В математике можно встретиться с изменениями объектов в пространстве: прямолинейным движением, вращением, деформацией. Например, нужно описать на плоскости уравнением, относительно какой-то системы отсчёта и известно уравнение относительно другой системы отсчёта. Подобные задачи можно решить, используя методы линейной алгебры. Однако, сложно использовать матрицы, не имея визуального представления об их природе.

Проблемы: визуализация геометрического смысла операций с матрицами; построение графиков функций и их линейное преобразование.

Цель: Автоматизировать сложные вычислительные процессы при построении, преобразовании графиков функции с помощью программы.

План реализации:

1. Изучение и подключение необходимых библиотек.
2. Реализация классов основных объектов на плоскости.
3. Создание плоскости для изображения графиков и векторов.
4. Создание окна для параметров.
5. Добавление векторов и матрицы преобразований .
6. Математическое решение проблемы поворота графиков функций.
7. Автоматизация решения, создание интерфейса для ввода функций.
8. Тестирование.

1.0

Библиотеки

Для написания программы в первую очередь требуется знание библиотек: для написания оконного приложения – PyQt, для моделирования плоскости, на которой можно изображать векторы уравнения графика – PyGame, для упрощения вычислений и применения операций с матрицами и векторами – NumPy.

Все три библиотеки, а также программирование на Python в общем я изучил на курсах дополнительного образования Яндекс.Лицей.

В файле *config.py* импортиреум нужные функции из библиотек и зададим константы:

import pygame  
import numpy as np  
from numpy import sqrt, log, sin, sinh, cos, cosh, tan, tanh, arccos, arcsin, arctan, e, pi  
from PyQt5 import QtGui  
from PyQt5.QtWidgets import QMainWindow, QApplication, QShortcut  
from PyQt5 import uic  
from PyQt5.QtCore import Qt  
import sys  
  
# screen size  
size = width, height = 1200, 800  
fps = 30  
# colors in rgb  
WHITE = (255, 255, 255)  
BLACK = (0, 0, 0)  
DARK\_BLUE = (0, 0, 255)  
BLUE = (100, 149, 237)  
GREY = (128, 128, 128)  
RED = (255, 0, 0)  
Epsilon = 10 \*\* (-10)  
COLORS = {'green': (128, 191, 132),  
 'blue': (57, 140, 191),  
 'orange': (217, 61, 4),  
 'yellow': (242, 183, 5),  
 'purple': (145, 38, 191)}

2.0

Реализация классов основных объектов на плоскости

В качестве графического движка я выбрал PyGame, для упрощения работы. Основные объекты на плоскости в данной реализации – точка (вершина) и вектор.

2.1

Точка

Создадим класс:

class Vertex:  
 def \_\_init\_\_(self, coords, unit, center):  
 # coordinates  
 self.coords = coords  
 # coords after applying transformation  
 self.transformed = coords  
 # information to get real coords if vertex  
 self.unit = unit  
 self.center = center

В инициализатор класс принимает координаты, количество пикселей в единичном отрезке и координаты центра плоскости, в *transformed* сохраняются координаты после перемещения.

def real\_cords(self):  
 return [int(self.transformed[0] \* self.unit + self.center[0]),  
 int(-self.transformed[1] \* self.unit + self.center[1])]

Данный метод возвращает “настоящие” координаты (т.е. относительно плоскости PyGame): координата точки \* количество пикселей + координата центра. Минус во второй координате стоит с учётом, что ось Oy в плоскости окна PyGame направлена вниз (Начало координат в PyGame в верхнем левом углу)

def transform(self, matrix):  
 cord = np.array(self.coords)  
 new = matrix.dot(cord)  
 self.transformed = [new[0], new[1]]

Метод *transform* представляет точку как радиус-вектор с концом в этой точке с изначальными координатами, а затем умножает его на данную матрицу. Координаты полученного вектора записываются в изменённые координаты точки.

def connect(self, window, another\_vertex, color, weight=1):  
 pygame.draw.line(window, color, self.real\_cords(), another\_vertex.real\_cords(), weight)

Метод *connect* соединяет данную точку с другой отрезком заданной толщины и указанным цветом.

def draw(self, window):  
 pygame.draw.circle(window, BLACK, self.real\_cords(), 2)

Метод *draw* рисует точку на экране.

2.2

Класс вектора унаследуем от точки:

class Vector(Vertex):  
 def \_\_init\_\_(self, coords, unit, central\_vertex, color):  
 super().\_\_init\_\_(coords, unit, central\_vertex.real\_cords())  
 self.color = color  
 # start of vector  
 self.central\_vertex = central\_vertex

Помимо параметров точки здесь сохраняется объект центральной точки.

def draw(self, window):  
 self.connect(window, self.central\_vertex, self.color, 2)  
 pygame.draw.circle(window, self.color, self.real\_cords(), 3)

Метод *draw* рисует радиус-вектор, соединяя конец вектора и центральную вершину. А также выделяется конец вектора

def change\_cords(self, real):  
 self.transformed = [(real[0] - self.center[0]) / self.unit, - (real[1] - self.center[1]) / self.unit]

Метод *change\_cords* по заданным ‘реальным’ координатам меняет относительные координаты.

А также вспомогательный класс для обозначения базисных векторов:

class BasisVector(Vector):  
 def \_\_init\_\_(self, coords, unit, center, color):  
 super().\_\_init\_\_(coords, unit, center, color)

3.0

Создание плоскости для изображения графиков и векторов

Создадим класс BackGrid для сохранения изначального состояния плоскости:

class BackGrid:  
 def \_\_init\_\_(self, center, side, cell\_size, color):  
 self.x = center[0]  
 self.y = center[1]  
 # size of grid  
 self.side = (side // 2) \* 2 + 1  
 self.cell\_size = cell\_size  
 self.color = color  
 # vertexes that determine plane  
 self.top\_vertexes = [Vertex((i, side // 2), cell\_size, center) for i in range(- side // 2, side // 2 + 1)]  
 self.bot\_vertexes = [Vertex((i, - side // 2), cell\_size, center) for i in range(- side // 2, side // 2 + 1)]  
 self.left\_vertexes = [Vertex((- side // 2, i), cell\_size, center) for i in range(- side // 2, side // 2 + 1)]  
 self.right\_vertexes = [Vertex((side // 2, i), cell\_size, center) for i in range(- side // 2, side // 2 + 1)]