Задача №1

Задача: Создать простейший пользовательский интерфейс калькулятора, позволяющий пользователю ввести два числа и произвести математические операции: сложение, вычитание, умножение и деление, результат отображается в специальном окне вывода.

Математическая модель: соответствует поставленной задаче.

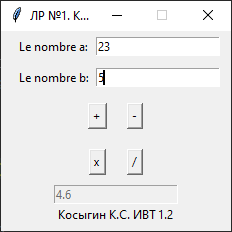
Список идентификаторов в программе:

|  |  |  |
| --- | --- | --- |
| Имя переменной в программе | Описание переменной | Тип данных |
| event |  | <class tkinter.Event> |
| btn\_txt | Текст, выводимый в кнопке с мат. операцией | str |
| a | Число a | float |
| b | Число b | float |
| OPERATIONS | Словарь с функциями для мат. операций | dict |

Код программы:

OPERATIONS = {  
 '+': lambda a, b: a + b,  
 '-': lambda a, b: a - b,  
 'x': lambda a, b: a \* b,  
 '/': lambda a, b: a / b,  
}  
  
  
def calculate(event):  
 *"""Calculate and display"""* btn\_txt = event.widget['text']  
  
 a = float(entry\_a.get())  
 b = float(entry\_b.get())  
  
 res = OPERATIONS[btn\_txt](a, b)  
  
 result\_entry.configure(state='normal')  
 result\_entry.delete(first=0, last=tk.END)  
 result\_entry.insert(tk.END, res)  
 result\_entry.configure(state='disabled')

Протокол работы программы:



Задача №2

Задача: Нарисовать треугольник, одна сторона которого черная, другая красная и третья – синяя. Создать программу, рисующую треугольник, вершины которого можно задать через пользовательский интерфейс.

Математическая модель:

xs, ys – расстояния от точки до осей x и y соответственно в экранной системе координат; xD, yD - расстояния от точки до осей x и y соответственно в Декартовой системе координат; offset – унарный терм, определяющий смещение по осям x и y соответственно в пикселах до центра дисплея; scale – унарный терм, определяющий коэффициент масштабирования по осям x и y соответственно.

Список идентификаторов в программе:

|  |  |  |
| --- | --- | --- |
| Имя переменной в программе | Описание переменной | Тип данных |
| line | Две точки, образующие линию | tuple |
| color | RGB значение цвета | str |
| DEFAULT\_TRIANGLE\_LINES | Список из линий, образующих базовый треугольник с разноцветными сторонами | list |
| COLORS | Список из цветов для сторон базового треугольника | list |
| pd1 | Первая точка линии в Декартовой системе координат | tuple |
| pd2 | Вторая точка линии в Декартовой системе координат | tuple |
| ps1 | Первая точка линии в экранной системе координат | tuple |
| ps2 | Вторая точка линии в экранной системе координат | tuple |
| d2s | Функция, переводящая точку из Декартовой системы координат в экранную | <class ‘function’> |
| point | Точка в Декартовой системе координат | tuple |
| window\_size | Ширина и высота дисплея в пикселах, необходимая для расчета смещения точки в экранной системе координат | tuple |
| scale | Коэффициенты масштаба | tuple |
| x | Координата по оси x | float |
| y | Координата по оси y | float |
| z | Координата по оси z | float |
| width | Высота окна в пикселах | int |
| height | Ширина окна в пикселах | int |
| xscale | Коэффициент масштаба по оси x | float |
| yscale | Коэффициент масштаба по оси y | float |
| x\_mid | Центр дисплея по ширине | int |
| y\_mid | Центр дисплея по высоте | int |
| BASE\_SCALE | Базовые коэффициенты масштабирования | tuple |
| canvas | Виджет canvas, используемые для рисования | <Class tkinter. Canvas > |
| users\_input | Строка, введенная пользователем, содержащая точки для построения треугольника | str |
| points | Точки для построения треугольника | list |
| p1 | Первая вершина треугольника | tuple |
| p2 | Вторая вершина треугольника | tuple |
| p3 | Третья вершина треугольника | tuple |
| parse\_points | Функция, вычленяющая из строки, введенной пользователем цифровые значения | <class ‘function’> |
| string | Строка для анализа | str |
| nums | Список из вычлененных чисел | list |
| t | Текущий токен | str |
| ch | Текущий символ из строки | str |

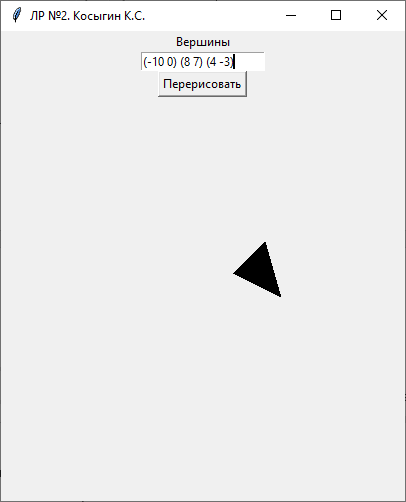
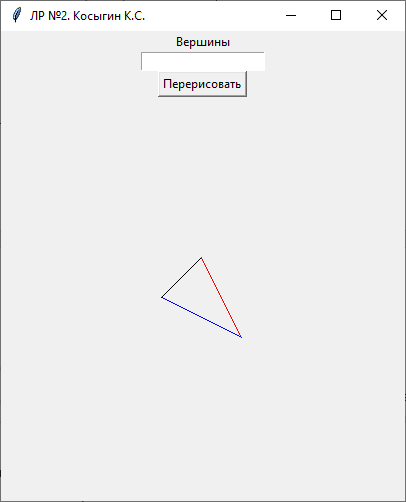
Код программы:

DIGITS = '0123456789'  
  
DEFAULT\_TRIANGLE\_LINES = [  
 ((-5, 0), (0, 5)),  
 ((0, 5), (5, -5)),  
 ((5, -5), (-5, 0))  
]  
  
COLORS = ['#000000', '#FF0000', '#0000FF']  
  
WINDOW\_SIZE = (400, 400)  
  
  
def draw\_default\_triangle():  
 for line, color in zip(DEFAULT\_TRIANGLE\_LINES, COLORS):  
 pd1, pd2 = line  
 ps1 = d2s(pd1, WINDOW\_SIZE)  
 ps2 = d2s(pd2, WINDOW\_SIZE)  
 canvas.create\_line(\*ps1, \*ps2, fill=color)  
  
  
def draw(event):  
 *"""Redraw content of the canvas. Display errors in the canvas"""* canvas.delete('all')  
  
 users\_input = entry\_points.get()  
 points = parse\_points(users\_input)  
  
 if len(points) != 6:  
 canvas.create\_text(X\_MID, Y\_MID, text='Некорректные вершины')  
 return  
  
 p1 = d2s(tuple(points[:2]), WINDOW\_SIZE)  
 p2 = d2s(tuple(points[2:4]), WINDOW\_SIZE)  
 p3 = d2s(tuple(points[4:]), WINDOW\_SIZE)  
 canvas.create\_polygon(\*p1, \*p2, \*p3)  
  
  
def parse\_points(string):  
 nums = []  
 t = ''  
 for ch in string:  
 if ch in DIGITS:  
 t += ch  
 elif t != '':  
 nums.append(int(t))  
 t = ''  
 if t != '':  
 nums.append(int(t))  
 return nums

BASE\_SCALE = (8, 8)

def d2s(point: tuple, window\_size: tuple, scale=None):  
 *"""Convert Descartes point to screen point."""* if len(point) == 2:  
 x, y = point  
 elif len(point) == 3:  
 x, y, z = point  
 else:  
 return point  
  
 # *TODO: z axis* width, height = window\_size  
 xscale, yscale = BASE\_SCALE if scale is None else scale  
  
 x\_mid = width // 2  
 y\_mid = height // 2  
 x = x\_mid + int(x \* xscale)  
 y = y\_mid - int(y \* yscale)  
  
 return x, y

Протокол работы программы:



Задача №3

Задача: Разработать алгоритм построения графика заданной функции в заданном диапазоне.

Математическая модель: см. Задача №2 (перевод декартовых координат в экранные).

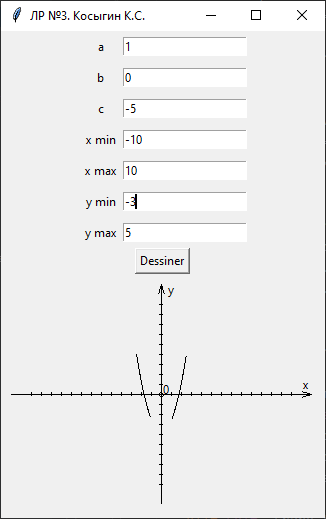
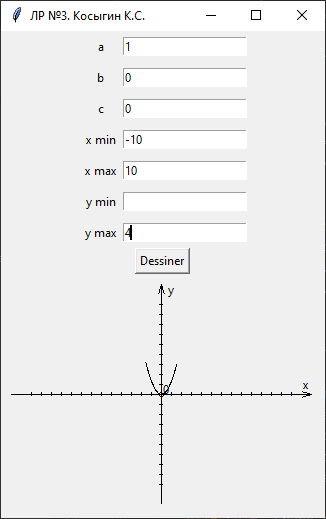
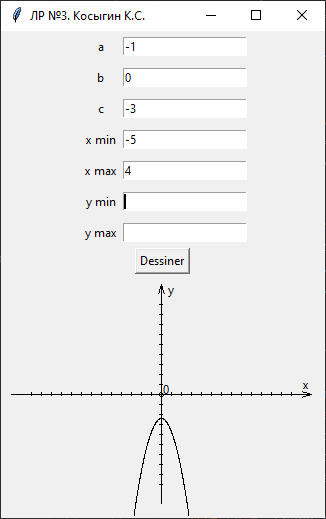
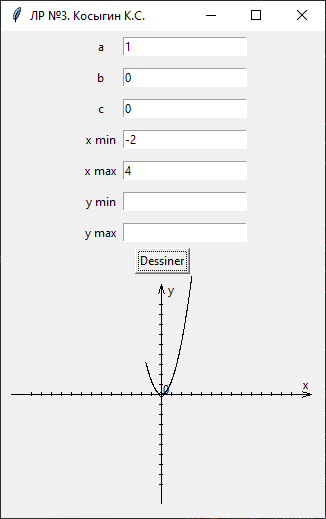
Список идентификаторов в программе:

Код программы:

from converter import d2s  
from axis import draw\_axis  
  
  
# Canvas parameters  
CANVAS\_SIZE = (320, 240)  
AXIS\_SCALE = (10, 10) # Attention to d2s!!!  
  
SERIF\_SIZE = 4  
AXIS\_INDENT = 10  
ARROW\_INDENT = 3  
  
NUM\_OF\_STEPS = 10\*\*3 # For function  
  
  
def func(x, a, b, c):  
 *"""Parabola function"""* return a \* x\*\*2 + b \* x + c  
  
  
# Counted constants  
WIDTH, HEIGHT = CANVAS\_SIZE  
X\_SCALE, Y\_SCALE = AXIS\_SCALE  
SERIF\_MID = SERIF\_SIZE // 2  
MID\_X = WIDTH // 2  
MID\_Y = HEIGHT // 2  
  
  
def get\_users\_input():  
 params = {}  
 try:  
 params['a'] = float(entry\_a.get())  
 params['b'] = float(entry\_b.get())  
 params['c'] = float(entry\_c.get())  
  
 params['x\_min'] = float(entry\_x\_min.get())  
 params['x\_max'] = float(entry\_x\_max.get())  
 except ValueError:  
 canvas.delete('all')  
 canvas.create\_text(MID\_X, MID\_Y, text='Некорректные значения')  
  
 y\_min = entry\_y\_min.get()  
 y\_max = entry\_y\_max.get()  
 if y\_min != '':  
 params['y\_min'] = y\_min  
 if y\_max != '':  
 params['y\_max'] = y\_max  
  
 return params  
  
  
def draw\_graphic(params: dict):  
 x\_max = params.get('x\_max')  
 x\_min = params.get('x\_min')  
 a = params.get('a')  
 b = params.get('b')  
 c = params.get('c')  
  
 y\_min = params.get('y\_min')  
 y\_max = params.get('y\_max')  
  
 h = (x\_max - x\_min) / NUM\_OF\_STEPS # literally dx  
  
 # Functional point  
 x = x\_min  
 y = func(x, a, b, c)  
  
 # Graphical point  
 point1 = d2s((x, y), CANVAS\_SIZE)  
  
 # Count points for function nad convert them  
 x += h  
 while x < x\_max:  
 y = func(x, a, b, c)  
  
 # New screen point  
 point2 = d2s((x, y), CANVAS\_SIZE)  
  
 # There is y(min) and the current f(x) is lower than that minimum  
 # OR  
 # There is y(max) and the current f(x) is bigger than that maximum  
 if y\_min is not None and y < float(y\_min) or y\_max is not None and y > float(y\_max):  
 pass  
  
 # The graphic must be drawn  
 else:  
 canvas.create\_line(\*point1, \*point2)  
  
 point1 = point2  
 x += h  
  
  
def redraw(event):  
 *"""Clear, Draw axis, Get user's input, Draw graphic"""* # Clear  
 canvas.delete('all')  
 # Draw axis  
 draw\_axis(canvas, 2, CANVAS\_SIZE, AXIS\_INDENT, ARROW\_INDENT, AXIS\_SCALE)  
 # Get user's input  
 users\_input = get\_users\_input()  
 # Draw graphic  
 if event is not None:  
 draw\_graphic(users\_input)

import tkinter as tk  
import numpy as np  
  
  
from converter import d2s, TRANSFORMATIONS  
  
  
AXIS\_NAMES = ['x', 'y', 'z']  
  
  
DEFAULT\_CANVAS\_SIZE = (480, 320)  
  
# Indent from the borders, in other words - padding  
DEFAULT\_AXIS\_INDENT = 10  
# Rise of arrows  
DEFAULT\_ARROW\_INDENT = 3  
# Descartes to screen  
DEFAULT\_AXIS\_SCALE = (8, 8)  
  
  
def draw\_axis(  
 canvas: 'tk.Canvas', num\_of\_axis: int,  
 canvas\_size=None,  
 axis\_indent=None,  
 arrow\_indent=None,  
 axis\_scale=None  
):  
 *"""Draw all stuff needed for axis"""* canvas\_size = DEFAULT\_CANVAS\_SIZE if canvas\_size is None else canvas\_size  
 axis\_indent = DEFAULT\_AXIS\_INDENT if axis\_indent is None else axis\_indent  
 arrow\_indent = DEFAULT\_ARROW\_INDENT if arrow\_indent is None else arrow\_indent  
 axis\_scale = DEFAULT\_AXIS\_SCALE if axis\_scale is None else axis\_scale  
  
 draw\_zero(canvas, canvas\_size, axis\_scale)  
 draw\_lines(canvas, num\_of\_axis, canvas\_size, axis\_indent)  
 draw\_arrows(canvas, num\_of\_axis, canvas\_size, axis\_indent, arrow\_indent)  
 draw\_axis\_names(canvas, num\_of\_axis, canvas\_size, axis\_indent)  
 draw\_serifs(canvas, num\_of\_axis, canvas\_size, axis\_indent, axis\_scale)  
  
  
def draw\_zero(canvas: 'tk.Canvas', canvas\_size: tuple, axis\_scale: tuple):  
 width, height = canvas\_size  
 x\_offset, y\_offset = width // 2, height // 2  
 x\_scale, y\_scale = axis\_scale  
 x = x\_offset + x\_scale // 2  
 y = y\_offset - y\_scale // 2  
 # Text  
 canvas.create\_text(x, y, text='0')  
 # Point  
 r = x\_scale // 5  
 canvas.create\_oval(  
 x\_offset - r, y\_offset - r,  
 x\_offset + r, y\_offset + r  
 )  
  
  
def draw\_lines(  
 canvas: 'tk.Canvas', num\_of\_axis: int,  
 canvas\_size=DEFAULT\_CANVAS\_SIZE, axis\_indent=DEFAULT\_AXIS\_INDENT):  
 *"""Draw lines forming axis"""* width, height = canvas\_size  
 x\_offset, y\_offset = width // 2, height // 2  
  
 if num\_of\_axis > 0:  
 # X  
 line = [  
 axis\_indent, y\_offset,  
 width - axis\_indent, y\_offset  
 ]  
 canvas.create\_line(\*line)  
 if num\_of\_axis > 1:  
 # Y  
 line = [  
 x\_offset, axis\_indent,  
 x\_offset, height - axis\_indent  
 ]  
 canvas.create\_line(\*line)  
  
  
def draw\_arrows(  
 canvas: 'tk.Canvas', num\_of\_axis: int,  
 canvas\_size=DEFAULT\_CANVAS\_SIZE,  
 axis\_indent=DEFAULT\_AXIS\_INDENT,  
 arrow\_indent=DEFAULT\_ARROW\_INDENT  
):  
 *"""Draw arrows indicating direction"""* width, height = canvas\_size  
 x\_offset, y\_offset = width // 2, height // 2  
  
 if num\_of\_axis > 0:  
 # X  
 line = [  
 width - axis\_indent, y\_offset,  
 width - 2\*axis\_indent, y\_offset - arrow\_indent  
 ]  
 canvas.create\_line(\*line)  
 line[3] = y\_offset + arrow\_indent  
 canvas.create\_line(\*line)  
  
 if num\_of\_axis > 1:  
 line = [  
 x\_offset, axis\_indent,  
 x\_offset - arrow\_indent, axis\_indent \* 2  
 ]  
 canvas.create\_line(\*line)  
 line[2] = x\_offset + arrow\_indent  
 canvas.create\_line(\*line)  
  
  
def draw\_axis\_names(  
 canvas: 'tk.Canvas', num\_of\_axis: int,  
 canvas\_size=DEFAULT\_CANVAS\_SIZE,  
 axis\_indent=DEFAULT\_AXIS\_INDENT,  
):  
 *"""Draw axis names"""* width, height = canvas\_size  
 x\_offset, y\_offset = width // 2, height // 2  
  
 axis\_name = 'x'  
 if num\_of\_axis > 0:  
 # X  
 x = width - int(1.5 \* axis\_indent)  
 y = y\_offset - axis\_indent  
 canvas.create\_text(x, y, text=axis\_name)  
 axis\_name = 'y'  
 if num\_of\_axis > 1:  
 # Y  
 x = x\_offset + axis\_indent  
 y = int(1.5 \* axis\_indent)  
 canvas.create\_text(x, y, text=axis\_name)  
  
  
def draw\_serifs(  
 canvas: 'tk.Canvas', num\_of\_axis: int,  
 canvas\_size=DEFAULT\_CANVAS\_SIZE,  
 axis\_indent=DEFAULT\_AXIS\_INDENT,  
 axis\_scale=DEFAULT\_AXIS\_SCALE  
):  
 *"""Draw serifs. Reflection\_type in REFLECTIONS"""* width, height = canvas\_size  
 x\_offset, y\_offset = width // 2, height // 2  
 x\_scale, y\_scale = axis\_scale  
  
 def reflector(  
 reflection\_type: str,  
 serif: list,  
 offset: int, length: int, scale: int  
 ):  
 reflection\_matrix = TRANSFORMATIONS[reflection\_type] # reflect at y  
  
 p1 = d2s(tuple(serif[0]), canvas\_size, axis\_scale)  
 p2 = d2s(tuple(serif[1]), canvas\_size, axis\_scale)  
  
 axis\_bias = offset + scale  
  
 while axis\_bias < length - 2\*axis\_indent:  
 canvas.create\_line(\*p1, \*p2)  
  
 reflected\_serif = np.matmul(  
 np.array(serif), np.array(reflection\_matrix)  
 ).tolist()  
 p1 = d2s(reflected\_serif[0], canvas\_size, axis\_scale)  
 p2 = d2s(reflected\_serif[1], canvas\_size, axis\_scale)  
 canvas.create\_line(\*p1, \*p2)  
  
 # Cycle parameters  
 axis\_bias += scale  
 if reflection\_type == 'ry':  
 serif[0][0] += 1  
 serif[1][0] += 1  
 elif reflection\_type == 'rx':  
 serif[0][1] += 1  
 serif[1][1] += 1  
  
 p1 = d2s(tuple(serif[0]), canvas\_size, axis\_scale)  
 p2 = d2s(tuple(serif[1]), canvas\_size, axis\_scale)  
  
 if num\_of\_axis > 0:  
 x\_serif = [ # Descartes  
 [1, 0.2],  
 [1, -0.2]  
 ]  
 reflector('ry', x\_serif, x\_offset, width, x\_scale)  
 if num\_of\_axis > 1:  
 y\_serif = [ # Descartes  
 [-0.2, 1],  
 [0.2, 1]  
 ]  
 reflector('rx', y\_serif, y\_offset, height, y\_scale)

Протокол работы программы:



Задача №4

Задача: Разработать алгоритм и программу решения прикладной задачи – построение траектории движения тела, брошенного под углом к горизонту.

Математическая модель:

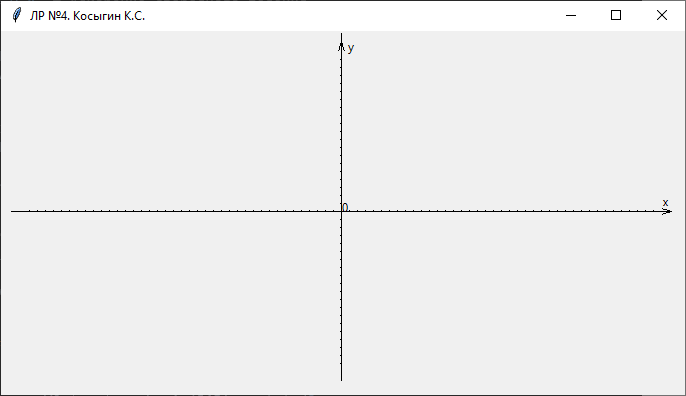
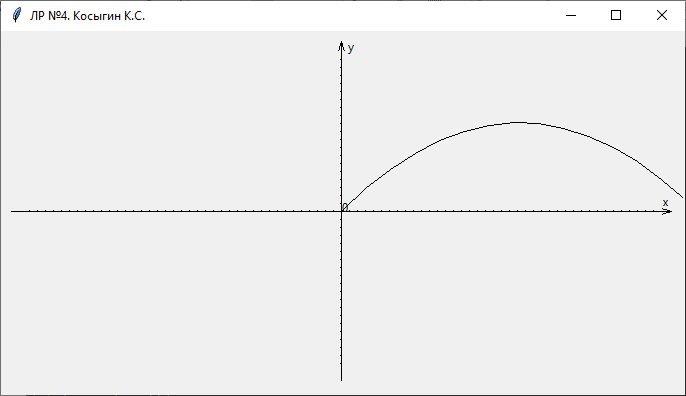
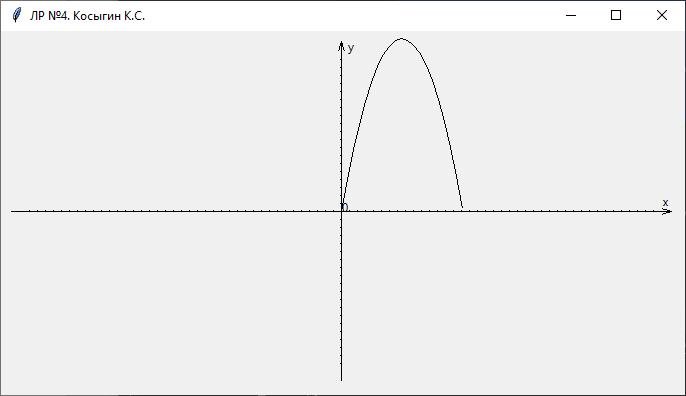
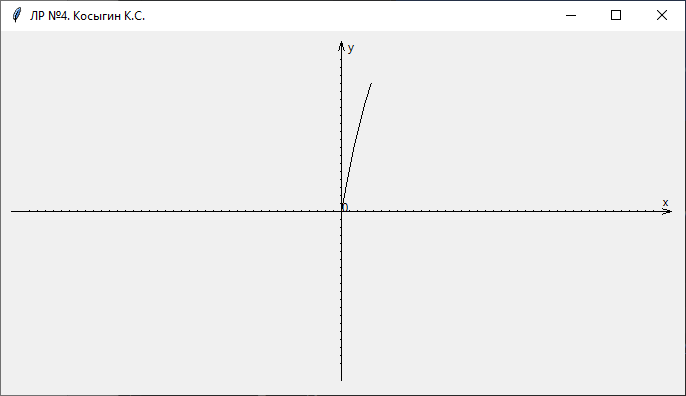
X, y – координаты точки в текущий момент времени t ; V0 – начальная скорость ; g – ускорение свободного падения.

Список идентификаторов в программе:

Код программы:

from converter import d2s  
from axis import draw\_axis  
  
  
# Number of frames  
N = 50  
  
V0 = 100 # Начальная скорость  
alpha = 80 # Угол  
g = 9.8 # Ускорение свободного падения  
  
  
CANVAS\_SIZE = (680, 360)  
SCALE = (0.35, 0.35)  
  
  
# Counted constants  
CANVAS\_WIDTH, CANVAS\_HEIGHT = CANVAS\_SIZE  
  
T\_START = 0 # Start time  
T\_FINAL = 2\*V0 \* math.sin(math.radians(alpha)) / g # Fly time  
  
# Period between frames  
PERIOD = int(round(T\_FINAL / N, 5) \* 1000)  
  
RAD = math.radians(alpha)  
  
  
def trajectory(t):  
 *"""Return Descartes point at time"""* x = V0 \* t \* math.cos(RAD)  
 y = V0 \* t \* math.sin(RAD) - g\*t\*t/2  
 return x, y  
  
  
def draw\_trajectory(t, xp, yp):  
 xc, yc = d2s(trajectory(t), CANVAS\_SIZE, SCALE)  
  
 if yc < CANVAS\_HEIGHT // 2:  
  
 canvas.create\_line(xp, yp, xc, yc)  
  
 canvas.after(PERIOD, lambda: draw\_trajectory(t+1, xc, yc))

Протокол работы программы:



Задача №5

Задача: Построить правильный n-угольник, количество углов которого задано и вводится через пользовательский интерфейс.

Математическая модель: Пусть x0, y0 – координаты центра, а R – радиус описанной вокруг правильного многоугольника окружности; ϕ0 – угловая координата первой вершины, тогда декартовы координаты вершин правильного n-угольника определяются формулами:

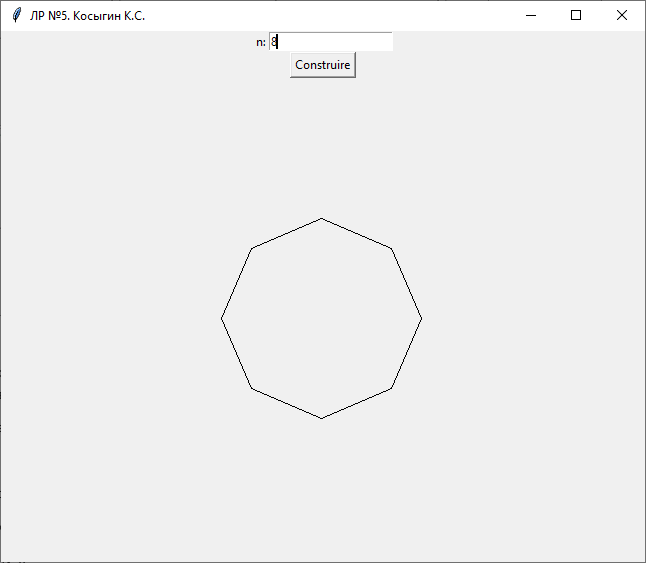
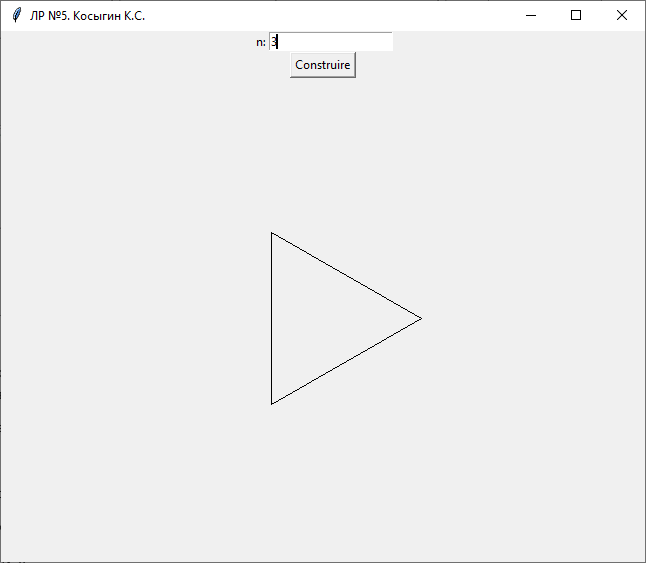
X0, y0 – координаты центра.

Список идентификаторов в программе:

Код программы:

def build\_figure(event):  
 canvas.delete('all')  
  
 n = int(entry\_n.get())  
  
 angle = PHI  
 x0 = RADIUS \* math.cos(angle)  
 y0 = RADIUS \* math.sin(angle)  
 p1 = d2s((x0, y0), CANVAS\_SIZE, SCALE)  
 for i in range(0, n+1):  
 angle = PHI + 2\*math.pi \* i / n  
 xi = round(RADIUS \* math.cos(angle), 5)  
 yi = round(RADIUS \* math.sin(angle), 5)  
  
 p2 = d2s((xi, yi), CANVAS\_SIZE, SCALE)  
  
 canvas.create\_line(\*p1, \*p2)  
  
 p1 = p2

Протокол работы программы:



~~Задача №6~~

~~Задача: Построить произвольный объект в первой четверти системы координат (кол-во вершин не меньше 7) и организовать следующие преобразования объекта:~~

~~- общее преобразование (изменение масштаба, симметричное отражение относительно осей, сдвиг);~~

~~- вращение объекта относительно начала координат;~~

~~- комбинированное преобразование (преобразование общего вида и вращение на произвольный угол относительно начала координат).~~

~~Математическая модель:~~

~~Список идентификаторов в программе:~~

~~Код программы:~~

~~Протокол работы программы:~~

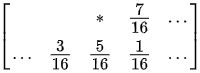
Самостоятельное задание №1

Вариативная часть

Задание №2

Задача: реализовать метод дизеринга в графической системе.

Математическая модель:

, где \* - текущий пиксель, числовые коэффициенты – значения ошибки.

Список идентификаторов в программе:

Код программы:

from math import floor  
  
  
def fs\_dithering(pixels):  
 *"""Floyd-Steiner dithering. Affect passed pixels"""* height = len(pixels)  
 width = len(pixels[0])  
  
 for y in range(1, height):  
 for x in range(1, width):  
 old\_pixel = pixels[x][y]  
 new\_pixel = map\_threshold(old\_pixel)  
  
 errors = [  
 old\_color - new\_color  
 for old\_color, new\_color  
 in zip(old\_pixel, new\_pixel)  
 ]  
  
 if x < width - 1: # Pixel at East  
 pixel = [  
 old\_color + round(err \* 7/16)  
 for old\_color, err  
 in zip(pixels[x+1][y], errors)  
 ]  
 pixels[x+1][y] = pixel  
  
 if x > 1 and y < height - 1: # Pixel at South-West  
 pixel = [  
 old\_color + round(err \* 3/16)  
 for old\_color, err  
 in zip(pixels[x-1][y+1], errors)  
 ]  
 pixels[x-1][y+1] = pixel  
  
 if y < height - 1: # Pixel at South  
 pixel = [  
 old\_color + round(err \* 5/16)  
 for old\_color, err  
 in zip(pixels[x][y+1], errors)  
 ]  
 pixels[x][y+1] = pixel  
  
 if x < width - 1 and y < height - 1: # Pixel at South-East  
 pixel = [  
 old\_color + round(err / 16)  
 for old\_color, err  
 in zip(pixels[x+1][y+1], errors)  
 ]  
 pixels[x+1][y+1] = pixel  
  
  
def map\_threshold(pixel):  
 *"""Map to the closest colors"""* return [255 \* floor(color/128) for color in pixel]

Протокол работы программы: