Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет систем управления и робототехники

Отчет по лабораторной работе №5 «Использование линейного и нелинейного закона управления»

по дисциплине «Введение в профессиональную деятельность»

Выполнили: студенты гр. R3137
Курчавый В.В.
Кирбаба Д.Д.
Кравченко Д.В.

Преподаватель: Перегудин А.А.,
ассистент фак. СУиР

Санкт-Петербург 2020

1. Цель работы

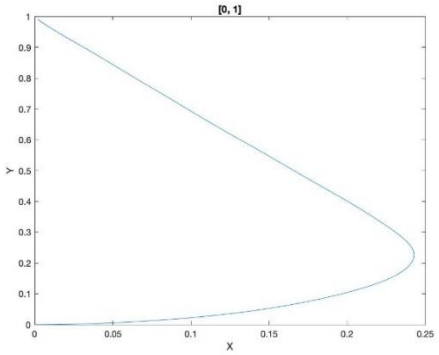
Изучить функцию Ляпунова и использовать её на примере управления роботом для езды в точки с заданными координатами. Сравнить управление с помощью функции Ляпунова и управление с помощью различных видов регуляторов.

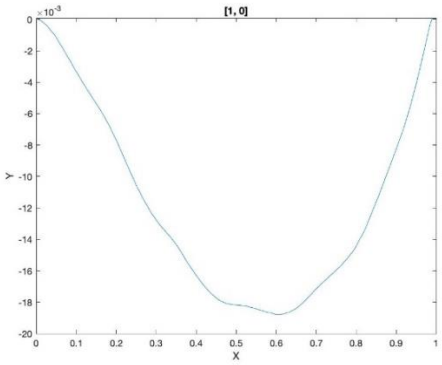
2. Материалы работы

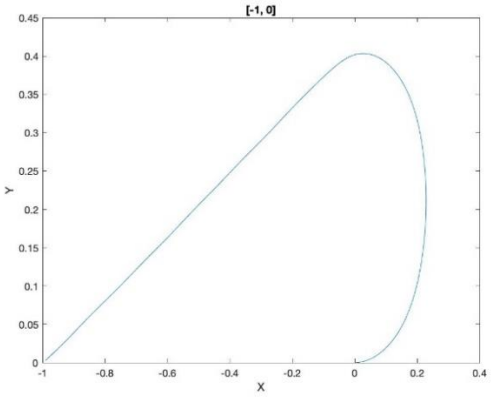
2.1 Результаты необходимых расчетов и построений

Графики координат для езды по точкам из эксперимента:

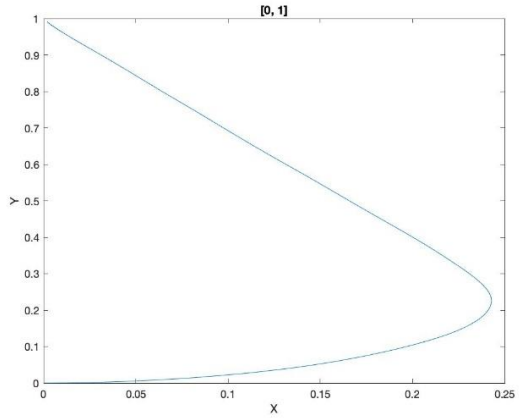
Линейный:

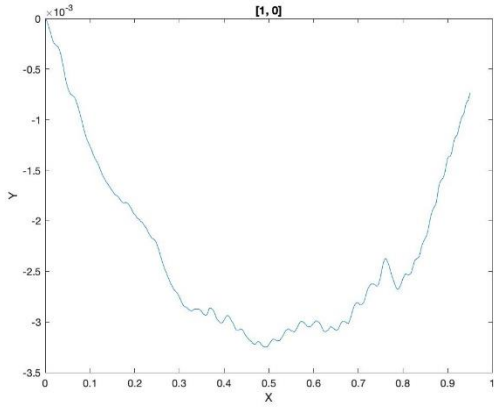


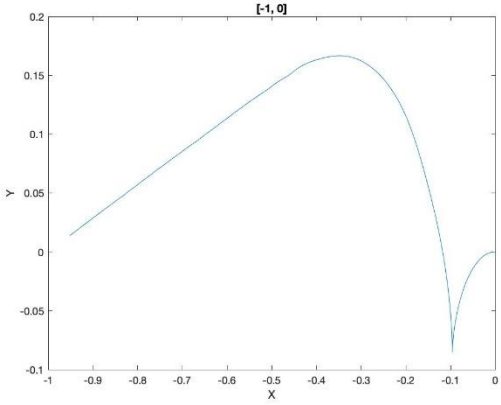




Нелинейный:

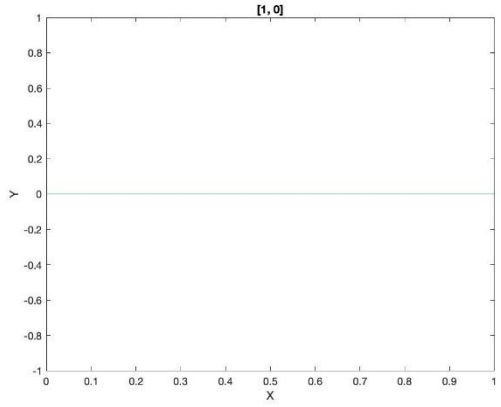


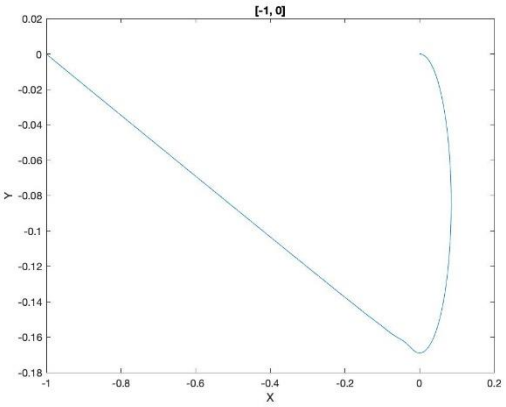


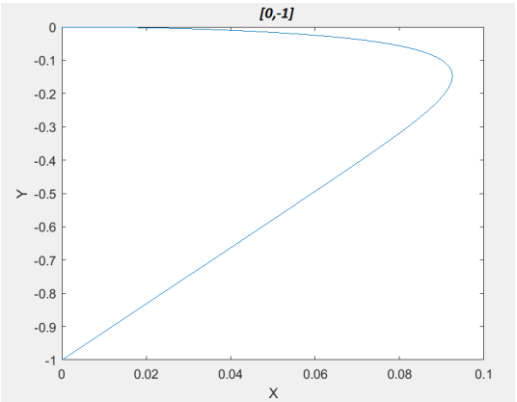


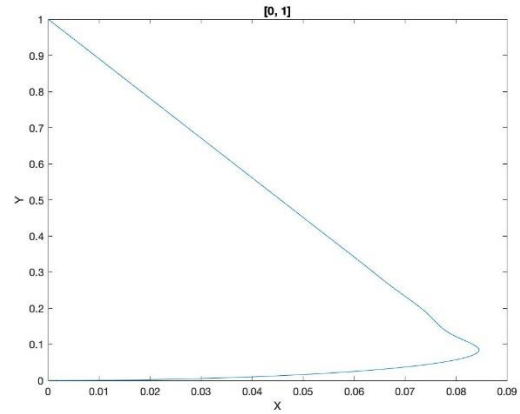
Графики координат для езды по точкам из модели:

Линейный

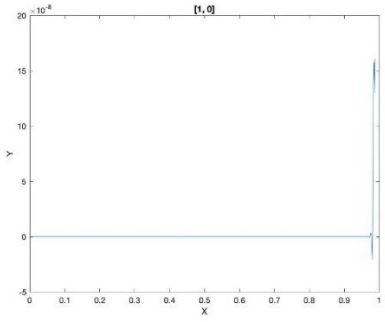


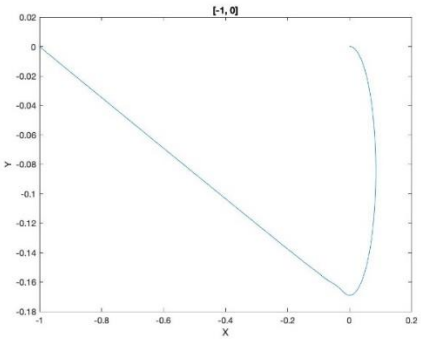


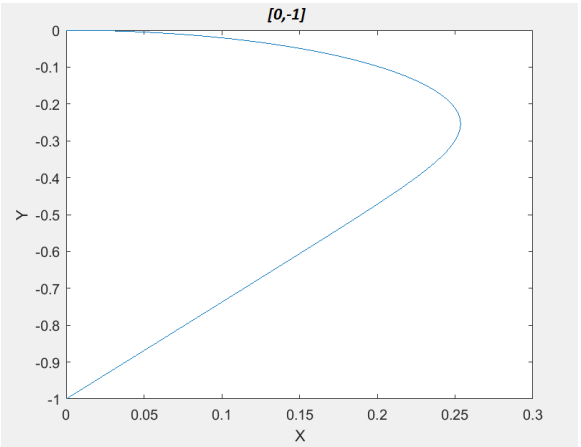


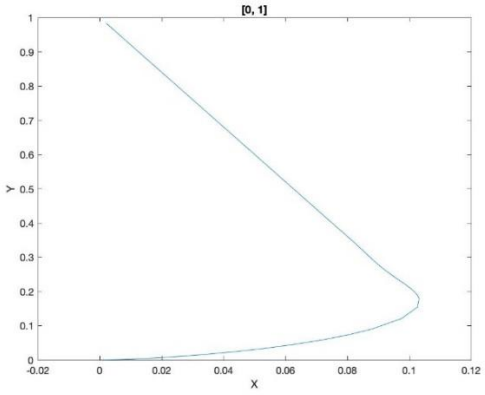


Нелинейный:

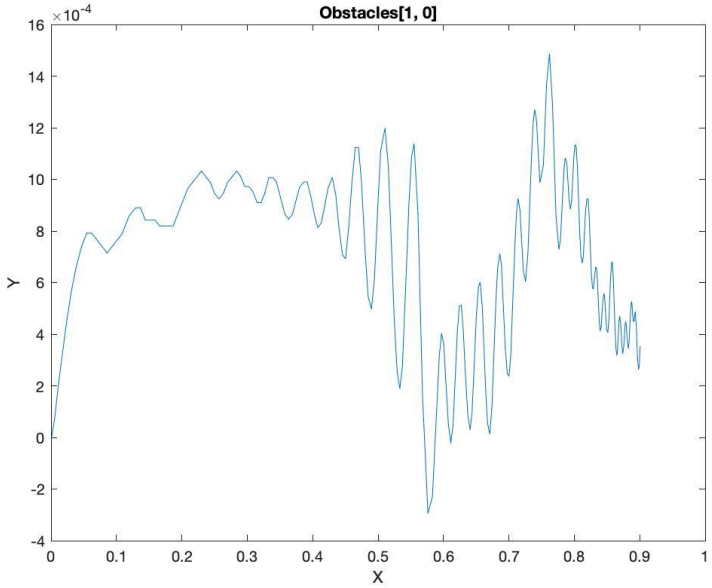


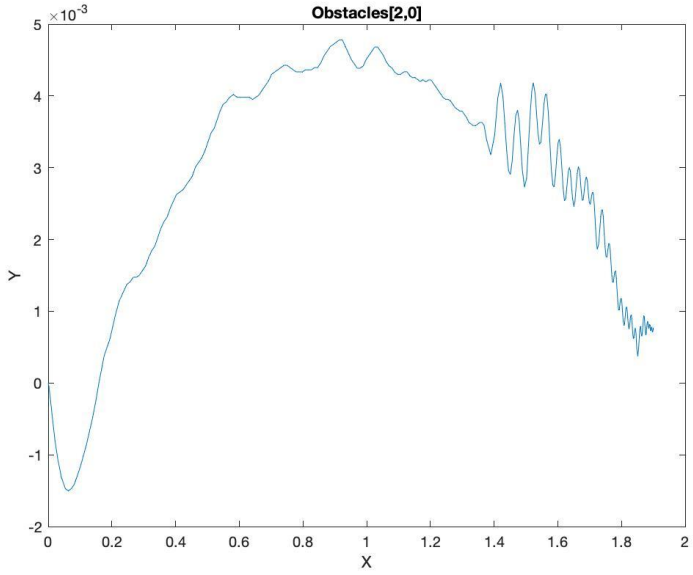


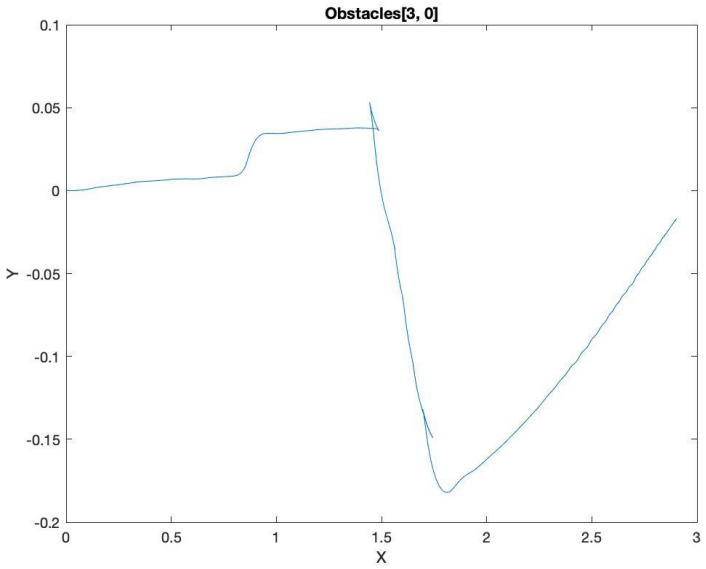




Графики координат при объезде препятствий:

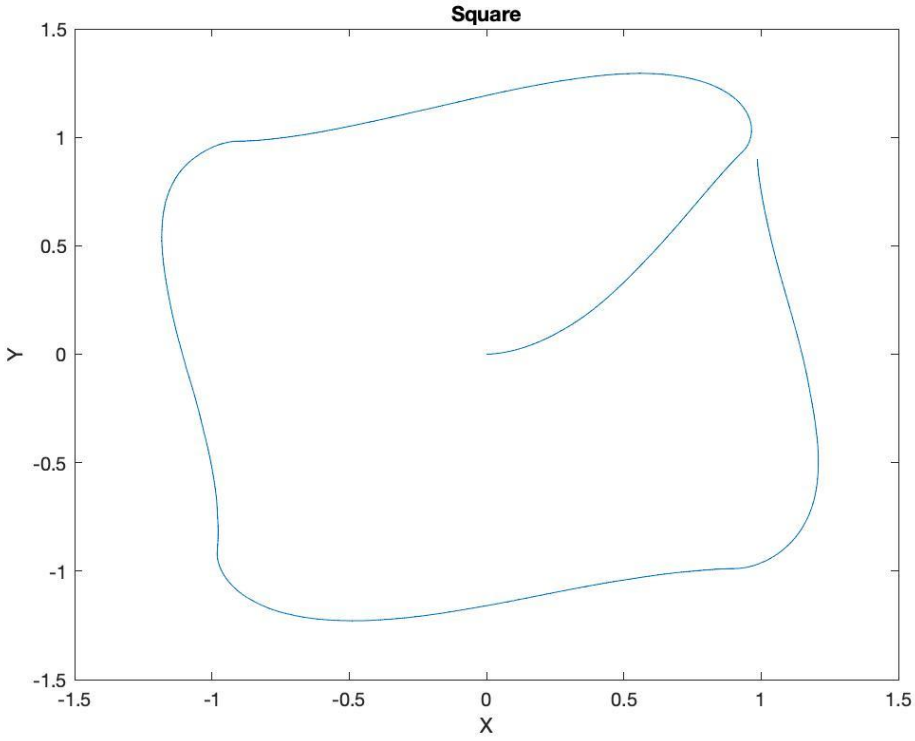




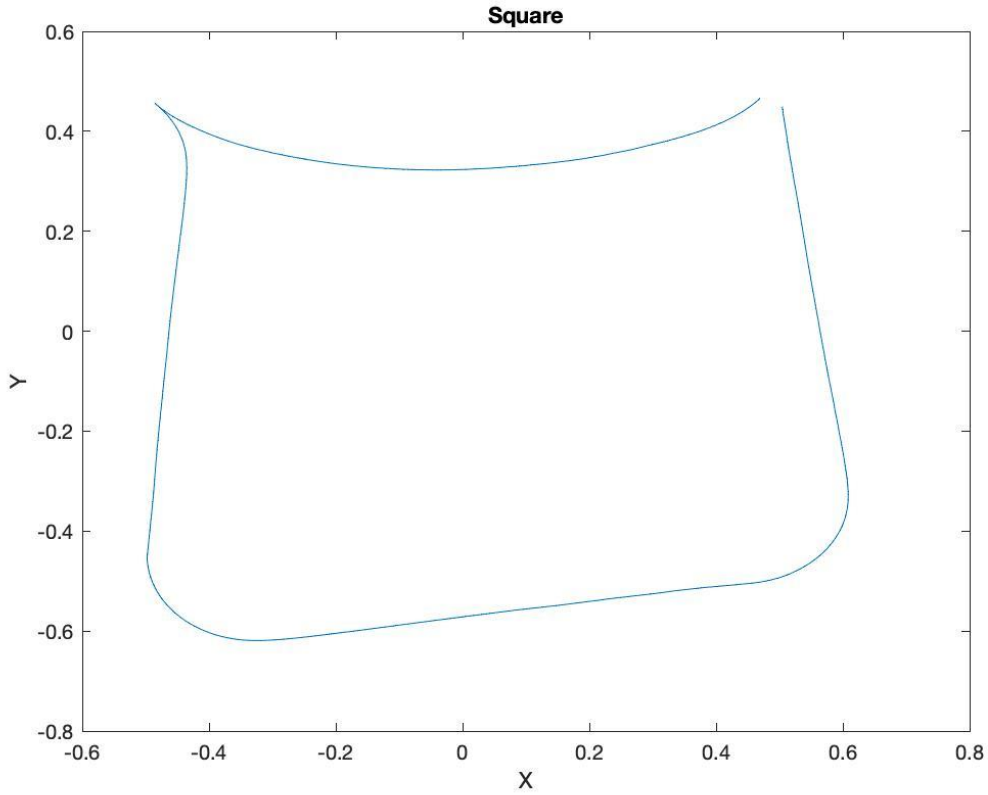


Графики координат для езды по квадрату (эксперимент):

Линейный:

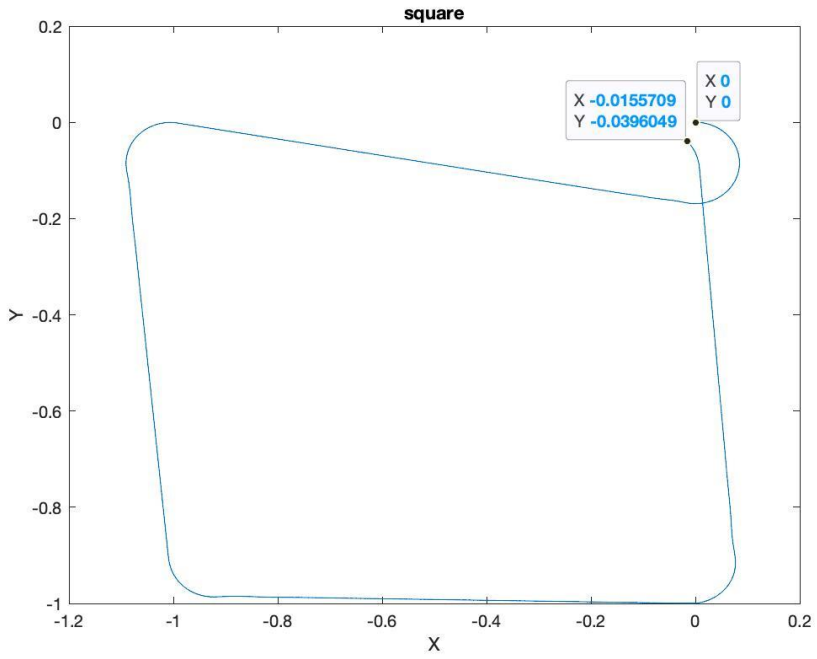


Нелинейный:

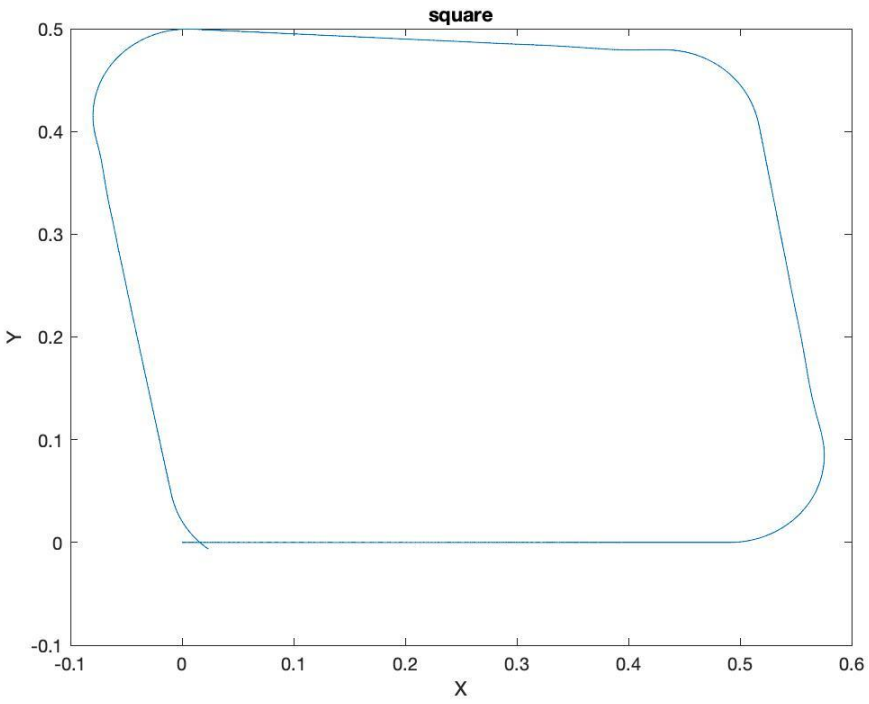


Графики координат для езды по квадрату (модель):

Линейный:

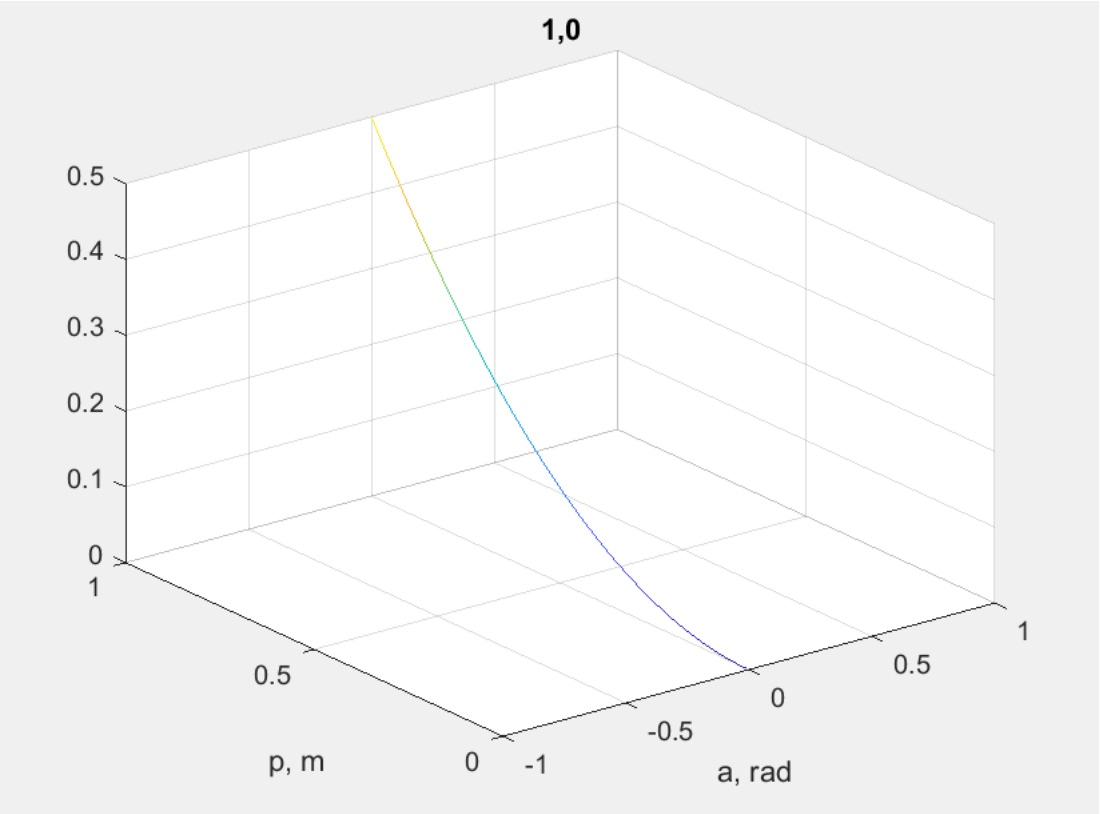


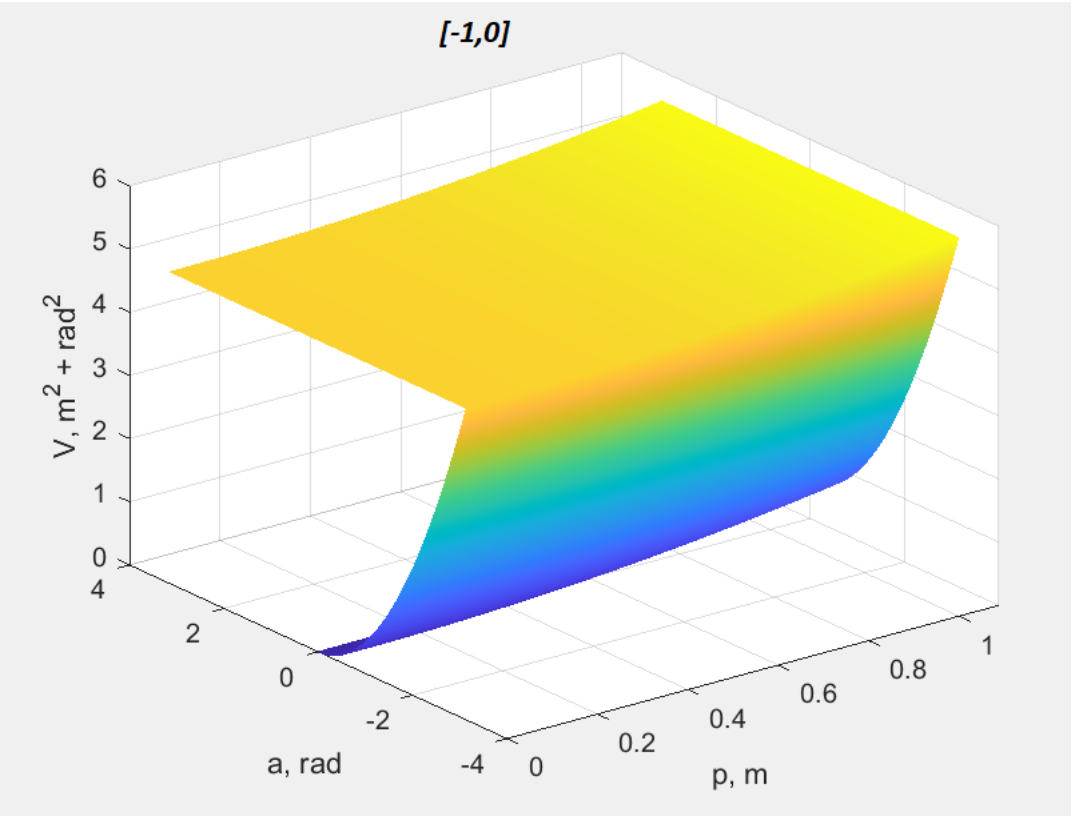
Нелинейный:

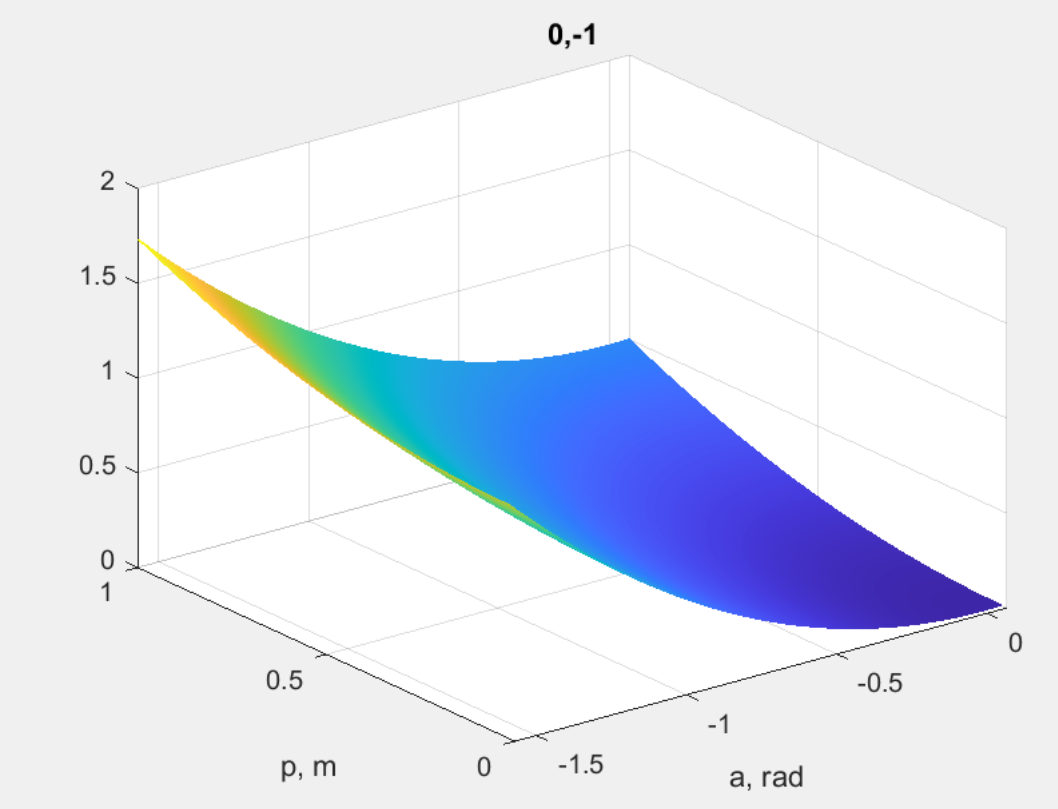


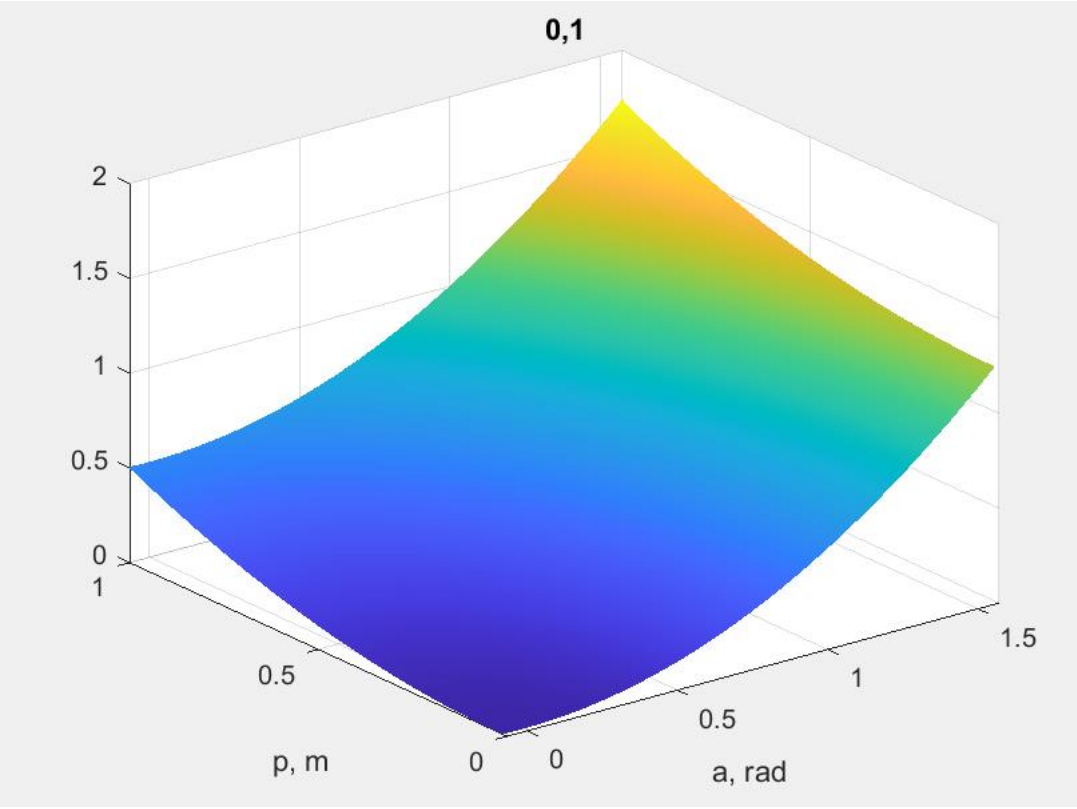
Графики для функции Ляпунова:

Линейный:

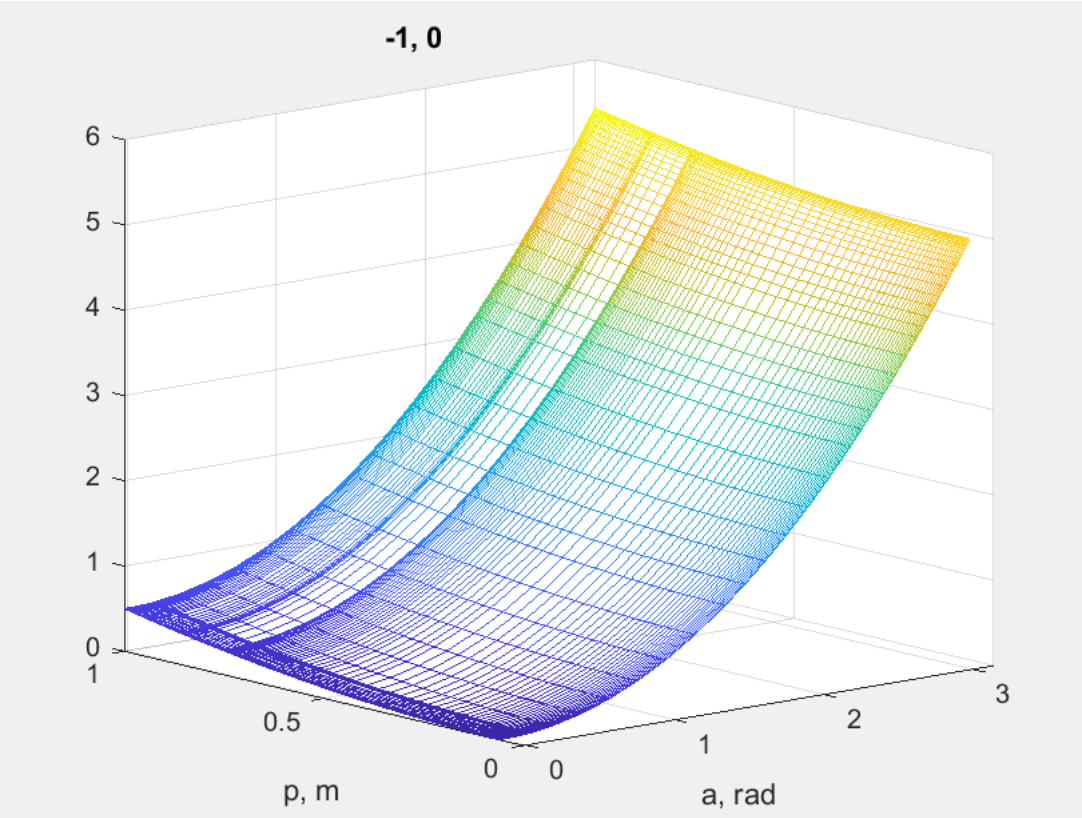

1,0
0.5
0.4
0.3
0.2
0.1
0
1
0.5
0
p, m
-1
-0.5
0
0.5
1
a, rad

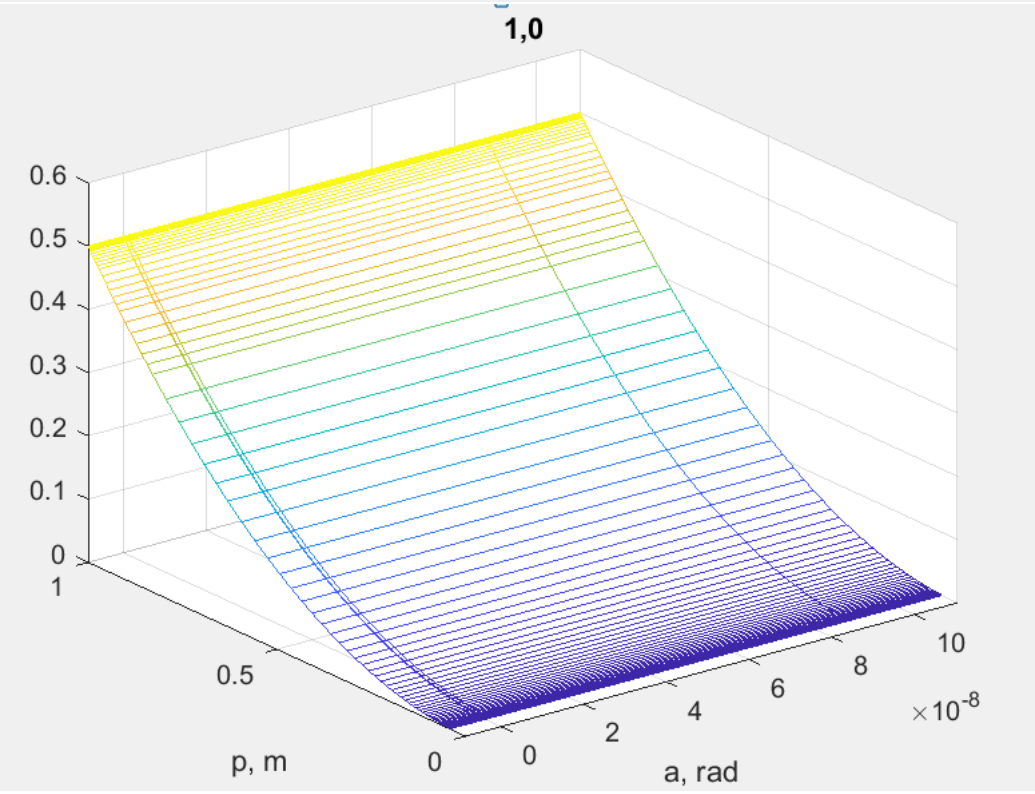

[-1,0]
V, m2 + rad2
6
5
4
3
2
1
0
4
2
0
-2
-4
a, rad
0
0.2
0.4
0.6
0.8
1
p, m

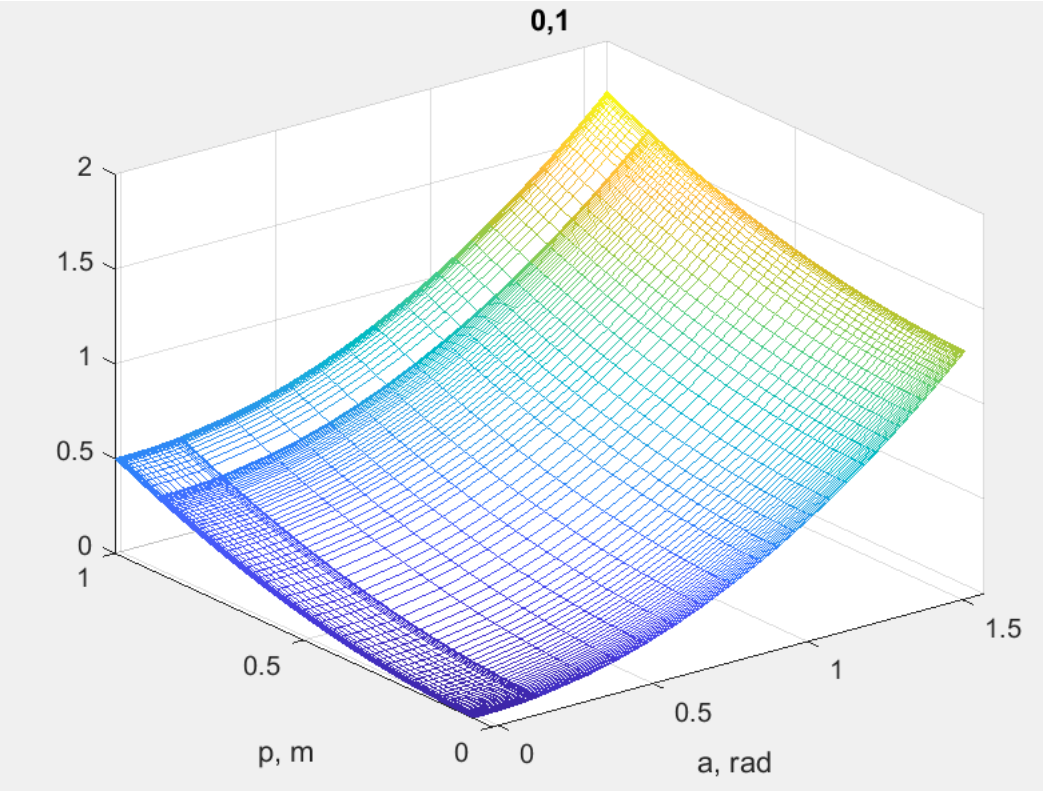

0,-1
2
1.5
1
0.5
0
1
0.5
p, m
0
-1.5
-1
-0.5
0
a, rad


0,1
2
1.5
1
0.5
0
1
0.5
p, m
0
0
0.5
1
1.5
a, rad

Нелинейный:


-1, 0
6
5
4
3
2
1
0
1
0.5
0
p, m
0
1
2
3
a, rad


1,0
0.6
0.5
0.4
0.3
0.2
0.1
0
1
0.5
0
p, m
0
2
4
6
8
10
×10-8
a, rad


0,1
2
1.5
1
0.5
0
1
0.5
0
p, m
0
0.5
1
1.5
a, rad

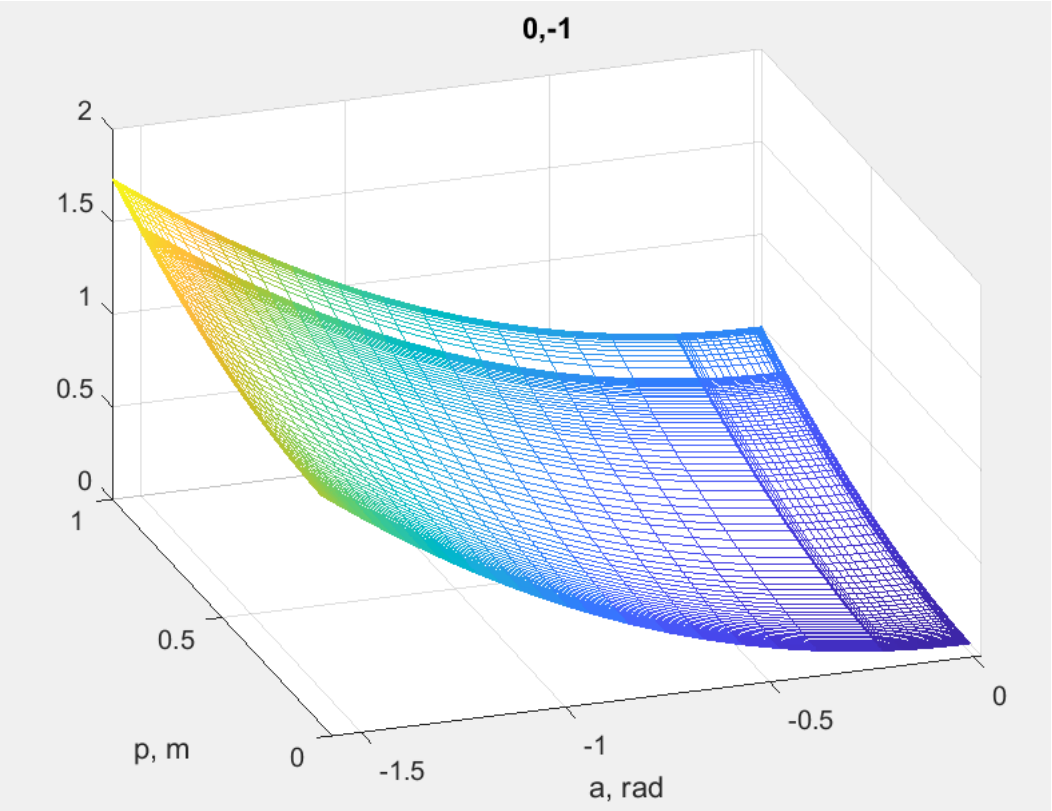
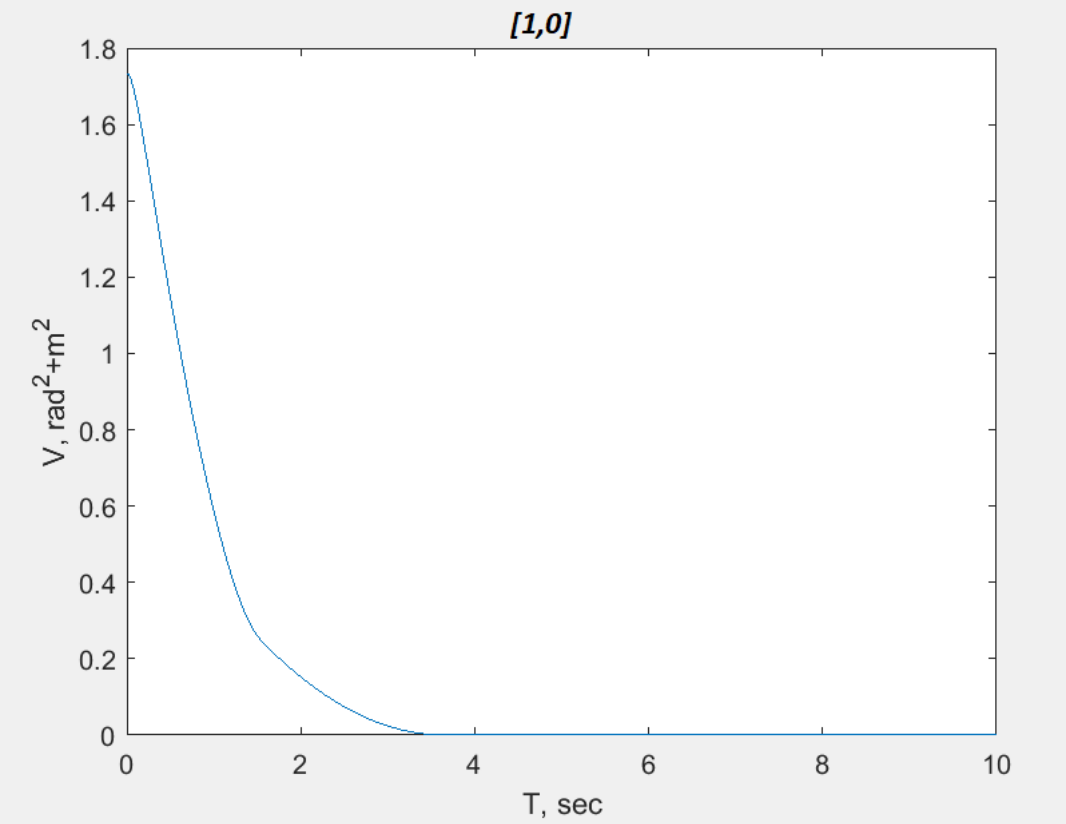
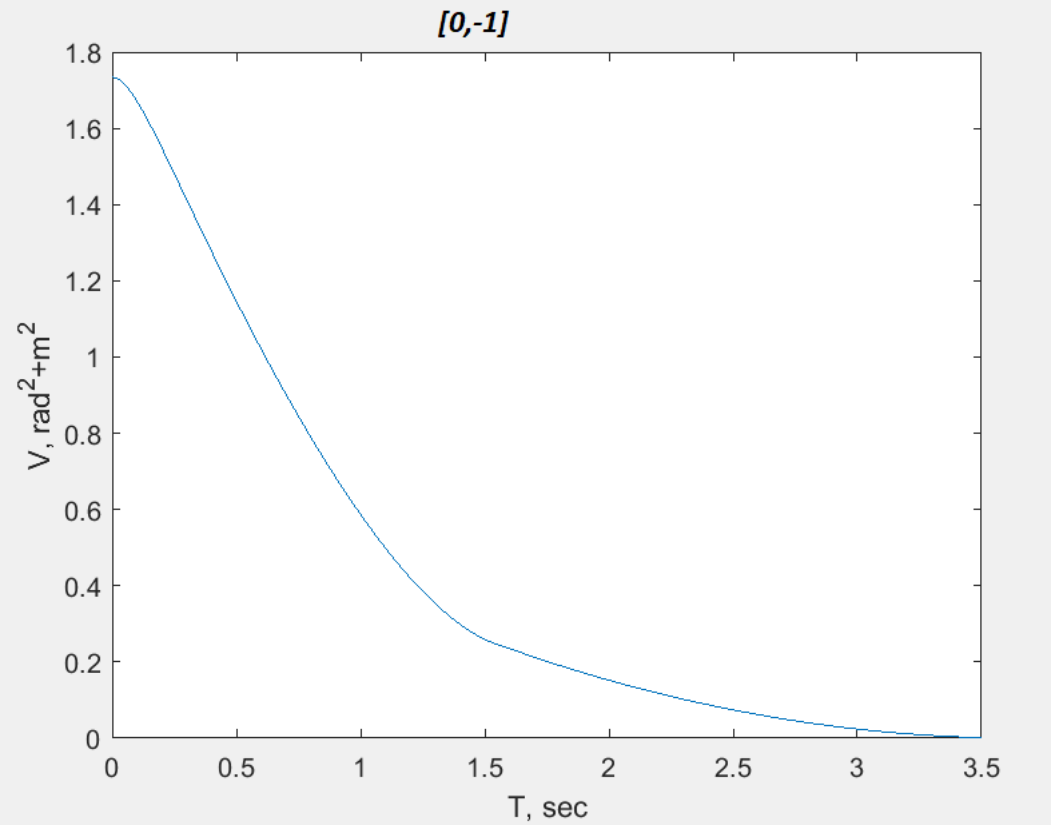
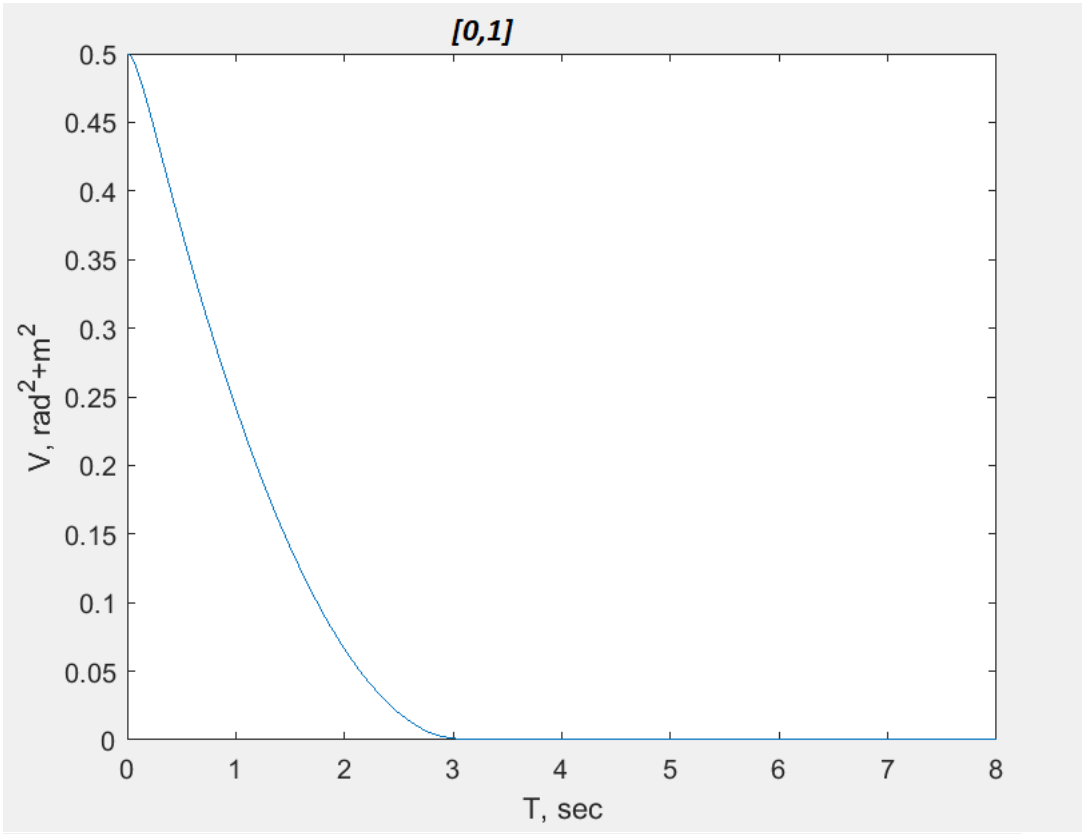
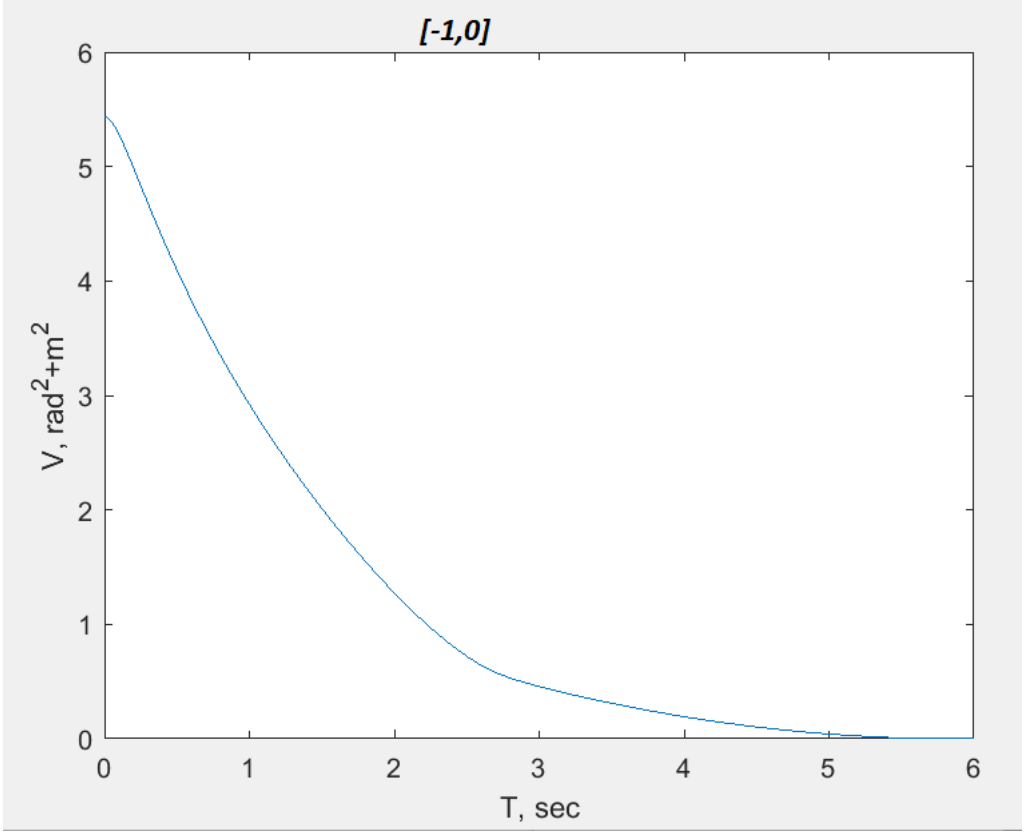

0,-1
2
1.5
1
0.5
0
1
0.5
0
p, m
-1.5
-1
-0.5
0
a, rad

График зависимости V(t):
Нелинейный:

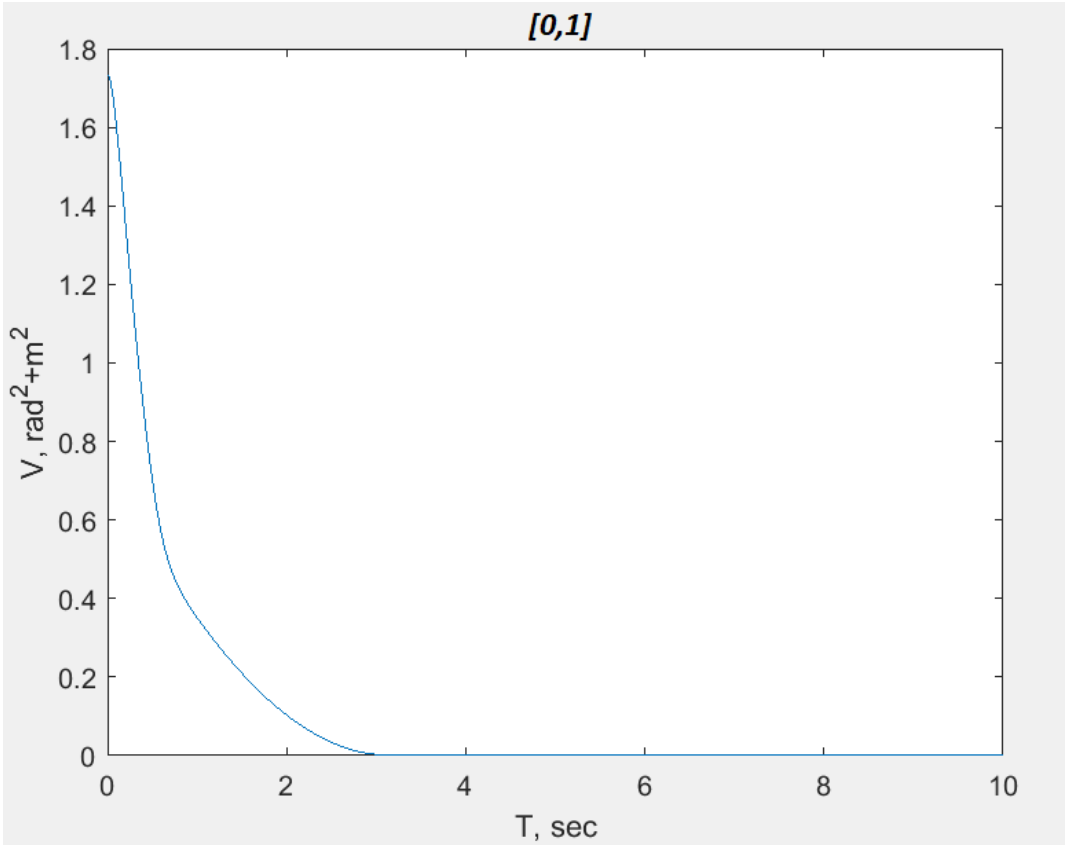


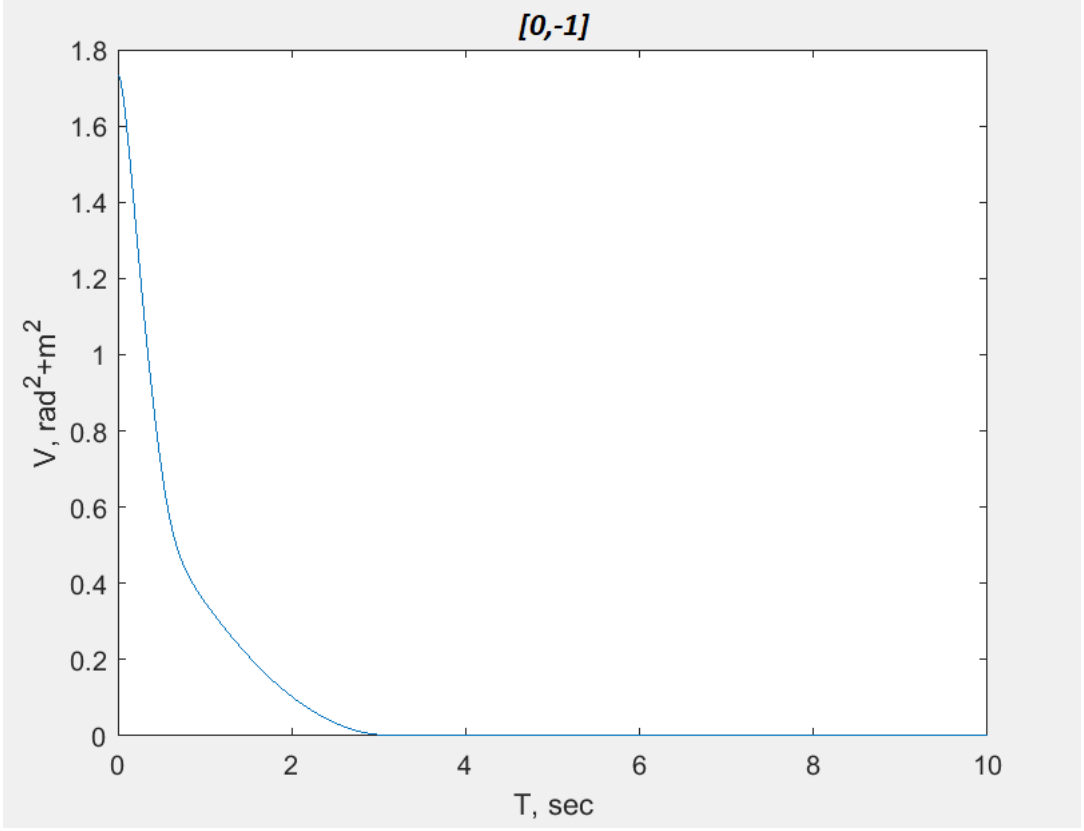


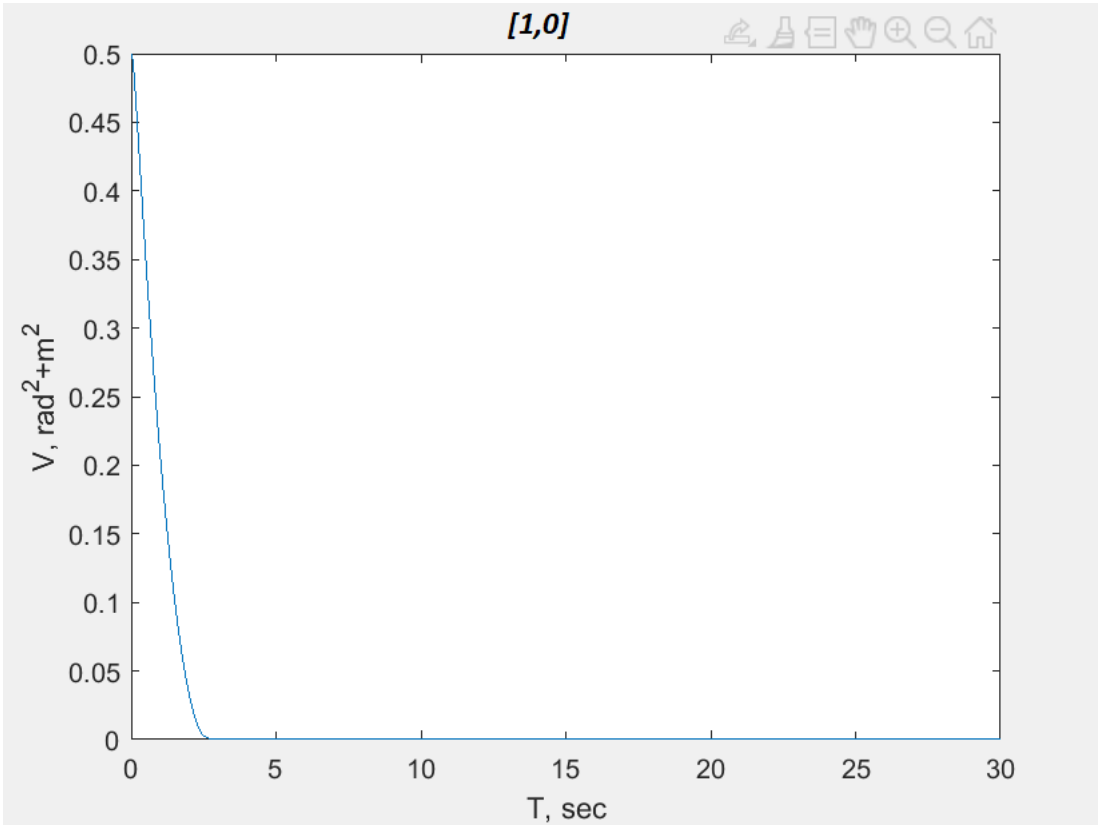

[0,1]
V, rad2+m2
T, sec
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0
1
2
3
4
5
6
7
8

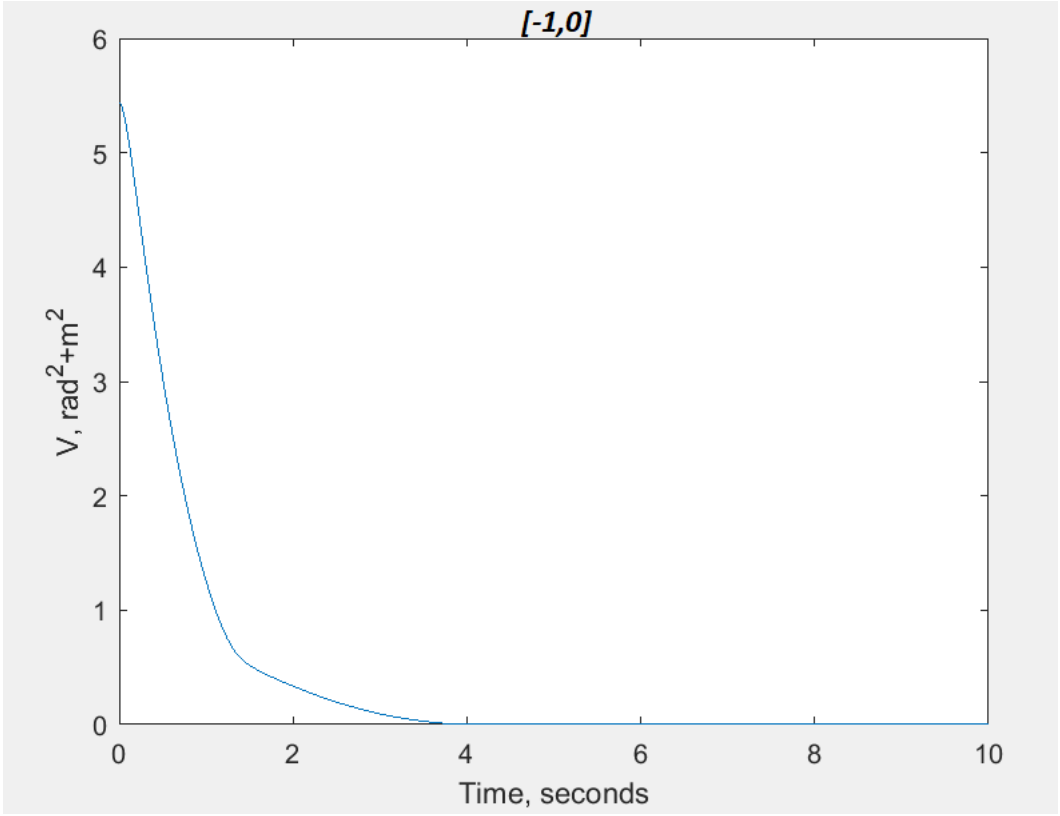

[-1,0]
V, rad2+m2
T, sec
0
1
2
3
4
5
6
0
1
2
3
4
5
6

Линейный:


[0,1]
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
V, rad²+m²
0
2
4
6
8
10
T, sec


[0,-1]
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
V, rad²+m²
0
2
4
6
8
10
T, sec


[1,0]
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
V, rad2+m2
0
5
10
15
20
25
30
T, sec


[-1,0]
6
5
4
3
2
1
0
V, rad2+m2
0
2
4
6
8
10
Time, seconds

Код для линейного закона:

```
#!/usr/bin/env python3
from ev3dev.ev3 import *
from math import cos, sin, pi, atan2
from time import time

def signum(a):

    if a > 0:
        return 1
    elif a < 0:
        return -1
    else:
        return 0

def run_square(cordtop, length, cordrob, tetta):

    cordrob, tetta = run_points(cordtop, cordrob, tetta)
    cordtop[0] -= length
    cordrob, tetta = run_points(cordtop, cordrob, tetta)
    cordtop[1] -= length
    cordrob, tetta = run_points(cordtop, cordrob, tetta)
    cordtop[0] += length
    cordrob, tetta = run_points(cordtop, cordrob, tetta)
    cordtop[1] += length
    cordrob, tetta  = run_points(cordtop, cordrob, tetta)

    return cordrob, tetta



def run_points(goal, cordrob, tetta):

    motorR = LargeMotor('outA')
    motorL = LargeMotor('outB')

    # test №1: goal[3,0]
    # test №2: goal[0,3]
    # test №3: goal[3,3]
    # test №4: goal[-3,0]
    # test №5: goal[0,-3]
    # test №6: goal[-3,3]
    # test №7: goal[3,-3]
    # test №8: goal[-3,-3]
    # Если прога проходит все тесты, то в записи в файл удалить вывод угла и взят
ь измерения

    r = 0.03 # радиус колеса в метрах
    b = 0.155 # рассотяние между центрами колес
```

```
    motorR.position = 0
    motorL.position = 0

    angleR, angleL = 0,0



    Umax = 7 # макимальное напряжение, может быть другим

    data = open(str("(" + goal[0]) + str(goal[1]) +") goal.txt", "w+") # название
 менять на номер теста

    Ks = 15 # коэффициент для движения вперед, возможно побольше, но точно меньше
, чем для поворота
    Kr = 25 # коэффициент для поворота, возможно побольше стоит сделать, но не си
льно много


    if (goal[0] != cordrob[0]):
            asimut = atan2((goal[1]-cordrob[1]), (goal[0]-
cordrob[0])) # считается угол между целью и осью Ox
        elif (goal[1] > 0):
            asimut = pi/2
        else:
            asimut = -pi/2

    p = ((goal[0] - cordrob[0])**2 + (goal[1] - cordrob[1])**2)**0.5 # начальное
расстояние
    prevp = p
    xprev = cordrob[0]
    yprev = cordrob[1]

    tettaprev = tetta
    a = asimut - tetta
    preva = a

    prevangleR = 0
    prevangleL = 0

    integralp = 0
    derivativep = 0
    ip = 5
    dp = 20

    integrala = 0
    derivativea = 0
    ia = 5
    da = 20

    previousTime = time()
```

```
    while (p > 0.2):

        currentTime = time()
        dt = currentTime - previousTime
        previousTime = currentTime

        angleR = motorR.position * pi/180
        angleL = motorL.position * pi/180

        difangleR = angleR - prevangleR
        difangleL = angleL - prevangleL

        prevangleR = angleR
        prevangleL = angleL

        srangle = (difangleR+difangleL)/2

        tetta = tettaprev + (difangleR - difangleL)*r/b # считается угол на котор
ый повернул робот/ возможно, нужно раньше считать, чем координаты

        tettaprev = tetta

        cordrob[0] = xprev + cos(tetta)*srangle*r # считается x
        cordrob[1] =  yprev + sin(tetta)*srangle*r # считается y

        xprev = cordrob[0]
        yprev = cordrob[1]

        p = ((goal[0] - cordrob[0])**2 + (goal[1] - cordrob[1])**2)**0.5 # считае
тся расстояние от робота до цели

        integralp += p*dt

        if integralp > 1:
            integralp = 1

        derivativep = (p - prevp)/dt
        prevp = p

        a = asimut - tetta # считается угол между желаемым положением робота и те
кущим положением
        integrala += a*dt
        derivativea = (a-preva)/dt
        preva = a
        if (abs(a) > pi):
            a = a - signum(a) * 2 * pi # для кратчайшего угла

        Usinvolt = Ks * p + dp*derivativep + ip*integralp # расчет напряжения для
 движения вперед с помощью пропорционального регулятора
        Urinvolt= Kr * a + da*derivativea + ia*integrala # расчет напряжения для
поворота с помощью П регултора
```

```
        Usinpercent = Usinvolt*100/Umax
        Urinpercent = Urinvolt*100/Umax

        if abs(Usinpercent) >= 70:
            Us = 70*signum(Usinpercent)
        if abs(Ur) >= 30:
            Ur = 30*signum(Urinpercent)

        # Блок для правого колеса

        UR = Us + Ur
        UL = Us - Ur

        # далее подаем напряжение

        motorR.run_direct(duty_cycle_sp = (UR))
        motorL.run_direct(duty_cycle_sp = (UL))

        # запись угла, на который надо повернуться и координат, здесь слежует уда
лить a, когда пройдет все тесты
        data.write(str(cordrob[0]) +  '\t' + str(cordrob[1]) + '\n')

    data.close()
    motorR.stop(stop_action = 'brake')
    motorL.stop(stop_action = 'brake')

    return cordrob, tetta
a,b = run_square([1, 1], 2, [0, 0], 0)
```

Код для нелинейного закона:

```
#!/usr/bin/env python3
from ev3dev.ev3 import *
from math import cos, sin, pi, atan2, tanh
from time import time

def signum(a):

    if a > 0:
        return 1
    elif a < 0:
        return -1
    else:
        return 0

def run_square(cordtop, length, cordrob, tetta):

    cordrob, tetta = run_points(cordtop, cordrob, tetta)
    cordtop[0] -= length
    cordrob, tetta = run_points(cordtop, cordrob, tetta)
```

```
    cordtop[1] -= length
    cordrob, tetta = run_points(cordtop, cordrob, tetta)
    cordtop[0] += length
    cordrob, tetta = run_points(cordtop, cordrob, tetta)
    cordtop[1] += length
    cordrob, tetta  = run_points(cordtop, cordrob, tetta)
    return cordrob, tetta


def run_points(goal, cordrob, tetta):

    motorR = LargeMotor('outA')
    motorL = LargeMotor('outB')

    # sensor = sensor.lego.UltrasonicSensor(address=None, name_pattern='sensor*',
 name_exact=False, **kwargs)

    # test №1: goal[3,0]
    # test №2: goal[0,3]
    # test №3: goal[3,3]
    # test №4: goal[-3,0]
    # test №5: goal[0,-3]
    # test №6: goal[-3,3]
    # test №7: goal[3,-3]
    # test №8: goal[-3,-3]
    # Если прога проходит все тесты, то в записи в файл удалить вывод угла и взят
ь измерения

    r = 0.028 # радиус колеса в метрах
    b = 0.155 # рассотяние между центрами колес

    motorR.position = 0
    motorL.position = 0

    angleR, angleL = 0,0



    Umax = 7 # макимальное напряжение, может быть другим

    data = open("goal(" + str(goal[0]) + ", " + str(goal[1] + ")"), "w+")

    if (goal[1]!=0 and goal[0]!=0):
        asimut = atan2((goal[1]-cordrob[1]), (goal[0]-cordrob[0]))

    if (goal[0] == 0) and (goal[1] > 0):
        asimut = pi/2
    elif (goal[0] == 0) and (goal[1] < 0):
        asimut = -pi/2

    if (goal[1] == 0) and (goal[0] > 0 ):
```

```
        asimut = 0
    elif (goal[1] == 0) and (goal[0] < 0):
        asimut = pi


    p = ((goal[0] - cordrob[0])**2 + (goal[1] - cordrob[1])**2)**0.5 # начальное
расстояние

    intp = 0 # интеграл оставшейся длины
    K_I = 0.1
    baseSpeed, control = 0, 0 # линейная и угловая скорость

    xprev = cordrob[0]
    yprev = cordrob[1]

    tettaprev = tetta

    prevangleR = 0
    prevangleL = 0

    k_dist = 20

    k_w = 20
    previousTime = time()

    while (p > 0.2):

        currentTime = time()
        dt = currentTime - previousTime
        previousTime = currentTime


        angleR = motorR.position * pi/180
        angleL = motorL.position * pi/180

        prevangleR = angleR
        prevangleL = angleL

        difangleR = angleR - prevangleR
        difangleL = angleL - prevangleL

        srangle = (difangleR+difangleL)/2

        tetta = tettaprev + (difangleR - difangleL)*r/b # считается угол на котор
ый повернул робот/ возможно, нужно раньше считать, чем координаты

        tettaprev = tetta

        cordrob[0] = xprev + cos(tetta)*srangle*r # считается x
        cordrob[1]=  yprev + sin(tetta)*srangle*r # считается y
```

```
        xprev = cordrob[0]
        yprev = cordrob[1]

        p = ((goal[0] - cordrob[0])**2 + (goal[1] - cordrob[1])**2)**0.5 # считае
тся расстояние от робота до цели

        intLength += length * dt # здесь стоит подумать над коэффицинтами

        prevp = p


        a = asimut - tetta # считается угол между желаемым положением робота и те
кущим положением


        if (abs(a) > pi):
            a = a - signum(a) * 2 * pi # для кратчайшего угла

        if (intp > 10):
            intp = 10


        baseSpeedinvolt = Umax * tanh (p) * cos (a) + K_I * intp
        baseSpeedinpercent = baseSpeedinvolt*100/Umax
        if (abs(baseSpeedinpercent) > 70):
            baseSpeedinpercent = signum(baseSpeedinpercent) * 70;

        controlinvolt = k_w * a + sin(a) * baseSpeedinvolt / p
        controlinpercent = controlinvolt*100/Umax
        if (abs(controlinpercent) > 30):
            controlinpercent = signum(controlinpercent)*30;

        # Блок для правого колеса

        UR = int(baseSpeed + control)

        # блок для левого колеса

        UL = int(baseSpeed - control)

        # далее подаем напряжение

        motorR.run_direct(duty_cycle_sp = (UR))
        motorL.run_direct(duty_cycle_sp = (UL))

        # запись угла, на который надо повернуться и координат, здесь слежует уда
лить a, когда пройдет все тесты
        data.write(str(cordrob[0]) +  '\t' + str(cordrob[1]) + '\n')

    data.close()
    motorR.stop(stop_action = 'brake')
```

```
    motorL.stop(stop_action = 'brake')

    return cordrob, tetta

run_square([1,1], 2, [0, 0], 0)
```

Код для объезда препятствий:

```
#!/usr/bin/env python3
from ev3dev.ev3 import *
from math import atan2, sqrt, pi, cos, sin, tanh
import time

wNls = 14.64 # max angular speed reached by the motor
r = 0.028 # wheel radius
vMax = wNls * r # max linear speed of the our robot
wheelDist = 0.151 # distance between wheels
wMax = 2 * r * wNls / wheelDist # max angular speed of our robot
uMax = 7 # max voltage

# from the equation: |wMax| = (kW ** distanceToGoal) * (pi / 4) + 0.5 * vMax
# we can find coefficient kW

kW = 10 # coefficient for calculating the required angular velocity (any > 0)
kpS = 15
kpR = 3

kiS = 0 # integral gain for linear speed
kdS = 0 # differential gain for linear speed

kiR = 1 # integral gain for angular speed
kdR = 0 # differential dain for angular speed

motorR = LargeMotor('outB') # right motor declaration
motorL = LargeMotor('outA') # left motor declaration

sensorR = UltrasonicSensor("in2") # right us sensor declaration
sensorL = UltrasonicSensor("in1") # left us sensor declaration

ts = TouchSensor("in4")

minDist = 0.15 # min distance to avoid obstacle
kObs = 50 # coefficient to form an obstacle avoidance angle

def sign(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0
```

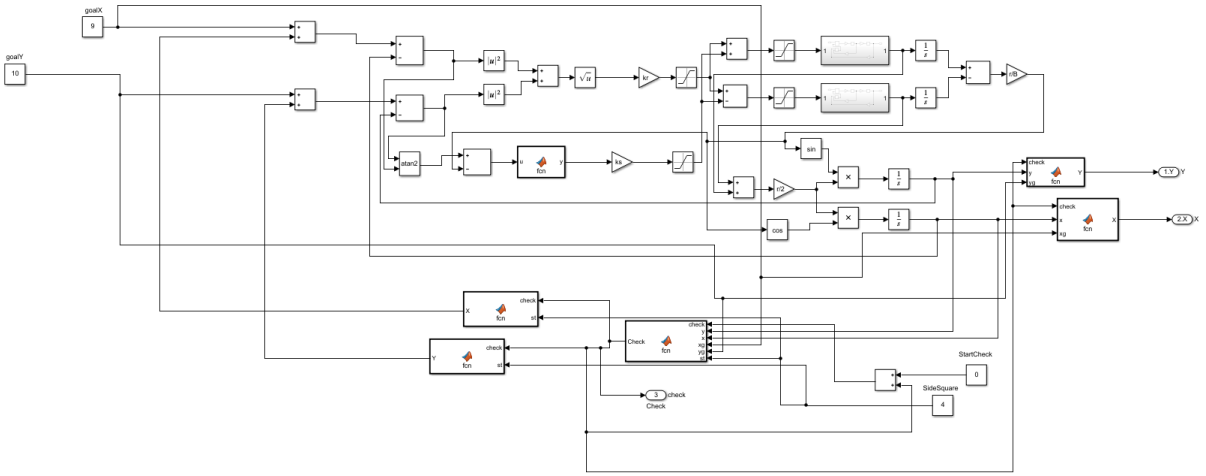
```
def run(curPosition, goal, tetta):

    # create file for output
    dataOutput = open("5LabPID_ObstaclesMeasurements [" + str(round(curPosition[0
])) + " " + str(round(curPosition[1])) + "] -
> [" + str(goal[0]) + " " + str(goal[1]) + "].txt","w+")

    motorR.position = 0 # in grad
    motorL.position = 0 # in grad
    angleR = 0 # angle of rotation of the right motor in radians
    angleL = 0 # angle of rotation of the left motor in radians

    prevPosition = [curPosition[0], curPosition[1]]

    distanceToGoal = sqrt(((goal[0] - curPosition[0])**2) + ((goal[1] - curPositi
on[1])**2))

    azimut = atan2(goal[1] - curPosition[1], goal[0] - curPosition[0]) # in radia
ns
    headingAngle = azimut - tetta

    intSumDistanceToGoal = 0 # integral sum for distance to goal
    intSumHeadingAngle = 0 # integral sum for heading angle

    currentTime = time.time()
    prevTime = time.time()
    startTime = time.time()

    while distanceToGoal > 0.1:

        # calculating distance to obstacle error
        distLeftObstacle = minDist - sensorL.value() / 1000
        distRightObstacle = minDist - sensorR.value() / 1000

        if distLeftObstacle < 0:
            distLeftObstacle = 0
        if distRightObstacle < 0:
            distRightObstacle = 0
        if distRightObstacle > distLeftObstacle:
            distClosestObstacle = distRightObstacle
        else:
            distClosestObstacle = distLeftObstacle

        # obstacleSide > 0, if nearest obstacle to the right
        # obstacleSide < 0, if nearest obstacle to the left
        # obstacleSide = 0, if there are no obstacles or at an equal distance
        obstacleSide = sign(distRightObstacle - distLeftObstacle)

        prevDistanceToGoal = distanceToGoal
        prevHeadingAngle = headingAngle
```

```
        prevTime = currentTime
        currentTime = time.time()
        dt = currentTime - prevTime

        tettaPrev = tetta

        angleRPrev = angleR
        angleLPrev = angleL
        angleR = motorR.position * pi / 180
        angleL = motorL.position * pi / 180
        difAngleR = angleR - angleRPrev
        difAngleL = angleL - angleLPrev

        prevPosition[0] = curPosition[0]
        prevPosition[1] = curPosition[1]

        tetta = tettaPrev + (difAngleR - difAngleL) * r / wheelDist # in radians

        curPosition[0] = prevPosition[0] + cos(tetta) * (difAngleR + difAngleL) /
 2 * r
        curPosition[1] = prevPosition[1] + sin(tetta) * (difAngleR + difAngleL) /
 2 * r

        distanceToGoal = sqrt(((goal[0] - curPosition[0])**2) + ((goal[1] - curPo
sition[1])**2))

        azimut = atan2(goal[1] - curPosition[1], goal[0] - curPosition[0]) # in r
adians
        headingAngle = azimut - tetta + kObs * distClosestObstacle * obstacleSide
 # in radians
        # print(kObs * distClosestObstacle * obstacleSide, distClosestObstacle)

        # shortest corner turn
        if abs(headingAngle) > pi:
            headingAngle = headingAngle - sign(headingAngle) * 2 * pi

        intSumHeadingAngle += headingAngle * dt
        intSumDistanceToGoal += distanceToGoal * dt

        # linear velocity
        linVel = vMax * tanh(distanceToGoal) * cos(headingAngle)

        # coefficient for angular velocity
        kW = ((abs(wMax) - 0.5 * vMax) / (pi / 4)) ** (1 / distanceToGoal)
        # angular velocity
        angVel = kW * headingAngle + vMax * (tanh(distanceToGoal) / distanceToGoa
l) * sin(headingAngle) * cos(headingAngle)

        # uSI - integral component of voltage for angular speed
        uSI = kiS * intSumDistanceToGoal
        # anti-windup for integral component (limited to 15% of maximum power)
```

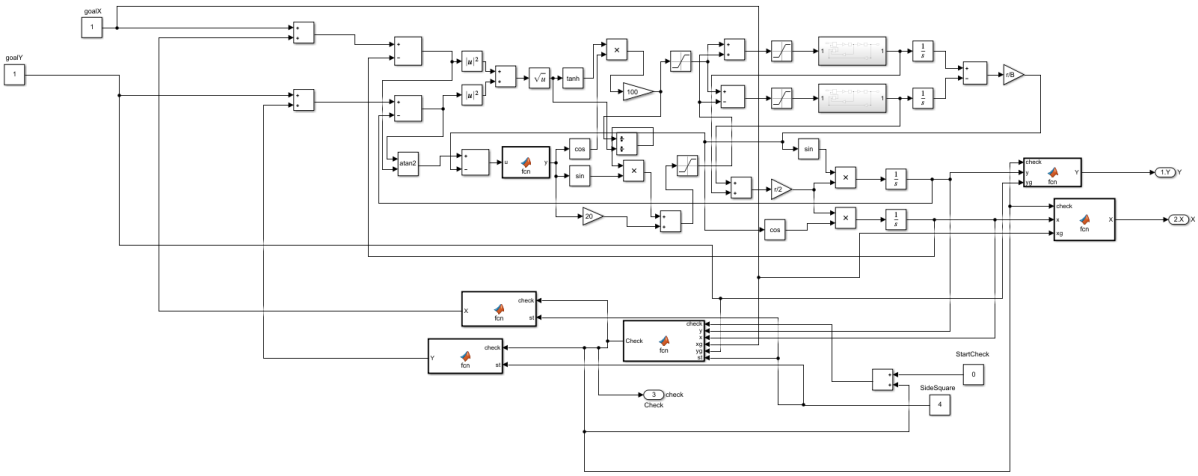
```
        if abs(uSI / uMax * 100) > 15:
            uSI = sign(uSI) * uMax * 0.15

        # voltage for linear speed and its limitation
        uS = kpS * linVel + uSI + kdS * (distanceToGoal - prevDistanceToGoal) / d
t
        if abs(uS / uMax * 100) > 70:
            uS = sign(uS) * uMax * 0.7

        # uRI - integral component of voltage for angular speed
        uRI = kiR * intSumHeadingAngle
        # anti-windup for integral component (limited to 15% of maximum power)
        if abs(uRI / uMax * 100) > 15:
            uRI = sign(uRI) * uMax * 0.15

        # voltage for angular speed and its limitation
        uR = kpR * angVel + uRI + kdR * (headingAngle - prevHeadingAngle) / dt
        if abs(uR / uMax * 100) > 30:
            uR = sign(uR) * uMax * 0.3

        # conversion of the applied voltage to the right and left motors in perce
nt and its limitation
        if abs(uS + uR) > uMax:
            voltageR = sign(uS + uR) * 100
        else:
            voltageR = (uS + uR) / uMax * 100

        if abs(uS - uR) > uMax:
            voltageL = abs(uS - uR) * 100
        else:
            voltageL = (uS - uR) / uMax * 100

        # print(uS, uR)
        print(sensorL.value() / 10, sensorR.value() / 10)
        # print(headingAngle)
        # power supply to the motors
        motorR.run_direct(duty_cycle_sp = (voltageR))
        motorL.run_direct(duty_cycle_sp = (voltageL))

        # output positions
        dataOutput.write("x = " + str(curPosition[0]) + '\t' + "y = " + str(curPo
sition[1]) + '\t' + "t = " + str(currentTime - startTime) + '\n')

        if ts.is_pressed:
            motorR.run_direct(duty_cycle_sp = 0)
            motorL.run_direct(duty_cycle_sp = 0)
            break
    dataOutput.close()

    motorR.stop(stop_action = 'brake')
    motorL.stop(stop_action = 'brake')
```

```
    return curPosition, tetta

curPosition, tetta = run([0, 0], [3, 0], 0)
```

Модель для линейного закона:



Модель для нелинейного закона:



3. Выводы: на практике мы управляли роботом с дифференциальным приводом с помощью двух законов управления: линейного и нелинейного. В основе линейного закона лежит Пи регулятор (управляющее воздействие складывается из двух ошибок, по углу и расстоянию, умноженных на коэффициент, и также интегральной составляющей этих величин). В основе нелинейного закона лежит функция Ляпунова (управляющее воздействие выбирается так, чтобы система была устойчива по Ляпунову (производная от функции должна быть отрицательна определена в окрестности точки равновесия), то есть ошибка по углу и расстоянию стремились к нулю). На практике мы столкнулись с настройкой Пи регулятора, что являлось сложной задачей, так как теоретически предсказать значение коэффициентов мы не в состоянии, а подбирать 4 коэффициента довольно сложно. Поэтому нам показалось,

что нелинейный закон проще в использовании, но для такого подхода нужно правильно подобрать управляющее воздействие, что не всегда просто. На практике был получен положительный результат в обоих случаях, различия в движении робота заключались в том, что при нелинейном законе он поворачивал, немного проезжая назад, и ехал по оптимальному пути (на видео). С относительной погрешностью 5% по расстоянию, существенных различий между линейным и нелинейным законом замечено не было, поэтому можно считать эти законы эквивалентными, если важен сам результат. Также мы реализовали объезд препятствий. Но при нашем управлении робот, когда “видит” препятствие начинает отклоняться от заданной траектории, из-за чего перестает “видеть” его и снова на него поворачивается. Принципиальное различие также заключается в том, что при нелинейном законе функция Ляпунова всегда убывает. А при линейном функция сначала может расти за счет того, что роботу надо повернуться и он едет прямо.