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 и сборка проекта.

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)]

Для прорисовки сетки создадим “рамку” из точек – списки: *top\_vertexes, bot\_vertexes, left\_vertexes, right\_vertexes.*

Метод, соединяющий верхние и нижние, левые и правые точки соединим с помощью метода *draw\_object:*

def draw\_object(self, window, weight=1):  
 for i in range(self.side):  
 if i == self.side // 2 and self.\_\_class\_\_ == Grid:  
 self.top\_vertexes[i].connect(window, self.bot\_vertexes[i], BLACK, weight)  
 else:  
 self.top\_vertexes[i].connect(window, self.bot\_vertexes[i], self.color, weight)  
 for i in range(self.side):  
 if i == self.side // 2 and self.\_\_class\_\_ == Grid:  
 self.left\_vertexes[i].connect(window, self.right\_vertexes[i], BLACK, weight)  
 else:  
 self.left\_vertexes[i].connect(window, self.right\_vertexes[i], self.color, weight)

Создадим класс плоскости, где к каждой точке будет применяться линейное преобразование:

class Grid(BackGrid):  
 def \_\_init\_\_(self, center, side, cell\_size, color):  
 super().\_\_init\_\_(center, side, cell\_size, color)  
 self.central\_vertex = Vertex((0, 0), cell\_size, center)  
 # Basis vectors determine all vertexes in plane.  
 # Changing basis vectors coordinates we apply  
 # linear transformation to all points in plane:  
 # [ i.x j.x ]  
 # M =[ i.y j.y ] , where i, j - basis vectors and x, y - their coordinates  
 self.basis\_vectors = [BasisVector((1, 0), self.cell\_size, self.central\_vertex, DARK\_BLUE),  
 BasisVector((0, 1), self.cell\_size, self.central\_vertex, RED)]  
 # common vectors, we can drag them by mouse  
 self.vectors = []  
 self.available\_colors = [(145, 38, 191), (242, 183, 5), (217, 61, 4), (57, 140, 191), (128, 191, 132)]  
 # buffer for previous vector, for changing it's coords via tab  
 self.prev\_vector = None  
 # function  
 self.function = ''  
 self.transformed\_function = ''  
 self.function\_dots = []

Здесь дополнительно хранятся базисные векторы, векторы, заданные пользователем, и их возможные цвета (всего не более 5 векторов) точки функции, по которым она будет строиться.

Метод для прорисовки плоскости:

def draw\_object(self, window, weight=1):  
 super().draw\_object(window, weight)  
 self.basis\_vectors[0].draw(window)  
 self.basis\_vectors[1].draw(window)  
 for dot in self.function\_dots:  
 try:  
 dot.draw(window)  
 except Exception as err:  
 pass  
 for vector in self.vectors:  
 try:  
 vector.draw(window)  
 except Exception as err:  
 pass  
 pygame.draw.circle(window, BLACK, (self.x, self.y), 3)

4.0

Создание окна для параметров

Для реализации графического интерфейса я использовал библиотеку PyQt.

4.1

Создадим класс окна:

class MainWindow(QMainWindow):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 # load pattern  
 self.ui = uic.loadUi('main\_window.ui', self)  
  
 # apply linear transformation  
 self.ApplyButton.clicked.connect(self.apply)  
  
 self.matrix00.returnPressed.connect(self.change\_focus)  
 self.matrix01.returnPressed.connect(self.change\_focus)  
 self.matrix10.returnPressed.connect(self.change\_focus)  
 self.matrix11.returnPressed.connect(self.apply)  
  
 # unapply transformation  
 self.ResetButton.clicked.connect(self.reset)  
  
 # adding and changing vector using coords of grid  
 self.AddVector.clicked.connect(self.add\_vector)  
 self.Vector\_x.returnPressed.connect(self.change\_focus)  
 self.Vector\_y.returnPressed.connect(self.add\_vector)  
 self.ChangeVectorRealCords.clicked.connect(self.change\_vector\_real\_cords)  
  
 # adding and changing vector using basis vector  
 self.AddVector\_rel.clicked.connect(self.add\_vector\_rel)  
 self.Vector\_x\_rel.returnPressed.connect(self.change\_focus)  
 self.Vector\_y\_rel.returnPressed.connect(self.add\_vector\_rel)  
 self.ChangeVectorRelativeCords.clicked.connect(self.change\_vector\_relative\_cords)  
  
 # deleting vectors  
 self.green\_btn.clicked.connect(self.delete\_vector)  
 self.blue\_btn.clicked.connect(self.delete\_vector)  
 self.orange\_btn.clicked.connect(self.delete\_vector)  
 self.yellow\_btn.clicked.connect(self.delete\_vector)  
 self.purple\_btn.clicked.connect(self.delete\_vector)  
  
 self.change\_focus\_map = {self.matrix00: self.matrix01,  
 self.matrix01: self.matrix10,  
 self.matrix10: self.matrix11,  
 self.Vector\_x: self.Vector\_y,  
 self.Vector\_x\_rel: self.Vector\_y\_rel}  
  
 # function  
 self.validator = Validator(self)  
 self.function.setValidator(self.validator)  
 self.draw\_btn.clicked.connect(self.draw\_function)  
 self.rotate\_btn.clicked.connect(self.rotate)

Дизайн окна для PyQt сделал в Designer и подключил ui файл шаблона. Кнопкам в окне назначим функции, которые реализуем позже.

5.0

Добавление векторов и матрицы преобразований

Сначала создадим декоратор, с помощью которого будем отслеживать и обрабатывать исключения:

def check\_is\_vector\_grabbed(method):  
 def wrapped(self):  
 try:  
 self.ErrorLog.setText('')  
 return method(self)  
 except Exception as err:  
 self.ErrorLog.setText('Vector not found. Choose vector.')  
 return wrapped

Добавим возможность создавать менять векторы, причём как через относительную систему отсчёта, так и изначальную. Всего можно создать 5 пользовательских векторов. В классе Grid добавим:

# add vector by changing transformed coords  
def add\_vector(self, coords):  
 if not self.available\_colors:  
 raise TooManyVectors()  
 untransformed\_cords = np.linalg.inv(self.get\_matrix()).dot(coords)  
 self.vectors.append(  
 Vector(untransformed\_cords, self.cell\_size, self.central\_vertex, self.available\_colors.pop(-1)))

# add vector(v) by coordinates using basis vectors i and j, where  
# v = ai, bj, where (a, b) - coords   
def add\_vector\_rel(self, coords):  
 if not self.available\_colors:  
 raise TooManyVectors()  
 self.vectors.append(Vector(coords, self.cell\_size, self.central\_vertex, self.available\_colors.pop(-1)))

# return what vector we grabbed by mouse  
def check\_grabbing(self, mouse\_cords):  
 for i in range(2):  
 if distance(mouse\_cords, self.basis\_vectors[i].real\_cords()) < 3:  
 return ('basis', i)  
 for i in range(len(self.vectors)):  
 if distance(mouse\_cords, self.vectors[i].real\_cords()) < 3:  
 return ('common', i)  
 return None

Последний метод даёт возможность изменять векторы с помощью курсора.

Добавим также возможность применять линейное преобразование с помощью матрицы:

# get matrix of linear transformation, using basis vectors  
def get\_matrix(self):  
 return np.array([list(self.basis\_vectors[0].transformed), list(self.basis\_vectors[1].transformed)]).T

# apply linear transformation to all vertexes and vectors  
def transform(self):  
 matrix = self.get\_matrix()  
 length = range(len(self.top\_vertexes))  
 for i in length:  
 self.top\_vertexes[i].transform(matrix)  
 for i in length:  
 self.bot\_vertexes[i].transform(matrix)  
 for i in length:  
 self.left\_vertexes[i].transform(matrix)  
 for i in length:  
 self.right\_vertexes[i].transform(matrix)  
 for i in range(len(self.vectors)):  
 self.vectors[i].transform(matrix)  
 for i in range(len(self.function\_dots)):  
 self.function\_dots[i].transform(matrix)

Теперь в класс MainWindow добавим методы, с помощью которых можно изменять класс Grid – применить/отменить преобразование, добавить/изменть/удалить вектор. Некоторые из них:

# get matrix and apply transformation  
@checking\_exceptions  
def apply(self):  
 matrix = np.array([[float(self.matrix00.text().replace(',', '.')), float(self.matrix01.text().replace(',', '.'))],  
 [float(self.matrix10.text().replace(',', '.')), float(self.matrix11.text().replace(',', '.'))]])  
 M = matrix.dot(transformed\_grid.get\_matrix())  
 for i in range(2):  
 transformed\_grid.basis\_vectors[i].transform(M)  
 transformed\_grid.transform()  
 update\_information(self)  
  
# use grid coords  
@checking\_exceptions  
def add\_vector(self):  
 coords = [float(self.Vector\_x.text()), float(self.Vector\_y.text())]  
 transformed\_grid.add\_vector(coords)  
 transformed\_grid.transform()  
 transformed\_grid.prev\_vector = ('common', -1)  
 update\_information(self)

# x-button  
@checking\_exceptions  
def delete\_vector(self):  
 color = self.sender().objectName().split('\_')[0]  
 for num, v in enumerate(transformed\_grid.vectors):  
 if v.color == COLORS[color]:  
 transformed\_grid.available\_colors.append(COLORS[color])  
 del transformed\_grid.vectors[num]  
 transformed\_grid.prev\_vector = None  
 break

# unapply all transformations  
@checking\_exceptions  
def reset(self):  
 # identity matrix  
 matrix = np.array([[1, 0], [0, 1]])  
 for i in range(2):  
 transformed\_grid.basis\_vectors[i].transform(matrix)  
 transformed\_grid.transform()  
 update\_information(self)

6.0

Математическое решение проблемы поворота графиков функций

Допустим есть график некоторой функции: .

И мы применили некоторое линейное преобразование ко всем точкам этой функции. Нам нужно найти каким уравнением описывается новый график.

В случае с растяжением вдоль осей это просто: , где n, k – коэффициенты сжатия-растяжения вдоль оси Oy и Ox соответственно. Однако в случае с поворотом на произвольный угол, сдвигом не просто.

Для общего решения преобразования будем задавать с помощью матрицы. Проведём в каждую точку изначального графика радиус-вектор. Если матрицу умножить на эти векторы, то они будут проведены к каждой точке изменённого графика.

Допустим есть точка ,и ,если провести радиус вектор , а затем умножить на матрицу : , получается, что преобразование, описанное матрицей , переведёт точку .

Если эти точки перевести обратно, то их координаты будут удовлетворять уравнению . Перевести новые точки мы можем, умножив на обратную матрицу : .

Пусть , а , где – произвольная точка искомого уравнения, тогда

Координаты должны удовлетворять , а значит

Это уравнение и задаёт преобразованную функцию.

7.0

Автоматизация решения, создание интерфейса для ввода функций

Добавим в класс Grid метод для инициализации функции:

def init\_function(self):  
 self.function\_dots = []  
 f = self.function.replace('^', '\*\*').lower()  
 for x in np.arange(-20.0, 20.0, 0.005):  
 try:  
 y = eval(f)  
 self.function\_dots.append(Vertex((x, y), self.cell\_size, (self.x, self.y)))  
 except Exception as err:  
 pass

Изменим метод *transform*:

# apply linear transformation to all vertexes and vectors  
def transform(self):  
 matrix = self.get\_matrix()  
 length = range(len(self.top\_vertexes))  
 for i in length:  
 self.top\_vertexes[i].transform(matrix)  
 for i in length:  
 self.bot\_vertexes[i].transform(matrix)  
 for i in length:  
 self.left\_vertexes[i].transform(matrix)  
 for i in length:  
 self.right\_vertexes[i].transform(matrix)  
 for i in range(len(self.vectors)):  
 self.vectors[i].transform(matrix)  
 for i in range(len(self.function\_dots)):  
 self.function\_dots[i].transform(matrix)  
 f = '{} = ' + self.function.replace('x', '{}')  
 # tranform dot coords to coords relatively to another system  
 # in this coordinates  
 try:  
 param = np.linalg.inv(matrix)  
 param = [[str(round(j, 3)) for j in i] for i in param]  
 n = f.count('{}') - 1  
 self.transformed\_function = f.format('(' + param[1][0] + '\*x + ' + param[1][1] + '\*y)',  
 \*(['(' + param[0][0] + '\*x + ' + param[0][1] + '\*y)']\*n))  
 except Exception as err:  
 self.transformed\_function = ''

Добавим в MainWindow метод ввода функции:

# add dots to plane and init  
@checking\_exceptions  
def draw\_function(self):  
 transformed\_grid.function = self.function.text()  
 transformed\_grid.transformed\_function = 'y = ' + self.function.text()  
 transformed\_grid.init\_function()  
 transformed\_grid.transform()  
 self.rotated\_function.setText('y = ' + transformed\_grid.function)

8.0

Тестирование и сборка проекта

Убедимся в работе программы на примере функции y = x^2. Если сжимать-растягивать вдоль оси Ox и Oy получим уравнения, которые можем легко построить и убедиться в их правильности. Если повернём функцию на 45 градусов вправо (-45) получим уравнение:

Можно проверить с помощью Geogebra.

Сборку проекта осуществим с помощью pyinstaller. В терминале в папке проекта введём команду:

pyinstaller -w main.py

В dist/main перенесём файл main\_window.ui для корректной работы программы.

Исполняемый файл: dist/main/main.exe