|  |  |
| --- | --- |
| 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 № 4**

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

Москва

2020 г.

Оглавление

[Цель работы 2](#_Toc38406186)

[Техническое задание 2](#_Toc38406187)

[Теоретическая часть 4](#_Toc38406188)

[Генерация окружности 4](#_Toc38406189)

[Описание и реализация алгоритмов генерации окружности 4](#_Toc38406190)

[Алгоритм на основе канонического уравнения 4](#_Toc38406191)

[Алгоритм на основе канонического уравнения 5](#_Toc38406192)

[Алгоритм Брезенхема построения окружностей 6](#_Toc38406193)

[Алгоритм средней точки построения окружностей 9](#_Toc38406194)

[Пользовательский интерфейс 11](#_Toc38406195)

[Сравнение визуальных характеристик 11](#_Toc38406196)

[Алгоритм … 11](#_Toc38406197)

[Все алгоритмы на единой плоскости 11](#_Toc38406198)

[Исследование временных характеристик 11](#_Toc38406199)

# Цель работы

**Р**еализация алгоритмов построения окружности, исследование и сравнение визуальных и временных характеристик алгоритмов.

# Техническое задание

1.Реализовать алгоритмы построения окружности на основе

* Канонического уравнения
* Параметрического уравнения ,
* Алгоритма Брезенхема
* Алгоритма средней точки
* Построение окружности с помощью библиотечной функции

Пользователь выбирает из списка определенный алгоритм, задает координаты центра, радиус, цвет рисования.

Визуальные характеристики исследуются путем рисования той же окружности цветом фона, но с помощью другого алгоритма.

2. Реализовать алгоритмы построения эллипса на основе

* Канонического уравнения
* Параметрического уравнения
* Брезенхема (модифицируется самостоятельно)
* Алгоритма средней точки
* Построение эллипса с помощью библиотечной функции

Пользователь выбирает из списка определенный алгоритм, задает координаты центра, полуоси, цвет рисования.

Визуальные характеристики исследуются путем рисования того же эллипса цветом фона, но с помощью другого алгоритма.

П. 1 и 2 предусматривают рисование одиночных кривых.

3. Сравнение визуальных характеристик разных алгоритмов при рисовании спектра концентрических окружностей.

Пользователь выбирает из списка определенный алгоритм, задает координаты центра, цвет рисования, три из следующих четырех параметров: начальный радиус, конечный радиус, шаг изменения радиуса, количество окружностей.

Визуальные характеристики исследуются путем рисования того же спектра окружностей цветом фона, но с помощью другого алгоритма.

4. Сравнение визуальных характеристик разных алгоритмов при рисовании спектра концентрических эллипсов.

Пользователь выбирает из списка определенный алгоритм, задает координаты центра, цвет рисования, начальные значения полуосей, шаг изменения одной из полуосей, количество эллипсов.

Визуальные характеристики исследуются путем рисования того же спектра эллипсов цветом фона, но с помощью другого алгоритма.

Дополнительное задание.

Сравнить временные характеристики разных алгоритмов, построив в одном поле вывода (в одной системе координат и одном масштабе) графики зависимости времени работы алгоритма от радиуса (для окружности).

Для эллипсов построить аналогичную зависимость (зависимость времени работы алгоритма от изменения полуоси. Имеется в виду, что вторая полуось тоже будет изменяться см. п.4).

# Теоретическая часть

## Генерация окружности

Чтобы построить полную окружность, достаточно сгенерировать ее одну восьмую часть. Остальные части получаются затем путем симметричного отражения относительно определенной прямой. Так, отражая одну восьмую часть, построенную в первом октанте для углов в диапазоне 0°-45°, относительно прямой с уравнением Y=X, получим одну четвертую часть, лежащую в первом квадранте. Отразив эту четверть относительно прямой X=0, получим одну вторую часть, лежащую выше оси абсцисс, наконец, отразив эту полуокружность относительно прямой Y=0, получим полную окружность.

Стоит отметить, что в каждом алгоритме предусмотрено построение только 1/8 части окружности, которые потом сначала отражаются относительно прямой x = y, проходящей через центр окружности, после – относительно прямой y = 0, проходящей через центр окружности, а потом – относительно прямой x = 0, проходящей через центр окружности (отражённые точки добавляются в переданный массив точек).

## Описание и реализация алгоритмов генерации окружности

### Алгоритм на основе канонического уравнения

Из курса аналитической геометрии нам известно, что окружность можно описать следующим каноническим уравнением:

, что справедливо для окружности с центром в точке начала координат.

Для окружности, центр которой не совпадает с точкой начала координат:

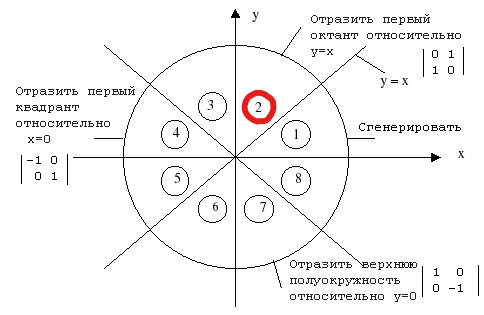
Где *x0, y0 –* координаты центра окружности.

С помощью уравнения (2) имеем возможность через *y* или *x* выразить точки окружности.

Выразим через уравнение (2) *y*:

Так как нам достаточно нарисовать только 1/8 часть окружности, а выше говорилось о том, что строим окружность мы во 2 октанте (красная отметка на рисунке ниже), то при построении мы воспользуемся следующей формулой:

, так как первый октант находится над прямой, проходящей через центр окружности и параллельной оси X.



Выбор 2-го октанта для построения 1/8 окружности обусловлен тем фактом, что в нём приращение значения *y* меньше приращения значения по *x*, что позволит нам построить непрерывную кривую.

#### Алгоритм, записанный на ЯП Python:

def canonicalCircleAlg(xCenter, yCenter, radius, colour = "#000000"):  
 pointsArray = []  
 sqrRad = radius \* radius  
 for curX in range(xCenter, round(xCenter + radius / sqrt(2)) + 1):  
 curY = yCenter + sqrt(sqrRad - (curX - xCenter) \* (curX - xCenter))  
 pointsArray.append((curX, curY, colour))

reflectPointsXY(pointsArray, xCenter, yCenter)  
 reflectPointsY(pointsArray, xCenter)  
 reflectPointsX(pointsArray, yCenter)  
 return pointsArray

### Алгоритм на основе канонического уравнения

Из курса аналитической геометрии нам также известно следующее параметрическое уравнение окружности:

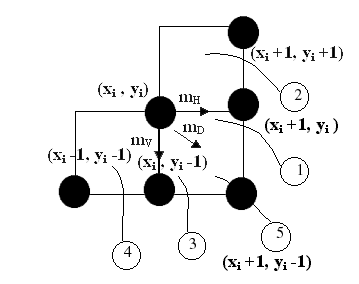
Чтобы определить точки окружности, нужно выбрать некоторый шаг, равный для параметра *t*, и рассчитать для каждого значения этого параметра значения координат соответствующих точек окружности. Значение величины шага приравнивается указанной величине, так как в таком случае угловой шаг будет уменьшаться при увеличении радиуса, что позволит отобразить кривую непрерывной.

#### Алгоритм, записанный на ЯП Python:

def parameterCircleAlg(xCenter, yCenter, radius, colour = "#000000"):  
 pointsArray = []  
 angleStep = 1 / radius  
 i = 0  
 while i <= pi / 4:  
 curX = xCenter + radius \* cos(i)  
 curY = yCenter + radius \* sin(i)  
 pointsArray.append((curX, curY, colour))  
 i += degreeStep  
  
 reflectPointsXY(pointsArray, xCenter, yCenter)  
 reflectPointsY(pointsArray, xCenter)  
 reflectPointsX(pointsArray, yCenter)  
 return pointsArray

### Алгоритм Брезенхема построения окружностей

В данный алгоритм строится на том, что для любой заданной точки на окружности при генерации по часовой стрелке существует только три возможности выбрать следующий пиксел, наилучшим образом приближающий окружность: горизонтально вправо, по диагонали вниз и вправо, вертикально вниз.

Таким образом, перед нами стоит выбор между тремя пикселями, показанных квадратами на рисунке.

Разность между квадратами расстояний от центра окружности до диагонального пиксела (*xi, + 1, уi - 1*) и от центра до точки на окружности *R2* равна следующей величине:

При этом, при реализации алгоритма, следует анализировать только знак ошибки. В таком случае, мы будем иметь три следующих случая:

Случай 1.

Расстояние до центра окружности больше, чем до диагонального пикселя, диагональная точка лежит внутри реальной окружности, выбор стоит между горизонтальным и диагональным пикселем.

Введём следующую величину:

Таким образом теперь мы можем проанализировать разницу расстояний от горизонтального и диагонального пиксела до окружности. Если полученная при вычислении величина положительна, то расстояние от диагонального пикселя до окружности меньше, чем от горизонтального. По аналогии для случая, когда полученная величина меньше нуля, расстояние до горизонтального пикселя меньше, следует взять именно его.

Чтобы сократить количество вычислений (а мы видим тут и возведение в квадрат, и модуль, что слишком для нас затратно по времени), рассматриваются два следующих случая:

1)

В таком случае, раскрыв модули, получим:

Дополнение до полного квадрата члена с помощью добавления и вычитания дает:

Таким образом получаем:

И это существенно упрощает вычисления.

2)

Раскрыв модули, получаем:

Заметим, что случай 2) в программной реализации алгоритма не рассматривается отдельно, так как при том же расположении пикселей, что и в рассмотренном 2), случай 1) тоже будет давать отрицательную величину.

Так как для рассмотрения и проверки случая (2) нам понадобятся формулы случая (3), для начала приведём случай (3).

Случай 3.

Расстояние до диагонального пикселя больше, чем до центра окружности, диагональная точка лежит вне окружности, выбираем либо диагональный, либо вертикальный.

Введём величину:

Вновь рассмотрим два случая и раскроем модули.

1)

Имеем:

И таким образом:

2)

Заметим, что случай 2) в программной реализации алгоритма не рассматривается отдельно, так как при том же расположении пикселей, что и в рассмотренном 2), случай 1) тоже будет давать положительную величину.

Случай 2.

Окружность проходит через диагональный пиксел.

Выбор – однозначно диагональный пиксел.

Проверим предыдущие условия:

Рекуррентные соотношения для пошагового алгоритма находятся с помощью подстановки *xi+1*и *yi+1*, которые мы выразим через *xi*и *yi*:

При вертикальном шаге , а

Таким образом

При горизонтальном шаге , а

Таким образом

При диагональном шаге а

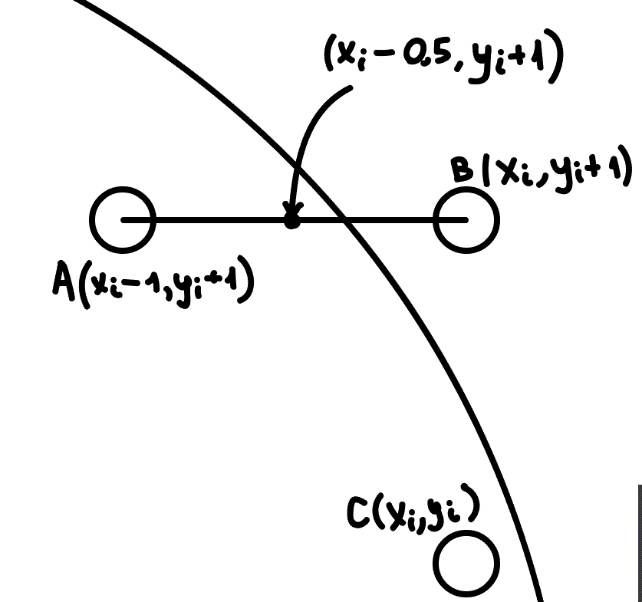
Таким образом

#### Алгоритм, записанный на ЯП Python:

def bresenhamCircleAlg(xCenter, yCenter, radius, colour = "#000000"):  
 pointsArray = []  
  
 curX = 0  
 curY = radius  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 delta = 2 - radius - radius  
 while curX < curY:  
 if delta <= 0:  
 d = delta + delta + curY + curY - 1  
 curX += 1  
 if d >= 0 :  
 curY -= 1  
 delta += 2 \* (curX - curY + 1)  
 else:  
 delta += curX + curX + 1  
 else:  
 d = delta - curX + delta - curX - 1  
 curY -= 1  
 if d < 0:  
 curX += 1  
 delta += curX + curX - curY - curY + 2  
 else:  
 delta -= curY + curY - 1  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 reflectPointsXY(pointsArray, xCenter, yCenter)  
 reflectPointsY(pointsArray, xCenter)  
 reflectPointsX(pointsArray, yCenter)  
 return pointsArray

### Алгоритм средней точки построения окружностей

Суть данного алгоритма также заключается в анализе проходящей «реальной» кривой.

Константный шаг по *y* = 1, в то время как по положению средней точки относительно идеальной кривой мы будем определять, понадобится ли нам шаг *x* или не понадобится.

При построении 1/8 части окружности важно, как и в алгоритме на основе канонического уравнения (только теперь в обратном случае, так как идёт работа с *х*), чтобы приращение по значению *x* было больше, чем приращение по значению *y*, что сразу относит нас к тому, что рассматриваться будет 1-я октанта окружности.

Также стоит отметить, что выбирать пикселы мы начнём с точки

Введём следующую функцию:

По знаку данной функции в дальнейшем мы сможем определить то, какой пиксел при отображении нам выбрать, так как она представляет собой разной расстояния от центра окружности до средней точки и расстояния от центра окружности до идеальной кривой. Таким образом, если функция отрицательна, то это будет значить, что средняя точка лежит в окружности, а в обратном случае – вне.

Таким образом, если , то мы выбираем диагональный пиксел, если же , то мы выбираем вертикальный пиксел (если обращаться к картинке в начале пункта).

Чтобы увеличить скорость работы алгоритма вводится следующая величина:

Откуда:

И таким образом…

Для диагонального хода:

Для вертикального хода:

#### Алгоритм, записанный на ЯП Python:

def middlePointCircleAlg(xCenter, yCenter, radius, colour = "#000000"):  
 pointsArray = []  
  
 curX = radius  
 curY = 0  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 func = 1 - radius  
  
 while curY < curX:  
 curY += 1  
 if func > 0:  
 curX -= 1  
 func -= curX - 2 + curX  
  
 func += curY + curY + 3  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
 reflectPointsXY(pointsArray, xCenter, yCenter)  
 reflectPointsY(pointsArray, xCenter)  
 reflectPointsX(pointsArray, yCenter)  
 return pointsArray

## Генерация эллипса

Алгоритмы построения эллипсов достаточно схожи с алгоритмами построения окружностей, но в это же время они несколько сложнее.

При переходе к рассмотрению эллипсов очень важно отметить тот факт, что теперь строиться будет не 1/8 часть фигуры, а 1/4 часть фигуры, так как данная кривая обладает из удобных для нас только осевой симметрией относительно прямой, параллельной оси Х, проходящей через центр эллипса, и прямой, параллельной оси Y, проходящей через центр эллипса. А в связи с тем, что теперь нам приходится строить 1/4 также в реализации некоторых алгоритмов понадобится вычислять точку перехода при построении из октанты 1 в октанту 2, так как приращения в данной точке по x и по y меняются, а следовательно, чтобы качественно и непрерывно отрисовать фигуру, нам потребуется «менять» между собой рассматриваемые величины в точке перехода.

### Алгоритм, основанный на каноническом уравнении эллипса

Из курса аналитической геометрии нам известно каноническое уравнение эллипса:

Оно описывает эллипс с центром в начале координат, оси которого совпадают с осями координат.

Чтобы перейти к эллипсу с центром в точке, не совпадающей с началом координат, воспользуемся следующим уравнением:

Выразив *x* и *y* можем записать алгоритм.

#### Алгоритм, записанный на ЯП Python:

def canonicalEllipseAlg(xCenter, yCenter, radiusX, radiusY, colour = "#000000"):  
 pointsArray = []  
  
 sqrRadX = radiusX \* radiusX  
 sqrRadY = radiusY \* radiusY  
 sqrMix = sqrRadX \* sqrRadY  
  
 limitX = niceRound(xCenter + radiusX / sqrt(1 + sqrRadY / sqrRadX))  
 limitY = niceRound(yCenter + radiusY / sqrt(1 + sqrRadX / sqrRadY))  
  
 for curX in range(xCenter, limitX):  
 curY = yCenter + sqrt(sqrMix - (curX - xCenter) \* (curX - xCenter) \* sqrRadY) / radiusX  
 pointsArray.append((curX, curY, colour))  
  
 for curY in range(limitY, yCenter - 1, -1):  
 curX = xCenter + sqrt(sqrMix - (curY - yCenter) \* (curY - yCenter) \* sqrRadX) / radiusY  
 pointsArray.append((curX, curY, colour))  
  
 reflectPointsX(pointsArray, xCenter)  
 reflectPointsY(pointsArray, yCenter)  
 return pointsArray

### Алгоритм, основанный на параметрическом уравнении эллипса

Из курса аналитической геометрии нам известно параметрическое уравнение эллипса:

При построении эллипса с помощью данного алгоритма возникает некоторое количество «проблем».

Первой проблей является выбор шага. Мы можем взять для реализации как шаг для большой полуоси, так и для малой полуоси эллипса, но и там, и там нас подстерегает неприятность. В случае выбора большой полуоси – мы начинаем строить «лишние» точки. В случае выбора маленькой полуоси – кривая теряет свою непрерывность. В целях улучшения ситуации, конечно, можно высчитывать шаг, анализируя текущий угол, но это очень неэффективно. Также в разряд «проблем» можно добавить функции вычисления косинуса и синуса угла.  
Вполне возможно, что, раз данный алгоритм имеет место, то существую ещё какие бы то ни было способы улучшить ситуацию, но, как я понимаю, это очень сложные способы, а к его восприятию мы ещё можем быть не вполне готовы.

#### Алгоритм, записанный на ЯП Python:

def parameterEllipseAlg(xCenter, yCenter, radiusX, radiusY, colour = "#000000"):  
 pointsArray = []  
  
 if radiusX > radiusY:  
 step = 1 / radiusX  
 else:  
 step = 1 / radiusY  
  
 i = 0  
 while i <= pi / 2 + step:  
 curX = xCenter + radiusX \* cos(i)  
 curY = yCenter + radiusY \* sin(i)  
 pointsArray.append((curX, curY, colour))  
  
 i += step  
  
 reflectPointsY(pointsArray, xCenter)  
 reflectPointsX(pointsArray, yCenter)  
 return pointsArray

### Алгоритм Брезенхема построения эллипсов

Данный алгоритм реализуется точно так же, как и алгоритм Брезенхема для окружностей, с той только поправкой, что теперь мы генерируем 1/4 фигуры и используется каноническое уравнение эллипса.

#### Алгоритм, записанный на ЯП Python:

def bresenhamEllipseAlg(xCenter, yCenter, radiusX, radiusY, colour = "#000000"):  
 pointsArray = []  
  
 curX = 0  
 curY = radiusY  
  
 sqrRadX = radiusX \* radiusX  
 sqrRadY = radiusY \* radiusY  
  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 delta = sqrRadY - sqrRadX \* (radiusY + radiusY + 1)  
 while curY > 0:  
 if delta <= 0:  
 negDek = delta + delta + sqrRadX \* (curY + curY - 1)  
 curX += 1  
 delta += sqrRadY \* (curX + curX + 1)  
 if negDek >= 0:  
 curY -= 1  
 delta += sqrRadX \* (-curY - curY + 1)  
 else:  
 posDek = delta + delta + sqrRadY \* (-curX - curX - 1)  
 curY -= 1  
 delta += sqrRadX \* (-curY - curY + 1)  
 if posDek < 0:  
 curX += 1  
 delta += sqrRadY \* (curX + curX + 1)  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 reflectPointsY(pointsArray, xCenter)  
 reflectPointsX(pointsArray, yCenter)  
  
 return pointsArray

### Алгоритм средней точки построения эллипсов

Данный алгоритм реализуется точно так же, как и алгоритм средней точки для окружностей, с той только поправкой, что теперь мы генерируем 1/4 фигуры (что понесёт за собой немаловажное разбиение на два цикла, чтобы кривая оставалась непрерывной) и используется каноническое уравнение эллипса.

#### Алгоритм, записанный на ЯП Python:

def middlePointEllipseAlg(xCenter, yCenter, radiusX, radiusY, colour = "#000000"):  
 pointsArray = []  
  
 sqrRadX = radiusX \* radiusX  
 sqrRadY = radiusY \* radiusY  
  
 limit = round(radiusX / sqrt(1 + sqrRadY / sqrRadX))  
  
 curX = 0  
 curY = radiusY  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 func = sqrRadY - round(sqrRadX \* (radiusY - 1 / 4))  
 while curX < limit:  
 if func > 0:  
 curY -= 1  
 func -= sqrRadX \* curY \* 2  
  
 curX += 1  
 func += sqrRadY \* (curX + curX + 1)  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 limit = round(radiusY / sqrt(1 + sqrRadX / sqrRadY))  
  
 curX = radiusX  
 curY = 0  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 func = sqrRadX - round(sqrRadY \* (curX - 1 / 4))  
 while curY < limit:  
 if func > 0:  
 curX -= 1  
 func -= 2 \* sqrRadY \* curX  
  
 curY += 1  
 func += sqrRadX \* (curY + curY + 1)  
 pointsArray.append((curX + xCenter, curY + yCenter, colour))  
  
 reflectPointsX(pointsArray, yCenter)  
 reflectPointsY(pointsArray, xCenter)  
  
 return pointsArray

# Пользовательский интерфейс

Если сильно захочется, то сделаешь.

# Сравнение визуальных характеристик

## Алгоритм …

## Все алгоритмы на единой плоскости

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

# Исследование временных характеристик

В завершении посмотри вывод Паши по каждому алгоритму, который с кодом лежит. А то потеряешь, а там умные вещи говорят…