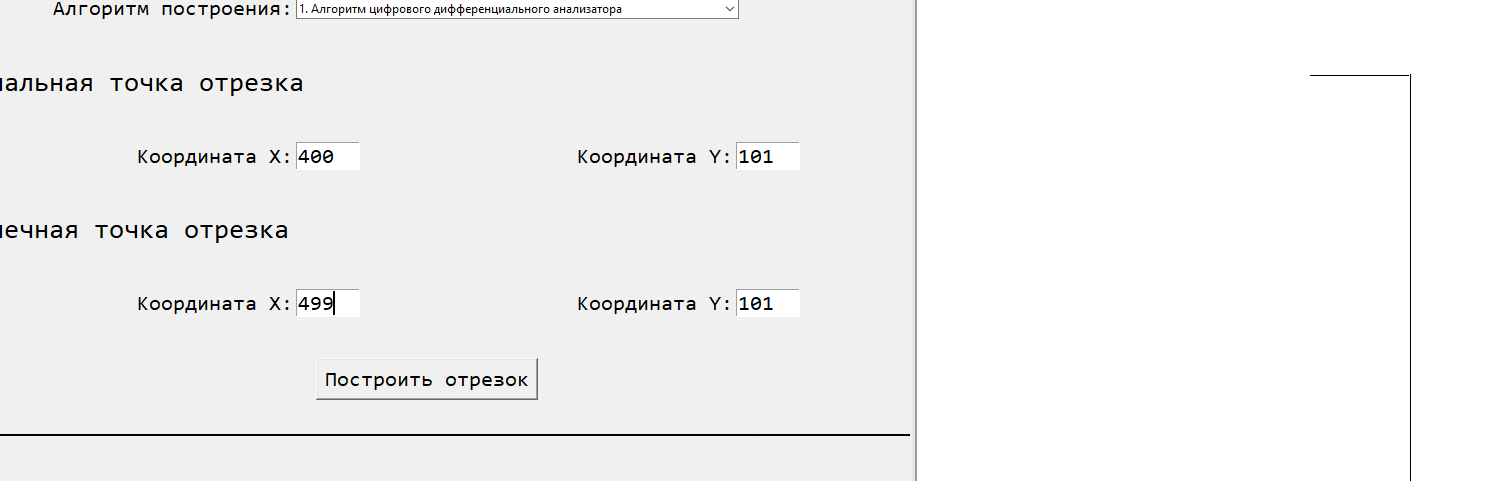
 

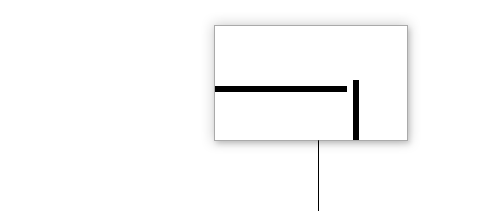
На лекции также было сказано, что данный алгоритм порой может не доходить до точки назначения. Проверим данный момент следующим способом: построим отрезок, параллельный оси Y: x = 500 и проследим за тем, доходит ли до указанного отрезка данный алгоритм.





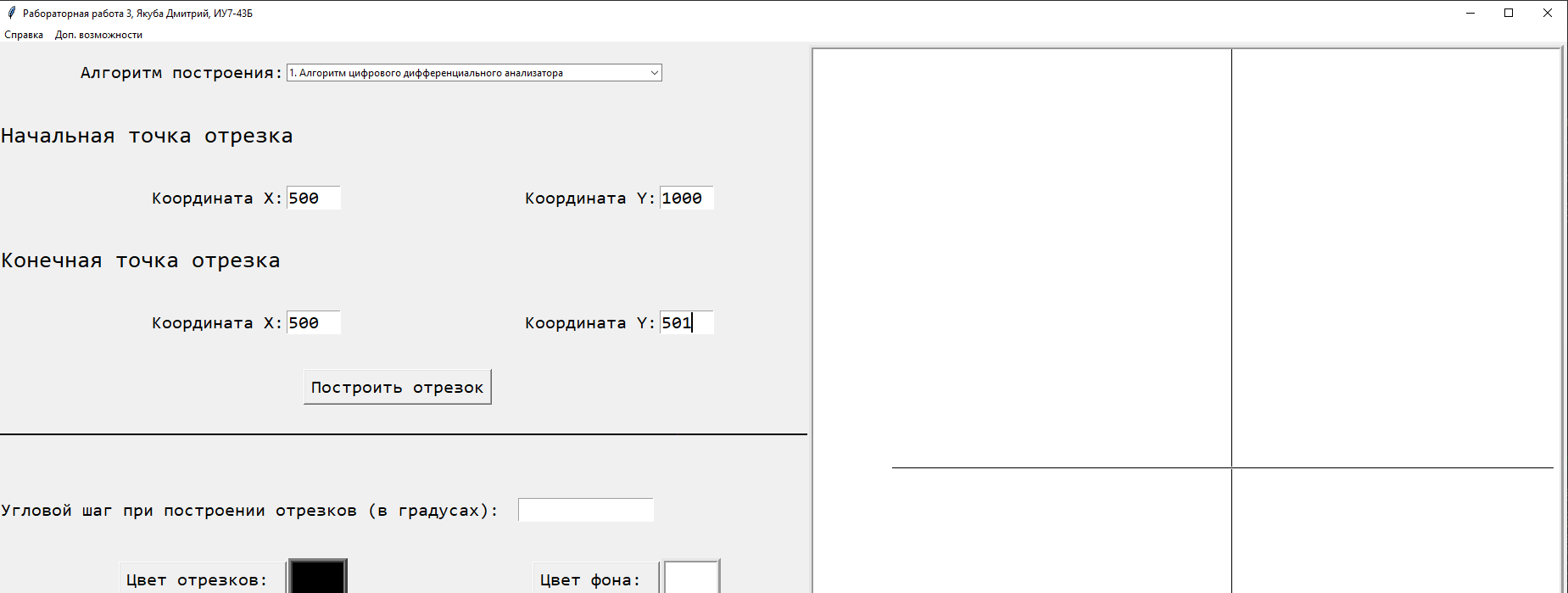


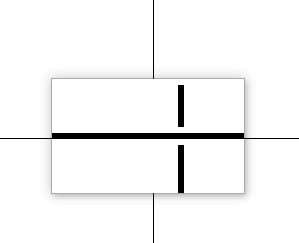




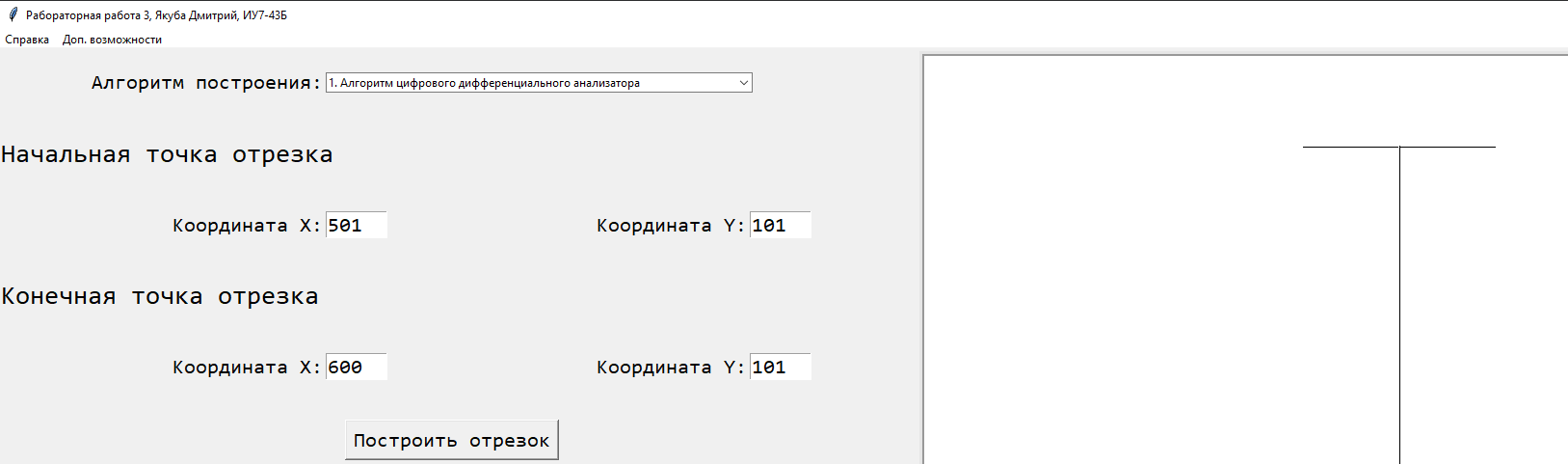
Сразу заметно, что отрезку не хватает ровно одного пикселя, чтобы достичь заданной точки.

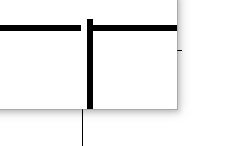
Заметим, то такой же эффект наблюдается и в следующем случае:





Проверим совпадение начала:

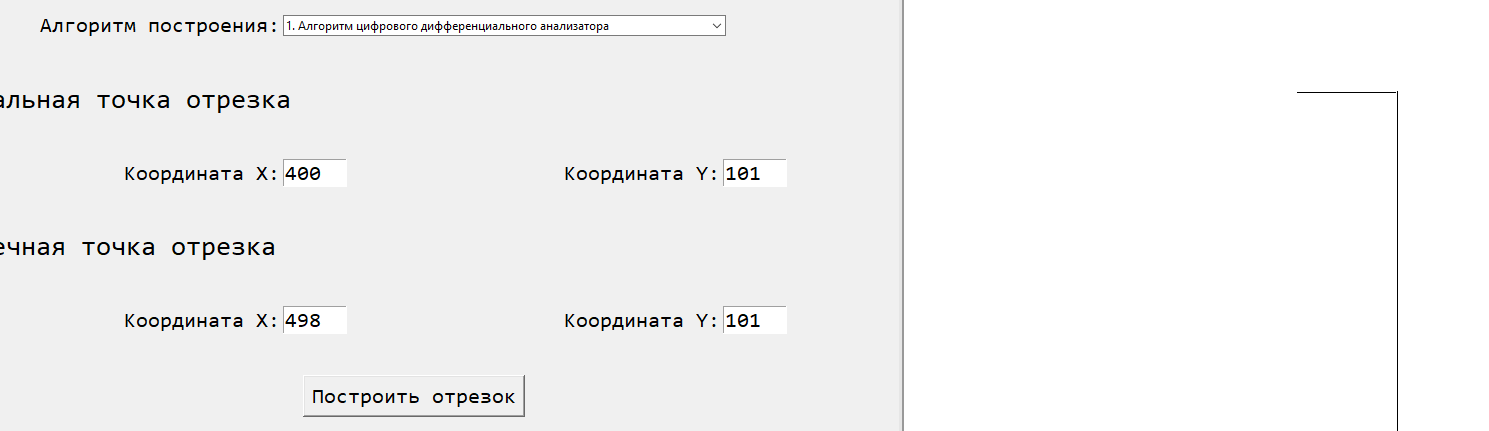


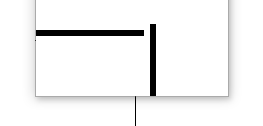


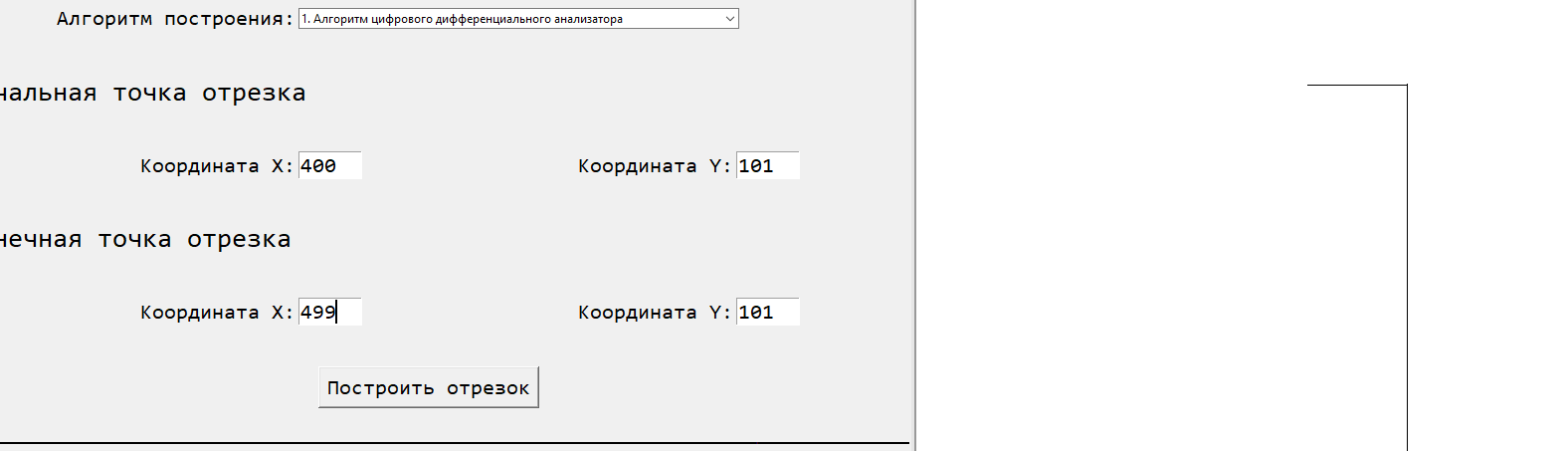
Видно, что начало попадает в заданное значение.  
Разберёмся в данной проблеме. Если отойти от канонов, к которым мы пришли на лекции, а именно, от цикла по *i от 1 до l+1* и прийти к следующей конструкции:

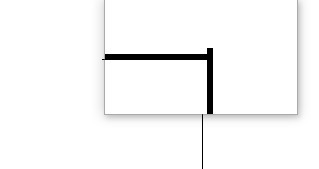


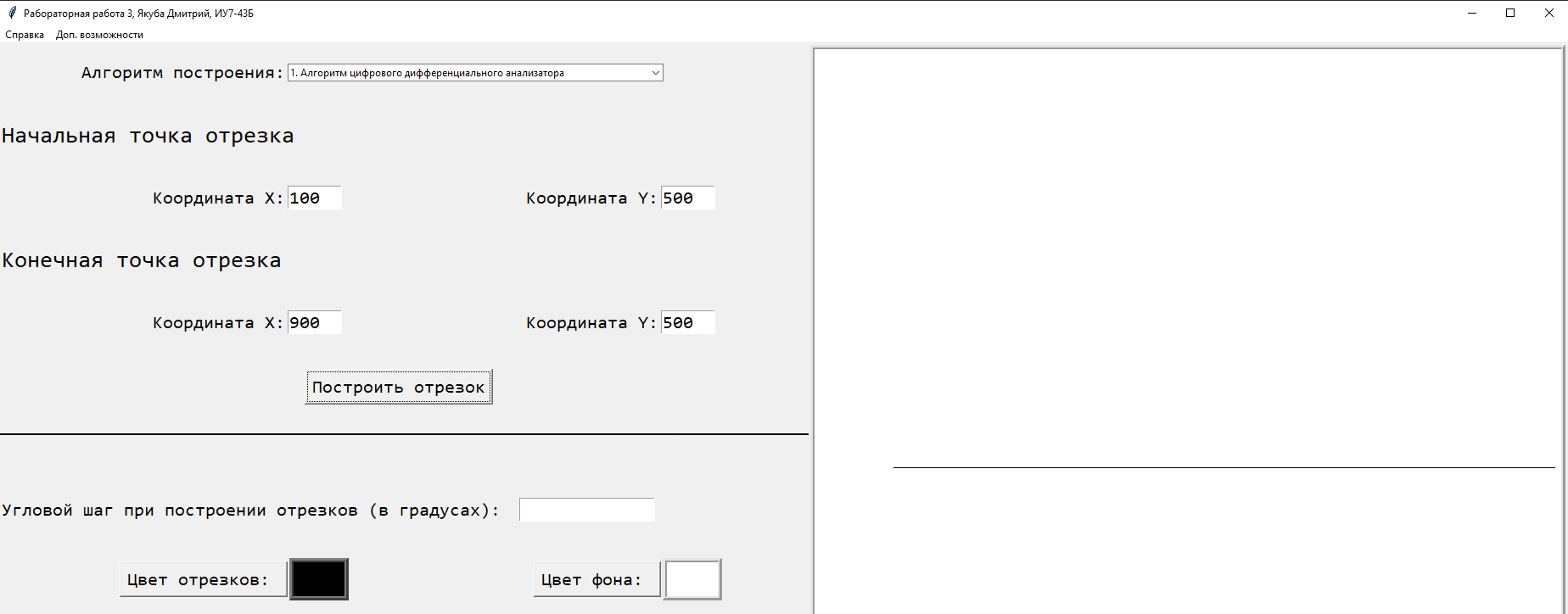
То есть к циклу по *i от 0 до l + 1*, то отрезок дойдёт до конечной точки:



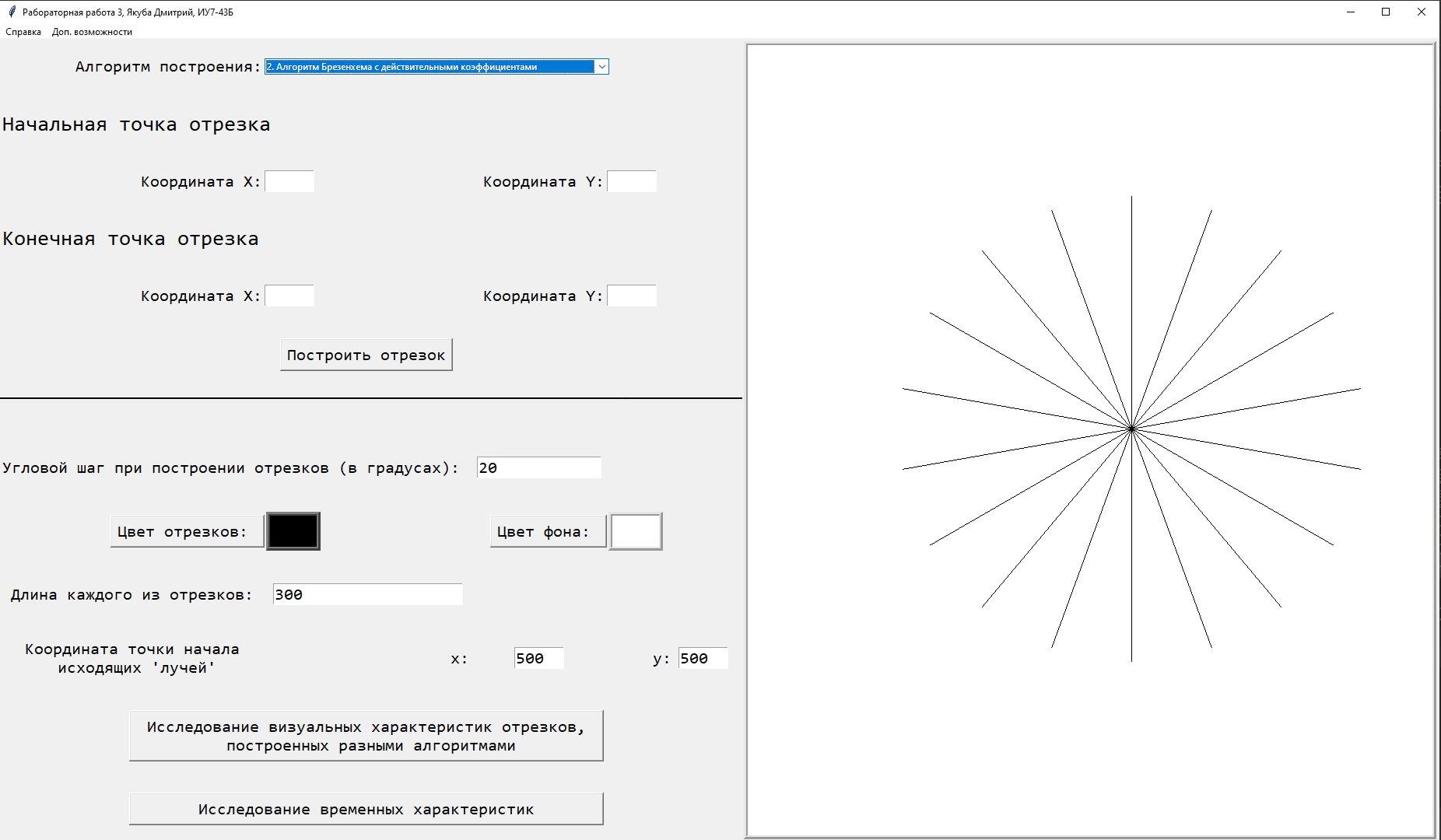


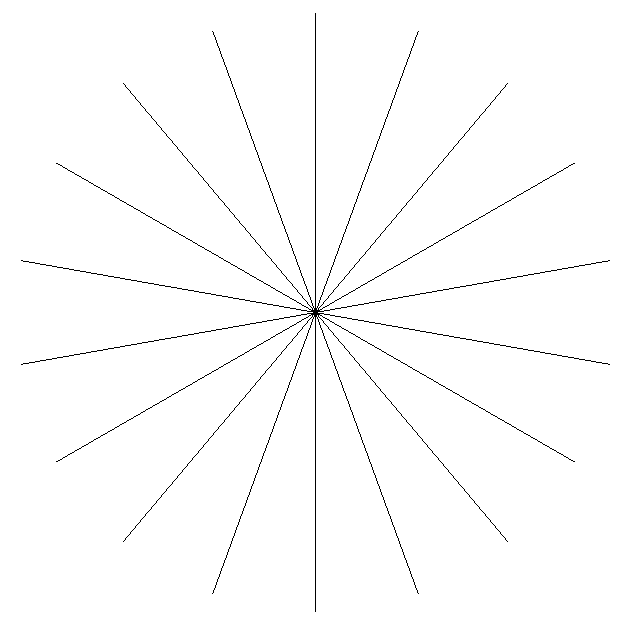


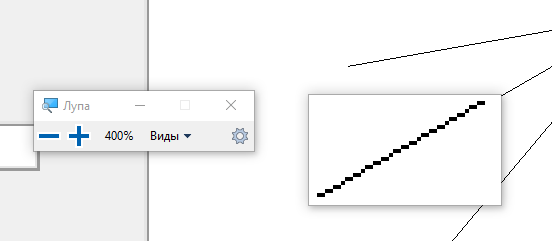


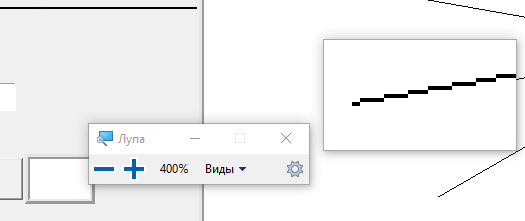


## Алгоритм Брезенхема с действительными коэффициентами

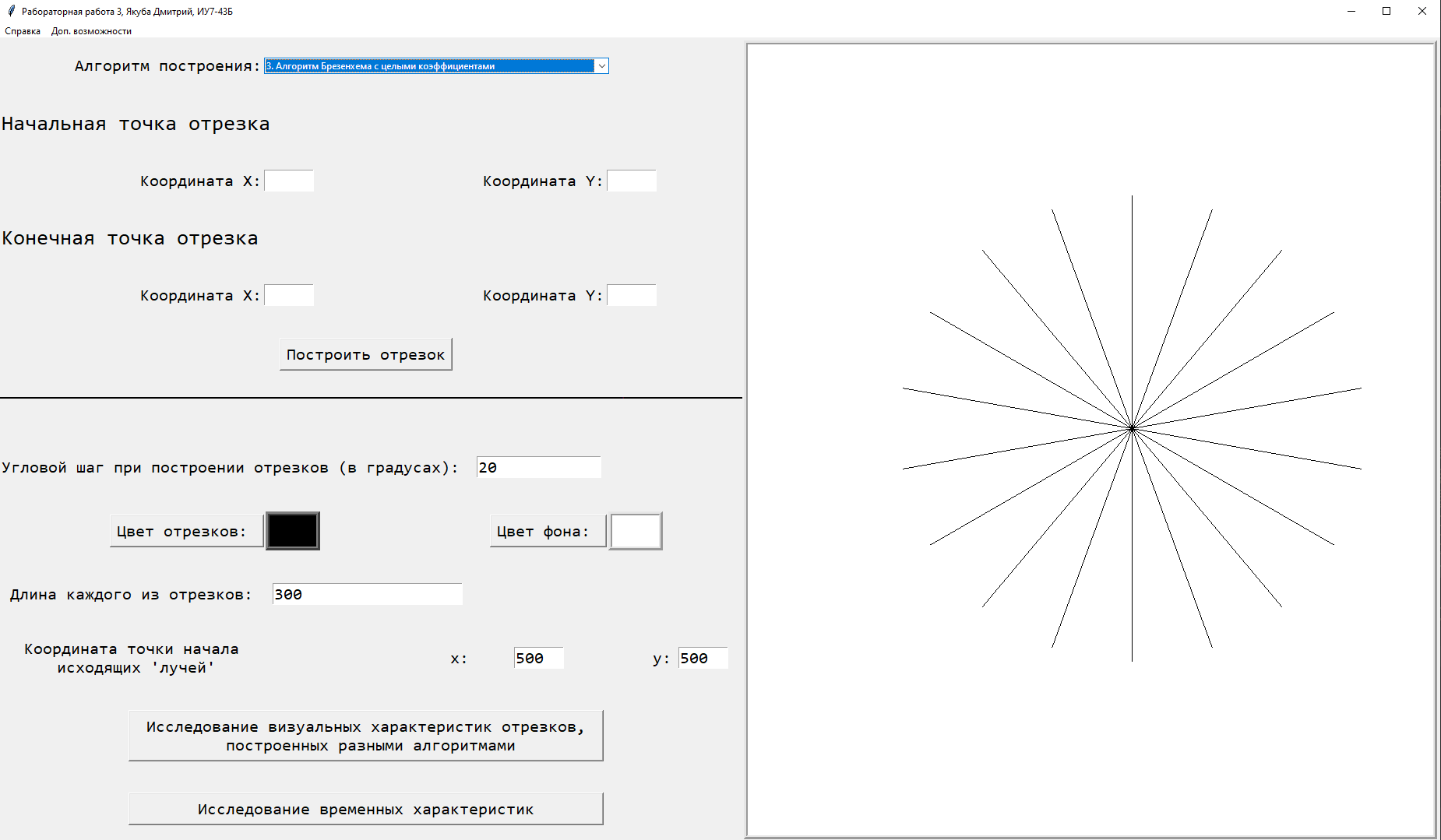


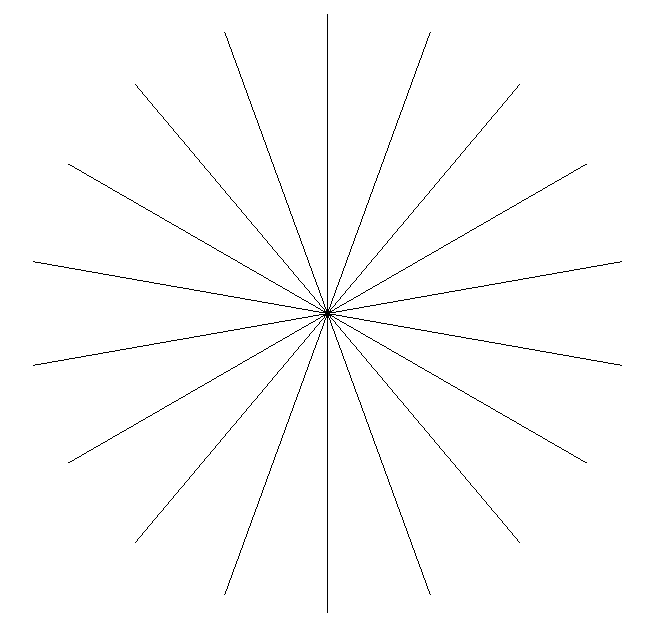




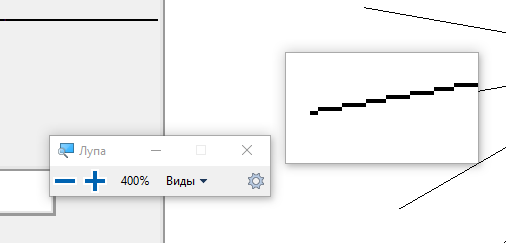


## Алгоритм Брезенхема с целыми коэффициентами

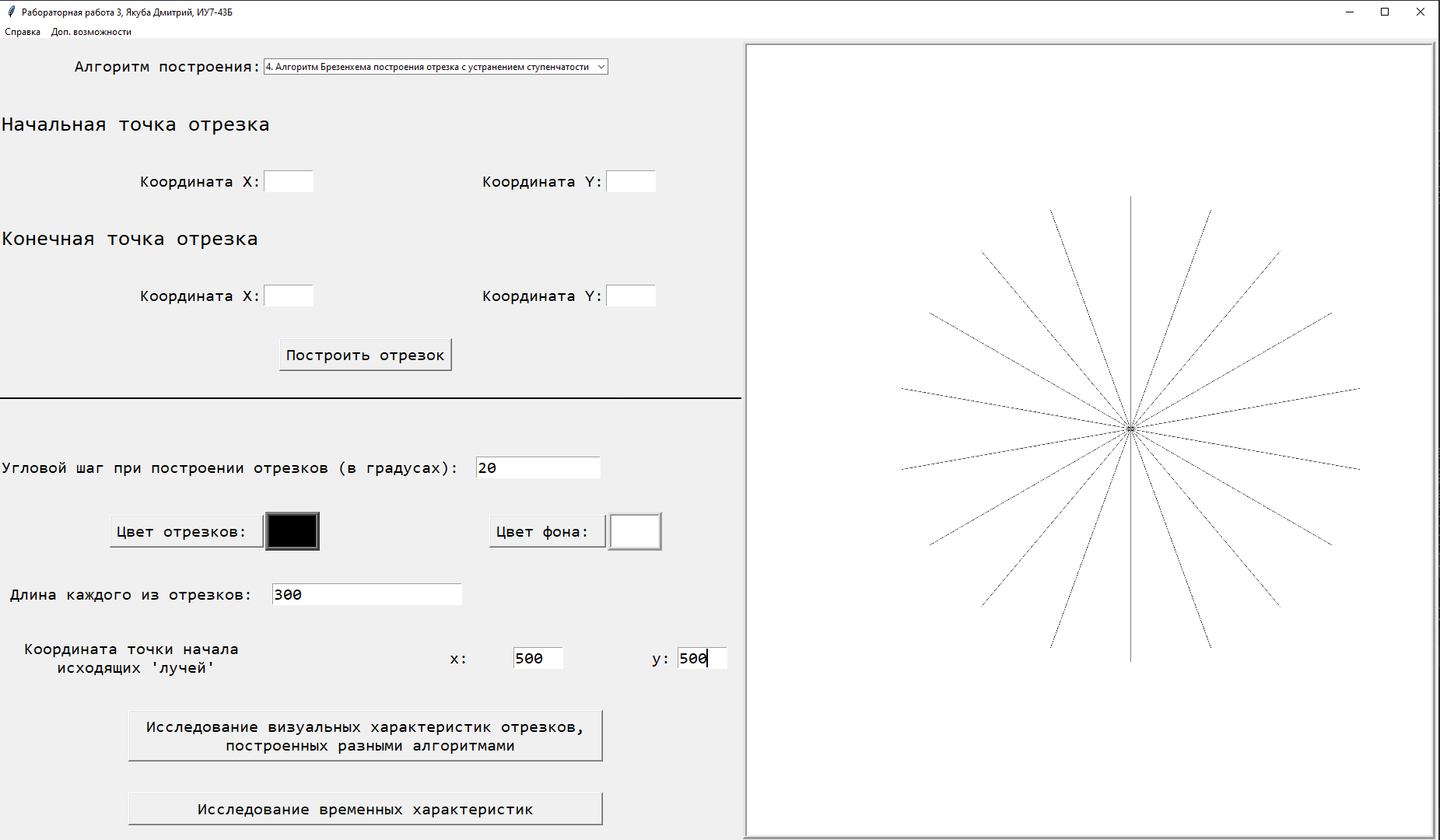


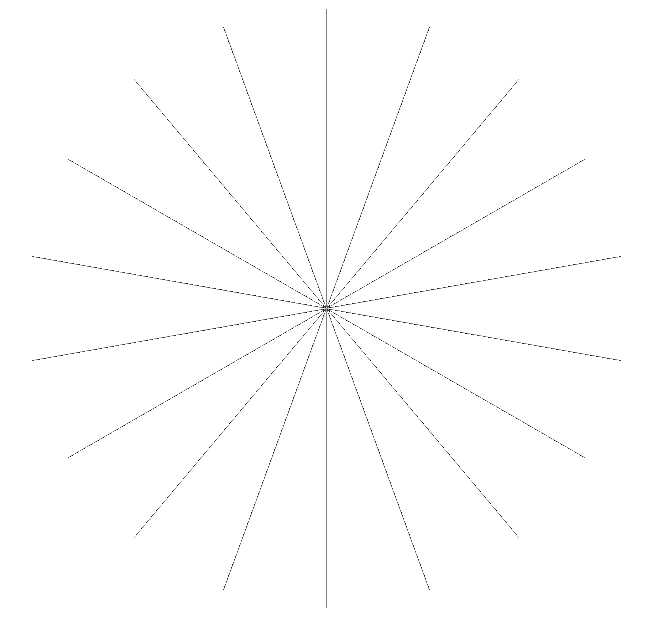


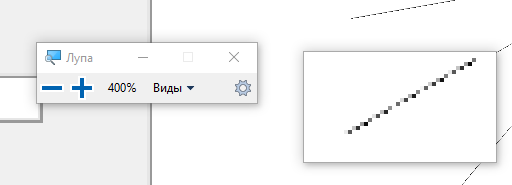


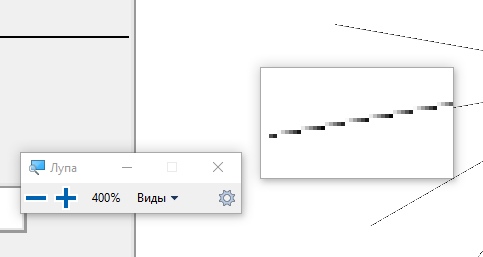


## Алгоритм Брезенхема с устранением ступенчатости

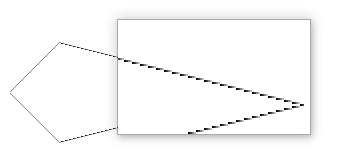




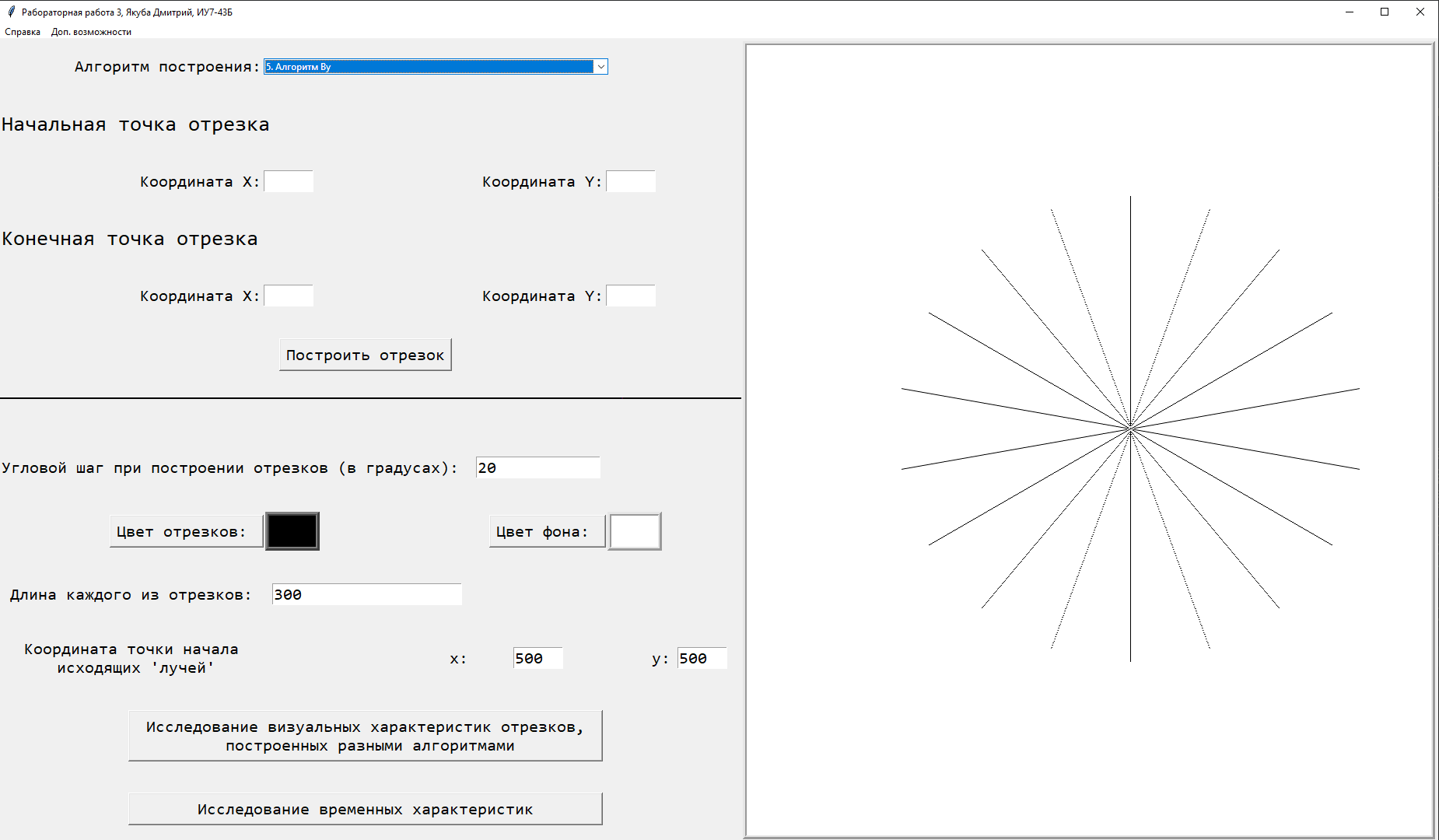


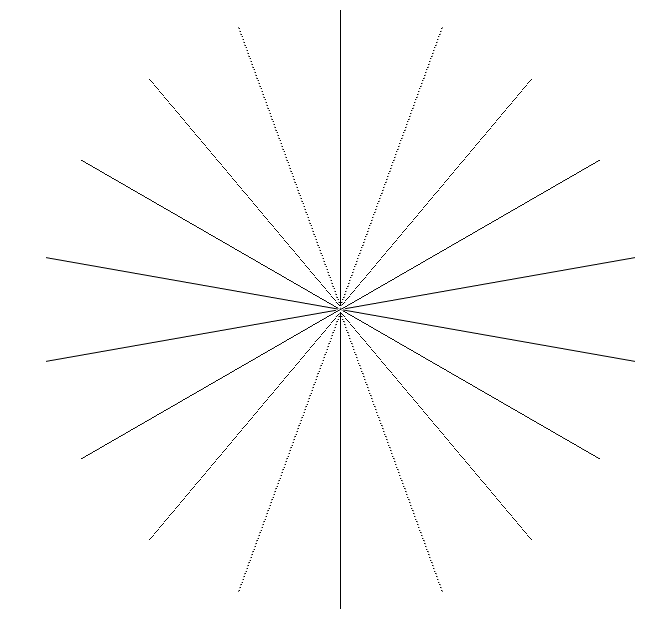


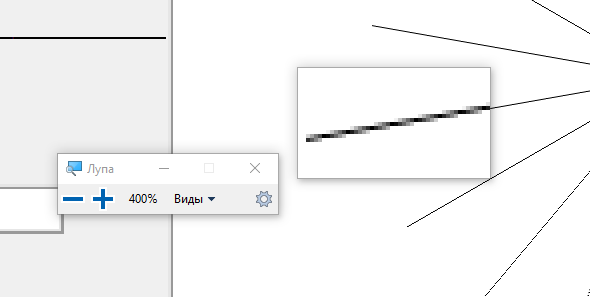
На следующей фотографии можно понять, почему считается, что данный алгоритм отлично подходит для скрытия ступенчатости геометрических фигур

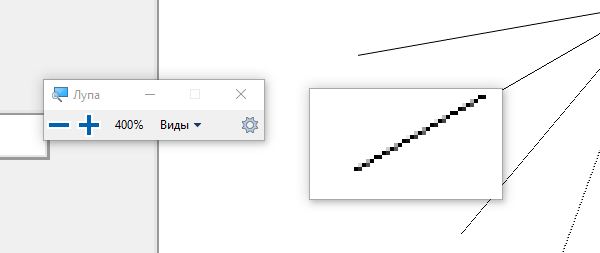


## Алгоритм Ву

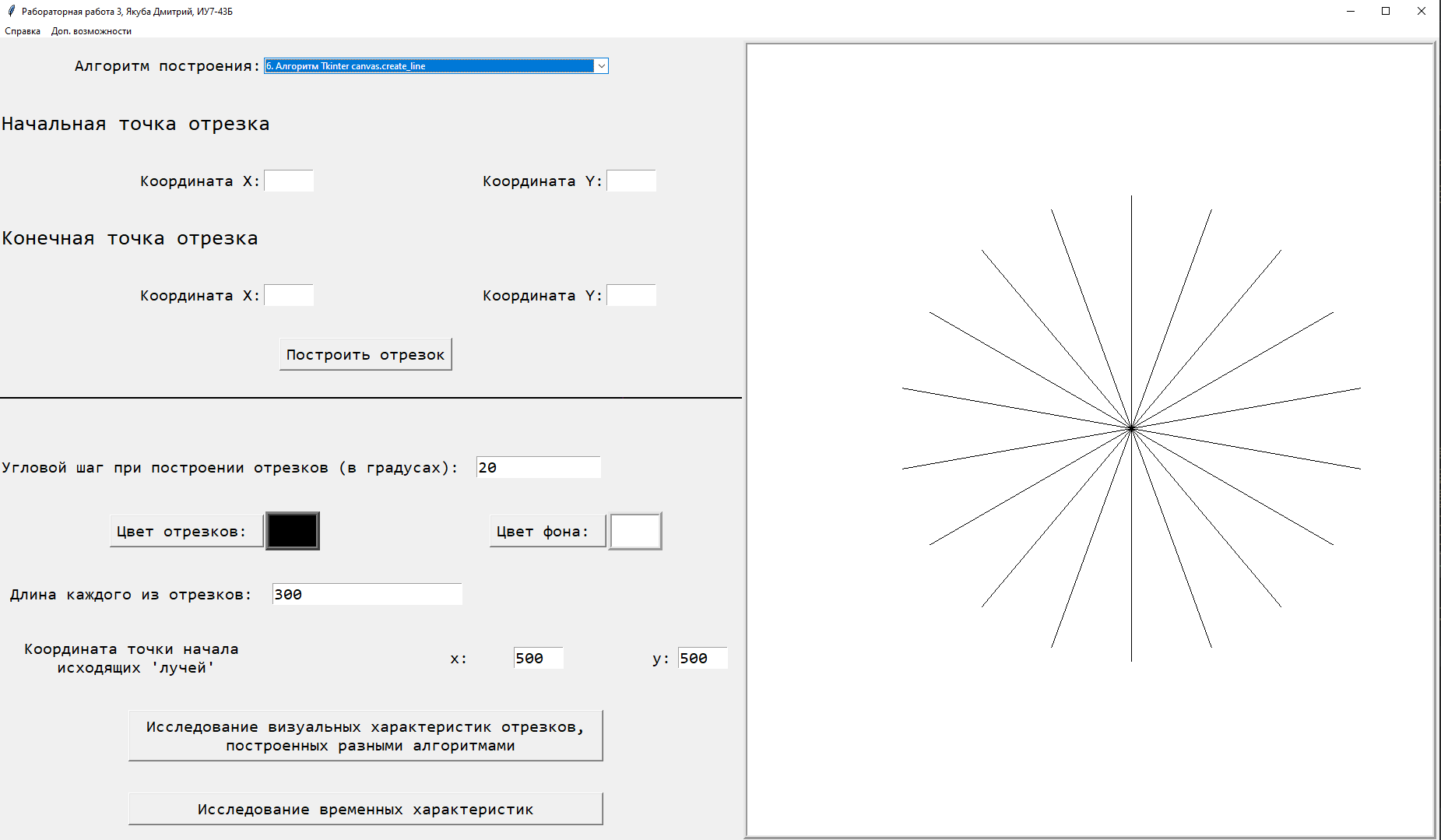


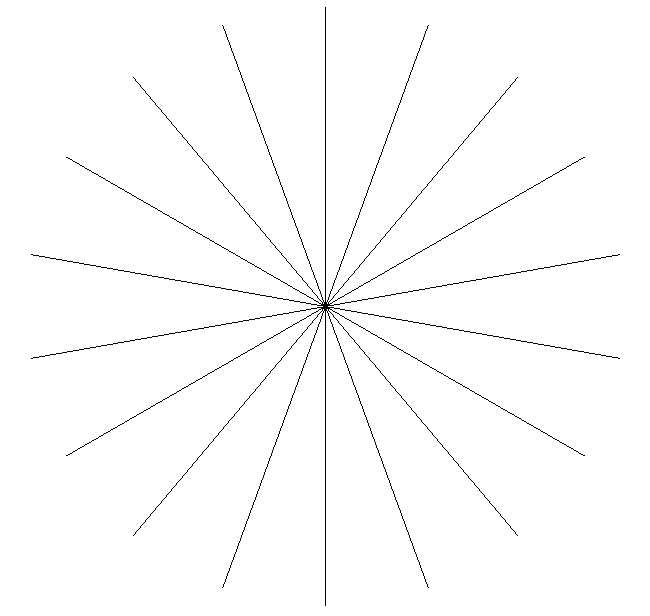


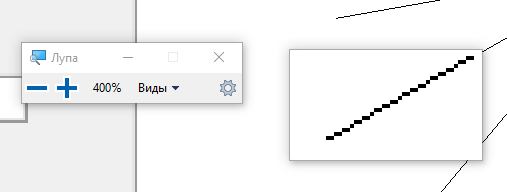


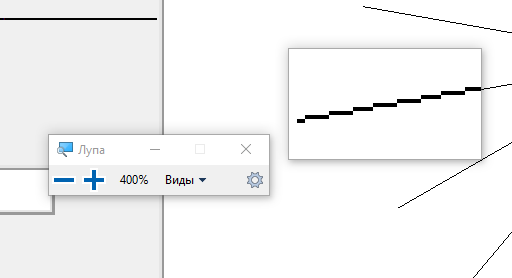


## Алгоритм Tkinter.canvas.create\_line

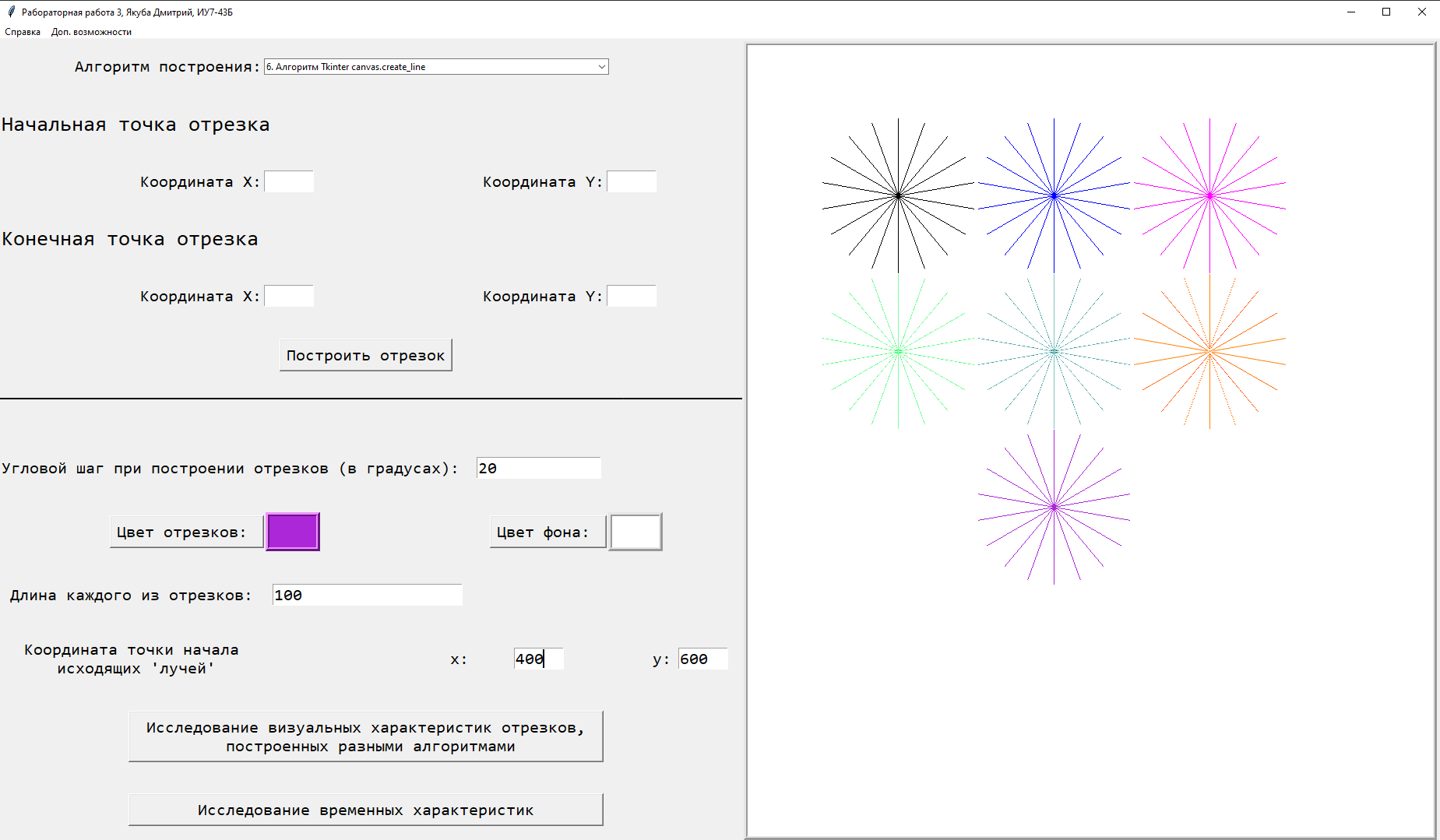


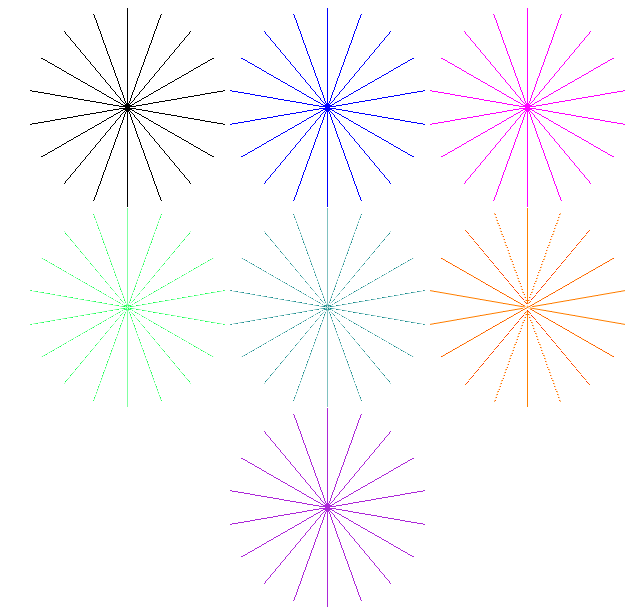






## Все алгоритмы на единой плоскости

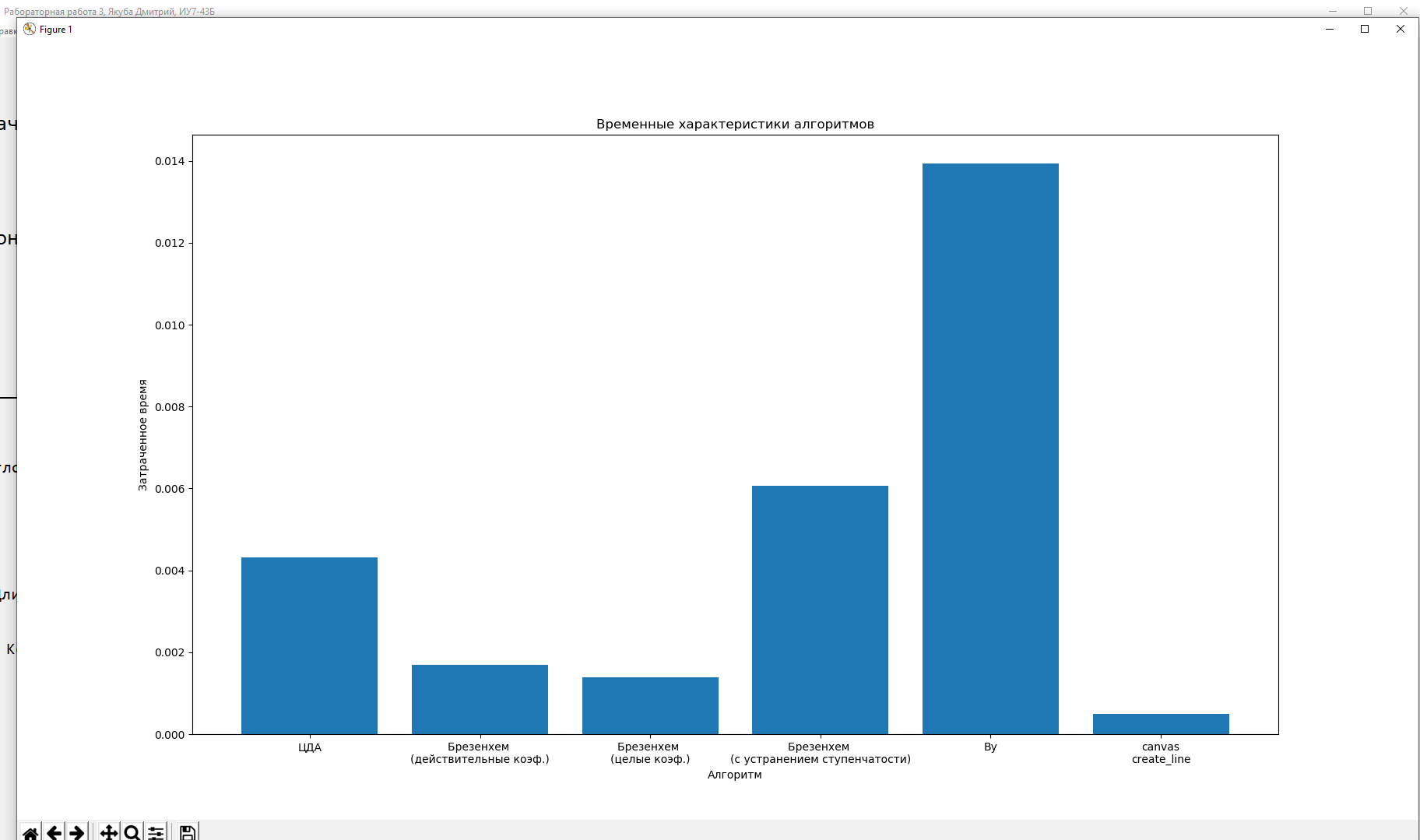




Алгоритмы идут слева-направо сверху-вниз как они упоминались в отчёте.

# Исследование временных характеристик

При исследовании временных характеристик предоставленных алгоритмов производилась тысяча замеров времени по построению одного «пучка», угловой шаг которого равен 20 градусам, а длина каждого отрезка равна 300 и вычислялось среднее время построения одного такого пучка. Результаты предоставлены в следующей диаграмме:



К моему несчастью, встроенный модуль рисования оказался слишком оптимизированным, и, таким образом, по времени никак судить о том, какой же алгоритм встроен в модуль canvas сказать нельзя. Единственное, о чём можно судить, так это о том, что это точно алгоритм без сглаживания, а, судя по скорости работы, хоть сравнивать и не с чем, я бы предположил, что это алгоритм Брезенхема с целыми коэффициентами, так как он показал наилучшую скорость среди всех прочих.