Старшая категория

Участникам было предоставлено одинаковое оборудование (наборы «ТРИК. Учебная пара») и одинаковые мобильные платформы на mecanum-колесах. Но так как перед проведением финала проводились трехдневные учебно-тренировочные сборы, то к началу финала команды располагали значительно отличающимися версиями роботов, с различиями в установке датчиков, камер, схватов и прочей периферии. Вследствие этого единое эталонное решение, гарантировано и одинаково исполняемое на роботах всех команд, подготовить невозможно. Поэтому предлагается декомпозировать задачу на составляющие и в дальнейшем описать решение ключевых ее элементов.

Финальная задача предполагала подготовку командами управляющих алгоритмов, реализующих следующие операции:

5. перемещение робота с платформой на mecanum-колесах по заданным векторам, дугам, линиям;
6. поиск и распознавание ArTag-меток;
7. захват объектов схватом и выгрузка их в определенных зонах.

Рассмотрим решение этих операций подробнее.

Перемещение mecanum-платформы.

В процессе выполнения задания робот команды мог перемещаться по заранее известному маршруту. Расположение зон установок ArTag-меток, зон погрузки и выгрузки, расположение препятствий были известны заранее. Так, движение от зоны старта/финиша к любой из зон установки ArTag-меток могло быть выполнено как движение по прямой на заданное расстояние. При этом установка робота вдоль левого борта поля упрощала управление мобильной платформой, требуя запуска всех моторов с одинаковой скоростью. Отслеживание пройденного роботом расстояния упрощалось, так как оно вычислялось как расстояние, пройденное любым колесом (или среднее арифметическое расстояний, пройденных всеми четырьмя колесами робота), и считывалось с энкодера.

Дальнейшее движение робота между различными зонами полигона предполагало включение моторов с разными скоростями. Пример кода одной из команд для движения по вектору под любым углом представлен ниже (язык программирования JavaScript):

```
1. function moveDeg(v, alpha) {
2.       alpha = alpha/180*Math.PI;
3.       setAllMotor(v*Math.cos(alpha-Math.PI/4),
4.                   v*Math.sin(alpha-Math.PI/4),
5.                   v*Math.sin(alpha-Math.PI/4),
6.                   v*Math.cos(alpha-Math.PI/4));
7. }
```

Функция принимает два входных параметра: скорость движения робота и угол перемещения. Так как передавать угол в функцию удобнее в градусах, а контроллер

работает вычисляет тригонометрические операции в радианах, то в строке 2 производится перевод угла в радианы. В строках 3-6 вызывается команда запуска всех моторов и вычисление скоростей для них. В зависимости от порядка нумерации моторов и колес функция может незначительно отличаться.

Индивидуальные особенности исполнения моторов, колес, неравномерность развесовки платформы и трения с поверхностью могли вызывать отклонения в движения робота. Для их минимизации участники могли реализовать синхронизацию моторов и отслеживание направления робота по IMU-сенсору (гироскопу). Способы синхронизации моторов описаны в учебном пособии «Управление моторами тележки с контроллером Трик на JavaScript»; для платформы с четырьмя колесами рациональнее было использовать алгоритм синхронизации «по виртуальному колесу».

Пример кода одной из команд для отслеживания направления робота по IMU-сенсору:

```
1.  def _drive(power_L1, power_R1, power_L2, power_R2, angle, error_sign):
2.      global prev_error
3.      kp = 3
4.      kd = 8
5.      error = Gyroscope.direction - angle
6.      error *= error_sign
7.      up = error * kp + (error - prev_error) * kd
8.      prev_error = error
9.      Motor.set_power('L1', (abs(power_L1) + up) * get_sign(power_L1))
10.     Motor.set_power('R1', (abs(power_R1) + up) * get_sign(power_R1))
11.     Motor.set_power('L2', (abs(power_L2) - up) * get_sign(power_L2))
12.     Motor.set_power('R2', (abs(power_R2) - up) * get_sign(power_R2))
```

В эту функцию передаются заранее вычисленные скорости моторов и угол, вдоль которого необходимо перемещаться роботу. Последним параметром передается «знак ошибки», который зависит от направления движения робота: положительный для любого движения влево и отрицательный для движения вправо.

Стоит отметить, что некоторым командам оказалось достаточно 8 базовых движений, прописанных в коде: вперед, назад, влево, вправо и по четырем диагоналям. Они прописали для своих роботов комбинации из этих движений с выравниванием по гироскопу, линиям или вдоль стен. Пример кода одной из команд для движения на заданное расстояние по одному из восьми базовых направлений (язык программирования Python):

```
1. def all_move(dist, sp, type):
2.     if type == 'F':
3.       brick.encoder("E4").reset()
4.       script.wait(10)
5.       move(sp, sp, sp, sp)
6.       while abs(brick.encoder("E4").read()) < dist:
7.            script.wait(1)
8.       move(0, 0, 0, 0)
9.       script.wait(100)
10.         elif type == 'B':
11.           brick.encoder("E4").reset()
12.           script.wait(10)
13.           move(-sp, -sp, -sp, -sp)
```

```
14.           while abs(brick.encoder("E4").read()) < dist:
15.                script.wait(1)
16.           move(0, 0, 0, 0)
17.           script.wait(100)
18.         elif type == 'DFR':
19.           brick.encoder("E2").reset()
20.           script.wait(10)
21.           move(0, sp, sp, 0)
22.           while abs(brick.encoder("E2").read()) < dist:
23.                script.wait(1)
24.           move(0, 0, 0, 0)
25.           script.wait(100)
26.         elif type == 'DBR':
27.           brick.encoder("E4").reset()
28.           script.wait(10)
29.           move(-sp, 0, 0, -sp)
30.           while abs(brick.encoder("E4").read()) < dist:
31.                script.wait(1)
32.           move(0, 0, 0, 0)
33.           script.wait(100)
34.         elif type == 'DFL':
35.           brick.encoder("E4").reset()
36.           script.wait(10)
37.           move(sp, 0, 0, sp)
38.           while abs(brick.encoder("E4").read()) < dist:
39.                script.wait(1)
40.           move(0, 0, 0, 0)
41.           script.wait(100)
42.         elif type == 'DBL':
43.           brick.encoder("E2").reset()
44.           script.wait(10)
45.           move(0, -sp, -sp, 0)
46.           while abs(brick.encoder("E2").read()) < dist:
47.                script.wait(1)
48.           move(0, 0, 0, 0)
49.           script.wait(100)
```

В зонах погрузки и выгрузки была нанесена разметка, облегчающая перемещение и выравнивание роботов. Кроме того, роботы участников могли двигаться вдоль стен. Рассмотрим несколько примеров из кода участников, реализующих перемещение робота по разметке и вдоль стен (для всех примеров язык программирования Python).

Движение робота до черной линии на белом фоне (перемещение к зоне погрузки):

```
1.    def go_to_line(spr):
2.       s3 = brick.sensor("A1").read
3.       brick.sensor("A1").readRawData()
4.       while s3() > 8:
5.          s3 = brick.sensor("A1").read
6.          move(spr, spr, spr, spr)
7.       move(0, 0, 0, 0)
```

Функция используется для линейного движения к линии с остановкой при показаниях датчика ниже заданного порога (до обнаружения темного фона; порог подбирается экспериментально и зависит как от индивидуальных особенностей датчика, так и от способа его установки). В этом примере все колеса запускаются с одинаковой скоростью, то есть направление движения робота совпадает с направлением «вперед». Но вызывая в строке 6 функции из предыдущих примеров можно добиться движения в одном из восьми базовых направлений или движения по вектору с любым заданным углом.

Движение по линии на заданное расстояние в сантиметрах:

```
1.  #движение по линии
2.  def go(sp):
3.    global err_old
4.    global k
5.    global kd
6.    s1 = brick.sensor("A6").read
7.    s2 = brick.sensor("A5").read
8.    brick.sensor("A6").readRawData()
9.    brick.sensor("A5").readRawData()
10.   err = s1() - s2()
11.   u = (err - err_old) * kd + k * err
12.   err_old = err
13.   move(sp + u,sp - u, sp + u, sp - u)
14.   script.wait(1)
15.
16. #движение по сантиметрам (дистанция в см, скорость, тип ('line', ''))
17. def moveCM(cm, sp, type):
18.    if type == 'line':
19.      brick.encoder("E4").reset()
20.      script.wait(10)
21.      en = (cpr * cm) / (math.pi * d)
22.      while abs(brick.encoder("E4").read()) < en:
23.           go(sp)
24.      move(0, 0, 0, 0)
25.    else:
26.      brick.encoder("E4").reset()
27.      script.wait(10)
28.      move(sp, sp, sp, sp)
29.      en = (cpr * cm) / (math.pi * d)
30.      while abs(brick.encoder("E4").read()) < en:
31.           script.wait(10)
32.      move(0, 0, 0, 0)
```

Функции взяты из кода участников, поэтому рассматриваются в исходном виде. Во входных параметрах функции *moveCM()* указывается не только расстояние и скорость перемещения, но и «тип» движения: по линии или прямолинейно в направлении «вперед». В результате проверки этого типа вызывается тот или иной участок кода и вложенные функции. При движении по линии в строке 23 вызывается вложенная функция *go()* для движения по линии на ПД-регуляторе. При движении вне линии в строке 28 вызывается функция *move()* для запуска моторов с заданными скоростями. Особого внимания в этом примере заслуживают функция *go()*, реализующая движение по линии (она универсальна и может использоваться в других функциях) и строки 21 / 29, в которых происходит пересчет заданного расстояния в «тики» энкодера.

Движение по линии до перекрестка:

```
1.  #движение по линии по левому датчику, смотрим линии по правому
2.  def perec_1_r_vn(sp):
3.     s1 = brick.sensor("A5").read
4.     s2 = brick.sensor("A6").read
5.     brick.sensor("A5").readRawData()
6.     brick.sensor("A6").readRawData()
7.     while s2() < light_2:
8.          u = (light_1 - s1()) * k
9.          move(sp + u,sp - u, sp + u, sp - u)
10.         script.wait(10)
11.    move(0,0)
```

В этом примере робот двигается и выравнивается за счет левого датчика, а линия определяется правым. В зависимости от алгоритма выполнения задачи

командой движение и датчики могут отличаться. В условии в строке 7 используется глобальная переменная *light_2* с пороговым значением черного на правом датчике. В строках 8–10 реализуется движение по линии на П-регуляторе.

Движение вдоль стены по датчику расстояния и проезд по линии до стены на заданное расстояние:

```
1.  #движение по стенке (скорость)
2.  def go_wall(sp):
3.      s = brick.sensor("D2").read
4.      while s() < 60:
5.          s = brick.sensor("D2").read
6.          v = (14 - s()) * 5
7.          move(sp + v, sp - v, sp + v, sp - v)
8.      move(0, 0, 0, 0)
9.  #движение по линии до стенки
10. def go_to_wall(sp, dist):
11.     s = brick.sensor("D1").read
12.     while s() > dist:
13.         s = brick.sensor("D1").read
14.         go(sp)
15.     move(0, 0, 0, 0)
```

В функции *go_wall()* расстояние до стены задается прямо в коде – 14 см, - но удобнее было бы реализовать передачу его как входной параметр функции. Это расстояние используется в качестве уставки П-регулятора, реализованного в строках 5-7. Передача расстояния в качестве параметра показана в функции *go_to_wall()*. Первая функция используется для движения вдоль стены, вторая – для проезда к стене.

Комбинируя подобные функции, можно добиться стабильного движения робота между точками и зонами полигона. При составлении иерархии функций следует придерживаться следующих правил:

1. На первом этапе необходимо добиться стабильного движения робота в свободном направлении, по линии, вдоль стены и т.д.; для этого разумно использовать регуляторы, реализовав их в виде базовых функций;
2. Далее на основе базовых функций из первого этапа следует создать функции с выполнением отдельных законченных действий: перемещение робота на заданное расстояние, до перекрестка и прочие;
3. На основе функций из второго этапа следует создать готовые «макрокоманды» для перемещения между зонами и отладить их; между этими «макрокомандами» могут добавляться команды считывания ArTag-меток, захвата и выгрузки объектов, логика выбора «макрокоманд» в зависимости от жеребьевки конкретной попытки.

Приведенные выше примеры подходят для 1-2 этапов. Далее рассмотрим работу с изображением и распознавание ArTag-меток. Всю ее можно разложить на следующие этапы:

1. получение изображения с камеры;

2. перевод изображения в оттенки серого – так как сама ArTag-метка черно-белая и цвет не несет информации, то для удобства и ускорения выполнения следующих операций разумно перевести кадр в формат с оттенками серого;
3. бинаризация – преобразование в строгий ч/б формат, в котором остается либо строго черный цвет, либо строго белый;
4. поиск углов метки на изображении – так как метка может занимать не весь кадр и быть искажена из-за съемки под углом (при этом прямоугольная метка воспринимается как трапеция), то необходимо найти ее углы и в дальнейшем использовать их при вычислении координат отдельных бит на ней;
5. распознавание отдельных бит на метке – определение координат отдельных бит и их распознавание, считывание информации из метки;
6. определение ориентации метки – так как метка может быть повернута или перевернута, то информация может быть считана неверно, для этого три угла метки строго черного цвета, а четвертый – строго белого, которые необходимо считать и распознать.

Получение изображения с камеры выполняется буквально одной командой *getPhoto()*. Ниже приведен пример из кода участников с незначительными изменениями – удалены закомментированные команды (язык программирования Python):

```
1.  def artag():
2.      pic = getPhoto()
3.      pic = grayScale(pic)
4.      pic = binarization(60, pic)
5.
6.      brick.display().show(pic, w, h, "grayscale8")
7.      brick.display().redraw()
8.
9.      brick.display().show(pic, w, h, "grayscale8")
10.     num = dots(pic)
11.     brick.display().redraw()
12.
13.     brick.display().clear()
14.     brick.display().setBackground("white")
15.     brick.display().addLabel(str(num[0]) + ' ' + str(num[1]) + ' ' +
    str(num[2]) + ' ' + str(num[3]), 35, 10 + 30 * 2)
16.     brick.playTone(4000, 500)
17.     brick.display().redraw()
18.
19.     return num
```

В строке 3 из этого примера вызывается функция grayScale(), выполняющая перевод изображения в оттенки серого. Ниже представлен код этой функции от тех же участников:

```
1.  def grayScale(sPic):
2.    bufPic = []
3.    for i in range(len(sPic)):
4.      p = sPic[i]
5.      r = (p & 0xff0000) >> 16
6.      g = (p & 0xff00) >> 8
7.      b = (p & 0xff)
8.      p = r * 0.299 + g * 0.587 + b * 0.114
9.      bufPic.append(p)
10.   return bufPic
```

В строках 4-9 происходит перебор пикселей, снятых камерой в формате RGB32, и разбирает их на три составляющих *r*, *g* и *b*. Перевод осуществляется за счет масок и сдвиговых операций на 16, 8 или 0 бит. Общий уровень серого для каждого пикселя вычисляется по формуле в строке 8. В строке 9 уровень серого каждого пикселя собирается в массив, который и возвращается как результат работы функции.

Бинаризация производится функцией *binarization()*, код которой приведен ниже:

```
1.  def binarization(threshold, sPic):
2.    bufPic = []
3.    for i in range(len(sPic)):
4.      if sPic[i] > threshold:
5.        bufPic.append(255)
6.      else:
7.        bufPic.append(0)
8.    return bufPic
```

Вся суть бинаризации сводится к простому сравнению уровня серого каждого пикселя с неким пороговым значением. Если уровень выше порога, то пикселю присваивается максимальная яркость (белый цвет), если меньше – минимальная (черный цвет).

В коде функции *artag()*, приведенном ранее, после бинаризации следует вывод получившегося изображения на экран контроллера. Это не обязательны шаг, он лишь облегчает отладку команде. Следующие этапы обработки изображения объединены в функцию *dots()*, вызываемую в строке 10. И далее опять происходит вывод сообщения на экран контроллера, но это уже обязательный вывод, за который начисляются баллы.

Ниже приведен код функции *dots()* и вызываемых в ней вложенных функций для определения углов и ориентации метки, распознавания отдельных бит:

```
1.    def bfs(start, dirs, m):
2.        layer = {start}
3.        timer3 = script.timer(1000)
4.        timer3.timeout.connect(blink)
5.        while layer:
6.            nextlayer = set()
7.            for el in layer:
8.                if check(el, m, dirs):
9.                    timer3.stop()
10.                   return el
11.               if 0 <= el[0] + dirs[0] <= w - 1 and 0 <= el[1] <= h - 1:
12.                   nextlayer.add((el[0] + dirs[0], el[1]))
13.               if 0 <= el[0] <= w - 1 and 0 <= el[1] + dirs[1] <= h - 1:
14.                   nextlayer.add((el[0], el[1] + dirs[1]))
15.           layer = nextlayer
16.       timer3.stop()
17.
18.   def dot(x, y, r=3):
19.      brick.display().drawRect(int(x * 240 / 160), int(y * 280 / 120), r, r,
      True)
20.
21.
22.   def corners(pic):
23.     x1 = bfs((offset, offset), [1, 1], pic)
24.     x2 = bfs((offset, h - 1 - offset), [1, -1], pic)
25.     x3 = bfs((w - 1 - offset, offset), [-1, 1], pic)
```

```
    x4 = bfs((w - 1 - offset, h - 1 - offset), [-1, -1], pic)

    brick.display().setPainterColor('darkRed')
    dot(*x1, 10)
    dot(*x2, 10)
    dot(*x3, 10)
    dot(*x4, 10)
    brick.display().redraw()

    return x1, x2, x3, x4

  def dots(pic):

    x1, x2, x3, x4 = corners(pic)
    brick.display().setPainterColor('yellow')
    dots = [[(0, 0) for i in range(6)] for i in range(6)]

    dots[0][0] = x1
    dots[5][0] = x2
    dots[0][5] = x3
    dots[5][5] = x4

    k1 = [(x1[0] - x3[0]) / 5, (x1[1] - x3[1]) / 5]
    k2 = [(x3[0] - x4[0]) / 5, (x3[1] - x4[1]) / 5]
    k3 = [(x4[0] - x2[0]) / 5, (x4[1] - x2[1]) / 5]
    k4 = [(x2[0] - x1[0]) / 5, (x2[1] - x1[1]) / 5]

    for i in range(1, 5):
     dots[0][i] = (x1[0] - k1[0] * i, x1[1] - k1[1] * i)
     dots[i][5] = (x3[0] - k2[0] * i, x3[1] - k2[1] * i)
     dots[5][5-i] = (x4[0] - k3[0] * i, x4[1] - k3[1] * i)
     dots[5-i][0] = (x2[0] - k4[0] * i, x2[1] - k4[1] * i)
     dot(*dots[0][i], 5)
     dot(*dots[i][5], 5)
     dot(*dots[5][5-i], 5)
     dot(*dots[5-i][0], 5)

    brick.display().redraw()
    brick.display().setPainterColor('blue')
    for i in range(1, 5):
       for j in range(1, 5):
        dots[i][j] = line_intersection((dots[0][i], dots[5][i]),
(dots[j][0], dots[j][5]))
        dot(*dots[i][j])
    brick.display().redraw()

    brick.display().setPainterColor('green')

    centers = [[(0, 0) for i in range(3)] for i in range(3)]
    values = [[-1 for i in range(3)] for i in range(3)]

    for i in range(1, 4):
       for j in range(1, 4):
          i1, i2 = i - 1, 3 - j
          centers[i1][i2] = line_intersection([dots[i][j], dots[i + 1][j +
1]], [dots[i][j + 1], dots[i + 1][j]])
          values[i1][i2] = get(*centers[i1][i2], pic) == 0
          if values[i1][i2]:
            dot(*centers[i1][i2])
    ctr = 0
    while values[-1][-1] and ctr < 10:
      values = list(zip(*values[::-1]))
      ctr += 1
    nums = [int(values[0][1]), int(values[1][0]), int(values[1][2]),
int(values[2][1])]
    print(values, nums)
    return nums

  def line_intersection(line1, line2):
      xdiff = (line1[0][0] - line1[1][0], line2[0][0] - line2[1][0])
      ydiff = (line1[0][1] - line1[1][1], line2[0][1] - line2[1][1])

      def det(a, b):
          return a[0] * b[1] - a[1] * b[0]

```

```
98.       div = det(xdiff, ydiff)
99.       if div == 0:
100.          return (0, 0)
101.
102.      d = (det(*line1), det(*line2))
103.      x = det(d, xdiff) / div
104.      y = det(d, ydiff) / div
105.
106.      return (int(x), int(y))
```

В приведенном коде удалены закомментированные участки кода, но оставлены команды для вывода сообщений на экран, так как они могут облегчить отладку. С другой стороны, эти команды могут значительно замедлить распознавание ArTag-метки, вплоть до остановки робота на 20 и более секунд, что повлечет остановку попытки судьями. Решение об удалении или комментировании отладочных команд принимает участники принимали исходя из скорости распознавания меток во время тренировок.

После вызова функции *dots()* в функции *artag()* производится вывод считанных бит на экран и вызов звукового оповещения. Вне функции *artag()* необходимо еще реализовать остановку робота на 5 секунд, чтобы судьи успели зафиксировать выведенное на экран сообщение и начислить за это баллы.

Для работы с кубиками роботу требуется осознанно управлять захватом (манипулятором). Так как в нем используются сервоприводы, то управление достаточно простое: указывается желаемое положение сервопривода и устанавливается пауза для того, чтобы привод успел прийти в нужное положение. Ниже приведен код одной из команд, реализовавшей для управления манипулятором класс с заранее указанными градусами и паузами, которые подбираются экспериментально в зависимости от механики захвата (язык программирования Python):

```
1.  class Manipulator:
2.      @staticmethod
3.      def set_position(servo, type):
4.          if servo == 'big':
5.              if type == 'up':
6.                  brick.motor("S2").setPower(90)
7.              elif type == 'down':
8.                  brick.motor("S2").setPower(-35)
9.          elif servo == 'small':
10.             if type == 'open':
11.                 brick.motor("S1").setPower(-50)
12.             elif type == 'closed':
13.                 brick.motor("S1").setPower(25)
14.         script.wait(1000)
15.
16.     @staticmethod
17.     def grab_up():
18.         Manipulator.set_position('small', 'open')
19.         Manipulator.set_position('small', 'closed')
```

Имея набор функций, примеры которых приведены выше, можно собрать программу для решения задачи финала.

Расстановка препятствий и двух игровых объектов были известны в течение всего времени подготовки. То есть, можно было как проложить маршрут движения

мимо препятствий, так и выполнять доставку первых двух элементов без считывания ArTag-меток. Жеребьевка расположения препятствий позволяла свободно перемещаться между зонами старта и погрузки. Далее можно было либо двигаться вдоль правого борта полигона (удерживая расстояние до стены по УЗ-датчику), либо выравниваться по линиям и перекресткам и в дальнейшем двигаться по заданному азимуту (удерживая направление по IMU-датчику). Аналогично с возвратом от зоны выгрузки в зону погрузки или финиша.

Логика выбора сектора в зоне выгрузки для двух кубиков (адреса доставки которых известны заранее) могла быть указана напрямую в коде программы. Для двух других кубиков логика доставки могла быть реализована инструментами поиска и выбора элементов в массиве по индексу / значению, задачи на знания подобных инструментов предлагались участникам во время первого и второго отборов.