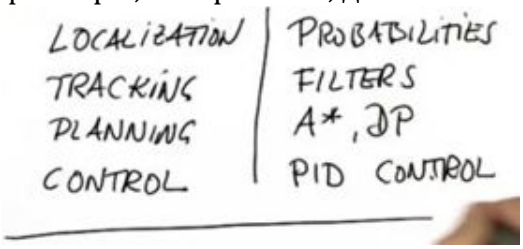
Переведенная версия
Udacity CS373: Программирование роботизированных автомобилей
Раздел 6: Собираем все вместе

Добро пожаловать в раздел 6!
В этой заключительной части, вам предстоит собрать все, что вы узнали до сих пор, вместе.
Вы изучали: локализацию, отслеживание, планирование, управление, вероятности, фильтры, алгоритм A*, динамическое программирование, PID.



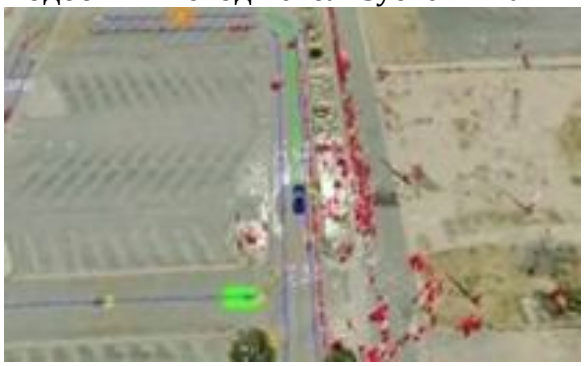
Это очень много материала!
В дополнение к изучению этих понятий вы также изучали программы по реализации этих вещей.
Теперь вы готовы взять все эти кусочки и сложить их вместе, чтобы построить действительно автономный робот.

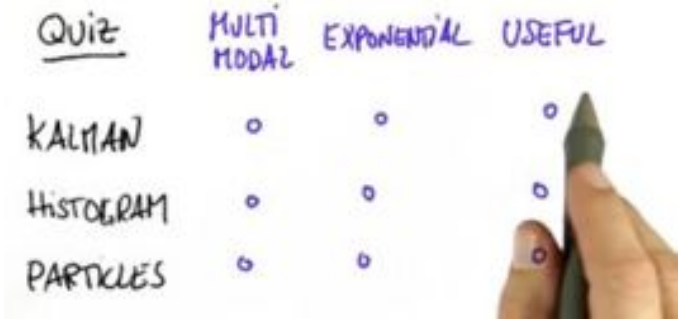
Обновим в памяти пройденный материал
Q6-2: Локализация
Ранее вы узнали, как карты используются для локализации движущегося робота.

На этом изображении вы можете видеть результат лазерного сканирования, и построения карты, которое производится, в этом случае автомобилем Стэндфордского университета.
И мы использовали частицы (particles) чтобы локализовать этот робот путем сопоставления мгновенного измерения с ранее полученной картой.

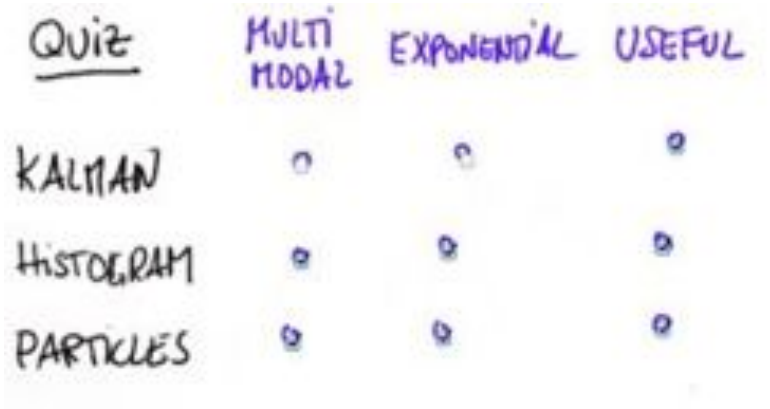
С помощью этой информации Junior мог выполнять локализацию на DARPA Urban Challenge с сантиметровой точностью.


Подобный метод используется в машине Google StreetView.

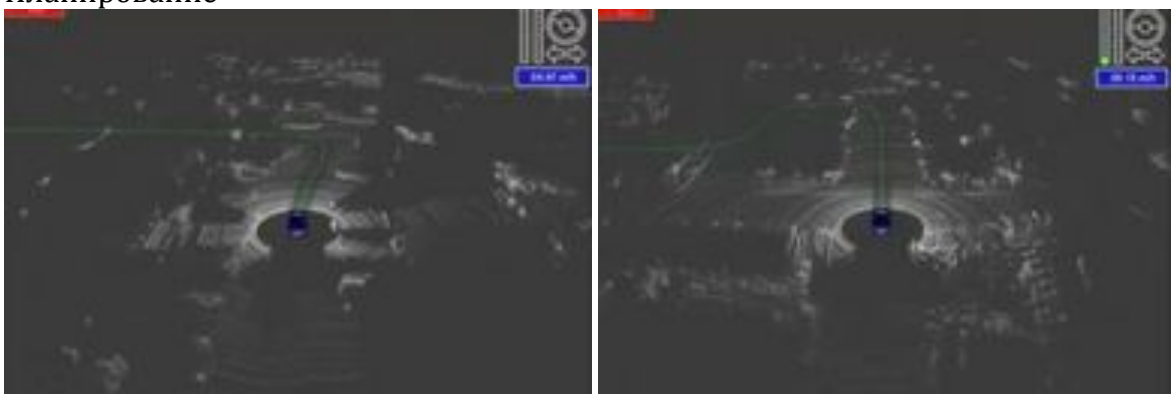

Ранее вы узнали о фильтрах Калмана, гистограммные фильтры (на первом занятии) и фильтре частиц (particle filter).
В следующем тесте выберите соответствующий вариант атрибута для каждого фильтра.



1. Есть ли среди этих мультимодальных фильтры в их распределении?
2. Вычислительная сложность фильтра растет экспоненциально относительно числа измерений?
3. Любой из этих фильтров полезен в контексте робототехники?
Правильный ответ:



Планирование

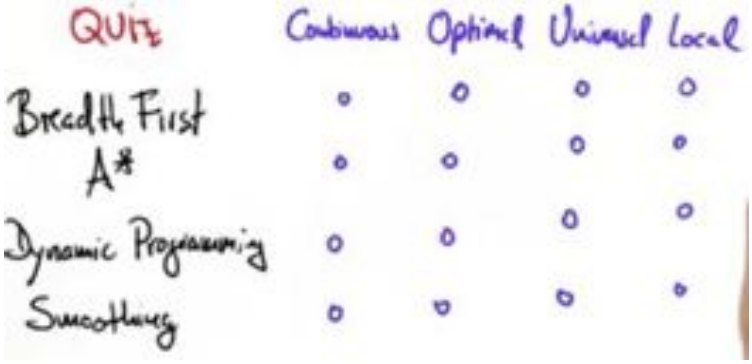


Вспомните рассмотренные алгоритмы планирования: поиск в ширину, планирование A*, планирование с использованием динамического программирования.

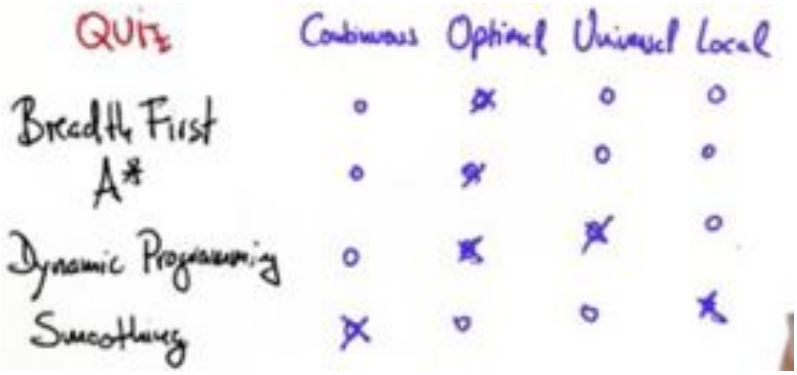
Q6-3: Планирование

Отметьте любой или все эти квадраты, если вы думаете, что соответствующий алгоритм планирования:

1. имеет дело с непрерывным пространством,
2. находит оптимальное решение,
3. строит универсальный план (это означает, что найденное решение может быть применено к произвольному состоянию)
4. решение является локальным, т.е. корректирует полученный план.

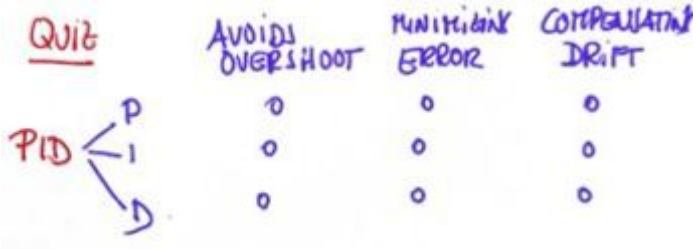


Ответ:

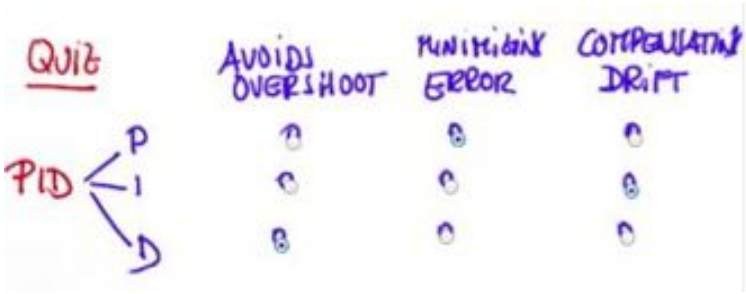


PID
Вы также узнали об управлении и реализовали регулятор.
Регулятор – это то, что делает возможным такие вещи, как автономная парковка.
Дополнительный материал : См. http://www.youtube.com/watch?v=gzI54rm9m1Q

Q6-4: PID
Как вы знаете, ПИД-регулятор включает составляющие P, I и D.
Пожалуйста, отметьте, какая из составляющих связана больше всего с:
1. избеганием перерегулирования (Avoiding overshoot),
2. минимизацией ошибки относительно заданной траектории (Minimizing error)
3. компенсацией систематической ошибки (Compensating drift).



Правильный ответ:



Соберем все это вместе теперь в единую программу!
Себастьяну потребовался целый день, чтобы сделать это, но вам не придется делать все это самостоятельно. Однако, хотя вы будете иметь некоторую помощь (потому что Себастьян знает, что вы занятой студент) вы все равно должны быть в состоянии выполнить все задания самостоятельно.

Описание класса робота:
Существует класс робота, который имеет определенные характеристики погрешностей, связанных с ним.

```
steering_noise    = 0.1
distance_noise    = 0.03
measurement_noise = 0.3
```

Существует знакомые функция __init__,

```
    # --------
    # init:
    #creates robot and initializes location/orientation to 0, 0, 0
    #

    def __init__(self, length = 0.5):
        self.x = 0.0
        self.y = 0.0
        self.orientation = 0.0
        self.length = length
        self.steering_noise    = 0.0
        self.distance_noise    = 0.0
        self.measurement_noise = 0.0
        self.num_collisions    = 0
        self.num_steps         = 0
```

функция задания положения,

```
    # --------
    # set:
    #sets a robot coordinate
    #

    def set(self, new_x, new_y, new_orientation):

        self.x = float(new_x)
        self.y = float(new_y)
        self.orientation = float(new_orientation) % (2.0 * pi)
```

а функция задания погрешностей set_noise.

```
    # --------
    # set_noise:
    #sets the noise parameters
    #
```

```
    def set_noise(self, new_s_noise, new_d_noise, new_m_noise):
        # makes it possible to change the noise parameters
        # this is often useful in particle filters
        self.steering_noise     = float(new_s_noise)
        self.distance_noise    = float(new_d_noise)
        self.measurement_noise = float(new_m_noise)
```

Есть два новых функции: check_collision которая проверяет если у вас есть столкновения в мире заданном сеткой grid (с границами сетки либо препятствием),

```
    # --------
    # check:
    #    checks of the robot pose collides with an obstacle, or
    # is too far outside the plane

    def check_collision(self, grid):
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == 1:
                    dist = sqrt((self.x - float(i)) ** 2 +
                                (self.y - float(j)) ** 2)
                    if dist < 0.5:
                        self.num_collisions += 1
                        return False
        return True
```

функция check_goal, которая проверяет, достигли ли вы цели в соответствии с определенным порогом расстояния.

```
    def check_goal(self, goal, threshold = 1.0):
        dist =  sqrt((float(goal[0]) - self.x) ** 2 + (float(goal[1])
- self.y) ** 2)
        return dist < threshold
```

Функция move должна быть хорошо вам знакома.

```
    # --------
    # move:
    #    steering = front wheel steering angle, limited by max_steering_angle
    #    distance = total distance driven, most be non-negative

    def move(self, grid, steering, distance,
             tolerance = 0.001, max_steering_angle = pi / 4.0):

        if steering > max_steering_angle:
            steering = max_steering_angle
        if steering < -max_steering_angle:
            steering = -max_steering_angle
        if distance < 0.0:
            distance = 0.0


        # make a new copy
        res = robot()
        res.length            = self.length
        res.steering_noise    = self.steering_noise
        res.distance_noise    = self.distance_noise
        res.measurement_noise = self.measurement_noise
        res.num_collisions    = self.num_collisions
        res.num_steps         = self.num_steps + 1

        # apply noise
        steering2 = random.gauss(steering, self.steering_noise)
        distance2 = random.gauss(distance, self.distance_noise)
```

```
        # Execute motion
        turn = tan(steering2) * distance2 / res.length

        if abs(turn) < tolerance:

            # approximate by straight line motion

            res.x = self.x + (distance2 * cos(self.orientation))
            res.y = self.y + (distance2 * sin(self.orientation))
            res.orientation = (self.orientation + turn) % (2.0 * pi)

        else:

            # approximate bicycle model for motion

            radius = distance2 / turn
            cx = self.x - (sin(self.orientation) * radius)
            cy = self.y + (cos(self.orientation) * radius)
            res.orientation = (self.orientation + turn) % (2.0 * pi)
            res.x = cx + (sin(res.orientation) * radius)
            res.y = cy - (cos(res.orientation) * radius)

        # check for collision
        # res.check_collision(grid)

        return res
```

Функция sense очень проста – она измеряет положение робота, как GPS на автомобиле, с учетом шума измерения.

```
    # --------
    # sense:
    #

    def sense(self):

        return [random.gauss(self.x, self.measurement_noise),
                random.gauss(self.y, self.measurement_noise)]
```

Функция measurement_probability, которую вы, возможно, захотите использовать в фильтре, оценивает погрешности измерений координат робота, используя гауссиан.

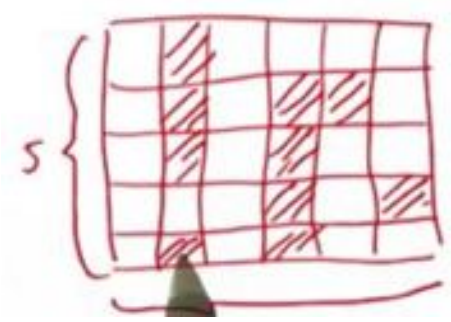
```
    # --------
    # measurement_prob
    #    computes the probability of a measurement
    #

    def measurement_prob(self, measurement):

        # compute errors
        error_x = measurement[0] - self.x
        error_y = measurement[1] - self.y

        # calculate Gaussian
        error = exp(- (error_x ** 2) / (self.measurement_noise ** 2) / 2.0) \
            / sqrt(2.0 * pi * (self.measurement_noise ** 2))
        error *= exp(- (error_y ** 2) / (self.measurement_noise ** 2) / 2.0)
\
            / sqrt(2.0 * pi * (self.measurement_noise ** 2))

        return error
```

Вооружившись всем этим, нужно решить следующую проблему!

Дано:
Сетка

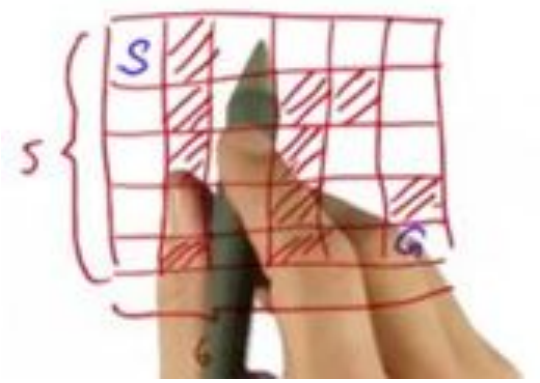
```
# grid format:
#   0 = navigable space
#   1 = occupied space

grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 1, 1, 0],
        [0, 1, 0, 1, 0, 0],
        [0, 0, 0, 1, 0, 1],
        [0, 1, 0, 1, 0, 0]]
```

В данном случае она выглядит так:

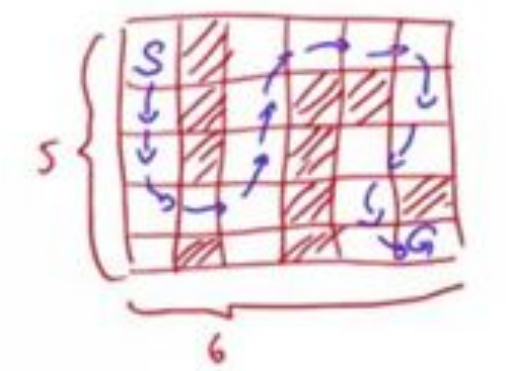


Даны начальное и целевое положения робота:

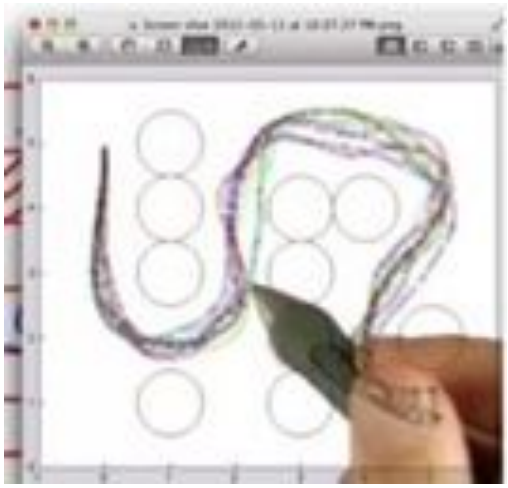
```
init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1]
```



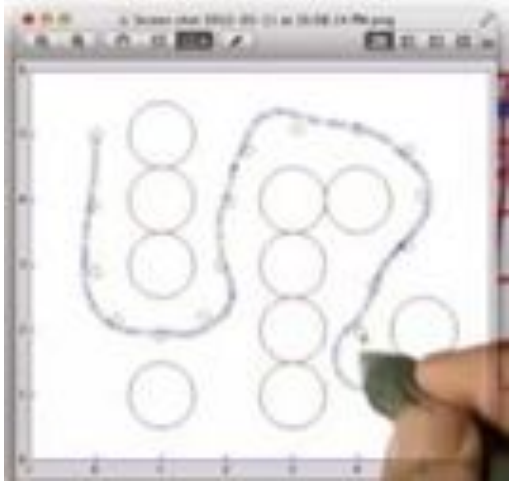
Задача:
Построить систему управления автомобилем для движения из начального положения в целевую точку.



Ниже вы можете увидеть, как решение реализовано Себастьяном.



Препятствия изображены как круги.
Показаны траектории робота после нескольких запусков. Их отличие связано со случайными шумами измерений и погрешностями движения.
Одна траектория более детально:



Вы также можете увидеть целевые точки – большие зеленые круги, которые являются сглаженными точками A*-планировщика.

На самом деле, в коде уже реализованы планировщик, фильтр частиц и регулятор. Вам нужно изучить его. Некоторые моменты рассмотрим ниже как упражнения.

Для начала рассмотрим упрощенный алгоритм управления. Робот всегда пытается повернуть к цели и едет вперед со скоростью 0.1. Каждый раз, когда он сталкивается с препятствием или границей, выводится сообщение.

```
195 while not myrobot.check_goal(goal):
196     theta = atan2(goal[1] - myrobot.y, goal[0] - myrobot.x) - myrobot.orientation
197     myrobot = myrobot.move(grid, theta, 0.1)
198     if not myrobot.check_collision(grid):
199         print '##### Collision ####'
200     print myrobot
```

На самом деле алгоритм выбирает в качестве цели управления идти прямо к цели, с использованием функции atan2.
Вывод программы:

```
[x=0.10399 y=0.01274 orient=0.24379]
[x=0.20111 y=0.04377 orient=0.37476]
[x=0.30213 y=0.09018 orient=0.48661]
[x=0.34724 y=0.11584 orient=0.54754]
[x=0.42914 y=0.17173 orient=0.65008]
[x=0.49369 y=0.22412 orient=0.71351]
[x=0.56464 y=0.29233 orient=0.79029]
[x=0.62578 y=0.35333 orient=0.81132]
[x=0.72258 y=0.46156 orient=0.87088]
[x=0.78644 y=0.54056 orient=0.91089]
##### Collision ####
[x=0.83841 y=0.60939 orient=0.93737]
##### Collision ####
[x=0.88160 y=0.66902 orient=0.95067]
##### Collision ####
[x=0.93017 y=0.73780 orient=0.96112]
##### Collision ####
[x=0.99721 y=0.83335 orient=0.95684]
##### Collision ####
[x=1.05073 y=0.90559 orient=0.91662]
##### Collision ####
[x=1.10500 y=0.97751 orient=0.82852]
##### Collision ####
[x=1.18166 y=1.07497 orient=0.89829]
##### Collision ####
[x=1.24250 y=1.15303 orient=0.89393]
##### Collision ####
[x=1.30913 y=1.23563 orient=0.89014]
##### Collision ####
[x=1.36236 y=1.30487 orient=0.94061]
[x=1.43459 y=1.40375 orient=0.93913]
[x=1.49210 y=1.48082 orient=0.92050]
[x=1.55274 y=1.56070 orient=0.92247]
[x=1.59813 y=1.62257 orient=0.95320]
[x=1.68775 y=1.75507 orient=0.99899]
[x=1.76209 y=1.87230 orient=1.01228]
[x=1.82006 y=1.95866 orient=0.94702]
[x=1.85821 y=2.01273 orient=0.96568]
[x=1.89461 y=2.06536 orient=0.96572]
[x=1.94350 y=2.13610 orient=0.97158]
[x=1.97528 y=2.18200 orient=0.95061]
[x=2.03340 y=2.26805 orient=1.00301]
[x=2.09053 y=2.36090 orient=1.03531]
[x=2.15359 y=2.46272 orient=0.99721]
[x=2.18975 y=2.51846 orient=0.99358]
##### Collision ####
[x=2.24548 y=2.60001 orient=0.94911]
##### Collision ####
[x=2.30711 y=2.68530 orient=0.94095]
##### Collision ####
[x=2.37176 y=2.77770 orient=0.97967]
##### Collision ####
[x=2.45045 y=2.89275 orient=0.96216]
##### Collision ####
[x=2.53409 y=3.01693 orient=0.99386]
##### Collision ####
[x=2.56974 y=3.07068 orient=0.97674]
##### Collision ####
[x=2.62932 y=3.15799 orient=0.96717]
##### Collision ####
[x=2.66578 y=3.21097 orient=0.96890]
##### Collision ####
[x=2.73859 y=3.32338 orient=1.02308]
##### Collision ####
```

```
[x=2.80073 y=3.42160 orient=0.99041]
[x=2.88287 y=3.54889 orient=1.00508]
[x=2.94144 y=3.64115 orient=1.00522]
[x=2.99035 y=3.71669 orient=0.98719]
[x=3.04862 y=3.80548 orient=0.99303]
[x=3.12166 y=3.91970 orient=1.01061]
[x=3.18221 y=4.00833 orient=0.93226]
```

Если вы посмотрите на вывод программы - координаты и ориентацию - можно увидеть частые столкновения, которым робот подвергается в своей попытке достичь цели. Робот в конечном счете действительно достигает своей цели, но вы можете видеть два больших региона столкновений.

Движемся далее. Теперь у нас более сложный пример. Есть функция main, которая вызывает планировщик A*, сглаживает полученную траекторию, а затем вызывает регулятор (функция run) для движения по ней.

```
# ------------------------------------------------
#
# this is our main routine
#

def main(grid, init, goal, steering_noise, distance_noise,
measurement_noise,
         weight_data, weight_smooth, p_gain, d_gain):

    path = plan(grid, init, goal)
    path.astar()
    path.smooth(weight_data, weight_smooth)
    return run(grid, goal, path.spath, [p_gain, d_gain])
```

Внутри функции run также используется фильтр частиц.

```
# --------
#
# run:  runs control program for the robot
#


def run(grid, goal, spath, params, printflag = False, speed = 0.1, timeout =
1000):

    myrobot = robot()
    myrobot.set(0., 0., 0.)
    myrobot.set_noise(steering_noise, distance_noise, measurement_noise)
    filter = particles(myrobot.x, myrobot.y, myrobot.orientation,
                       steering_noise, distance_noise, measurement_noise)

    cte  = 0.0
    err  = 0.0
    N    = 0

    index = 0 # index into the path

    while not myrobot.check_goal(goal) and N < timeout:

        diff_cte = - cte


        # ----------------------------------------
        # compute the CTE
```

```
        # start with the present robot estimate
        estimate = filter.get_position()

        ### ENTER CODE HERE


        # ----------------------------------------


        diff_cte += cte

        steer = - params[0] * cte - params[1] * diff_cte

        myrobot = myrobot.move(grid, steer, speed)
        filter.move(grid, steer, speed)

        Z = myrobot.sense()
        filter.sense(Z)

        if not myrobot.check_collision(grid):
            print '##### Collision ####'

        err += (cte ** 2)
        N += 1

        if printflag:
            print myrobot, cte, index, u

    return [myrobot.check_goal(goal), myrobot.num_collisions,
myrobot.num_steps]
```

В коде циклически рассчитывается траекторная ошибка, рассчитывается управления (с помощью ПД-контроллера, без И-составляющей) и используется фильтр частиц для оценки положения робота.
Однако этот код все еще немного неполный. Ваша задача – рассчитать траекторную ошибку для заданной траектории в виде ломаной.

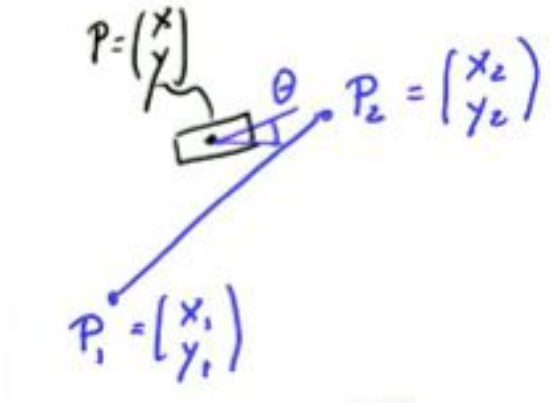
Q6-6: сегментированные CTE
Для этой задачи реализации расчета траекторной ошибки, используйте в качестве входной величины оценку позиции робота, выполнив команду filter.get_position().

```
        # ----------------------------------------
        # compute the CTE

        # start with the present robot estimate
        estimate = filter.get_position()
```

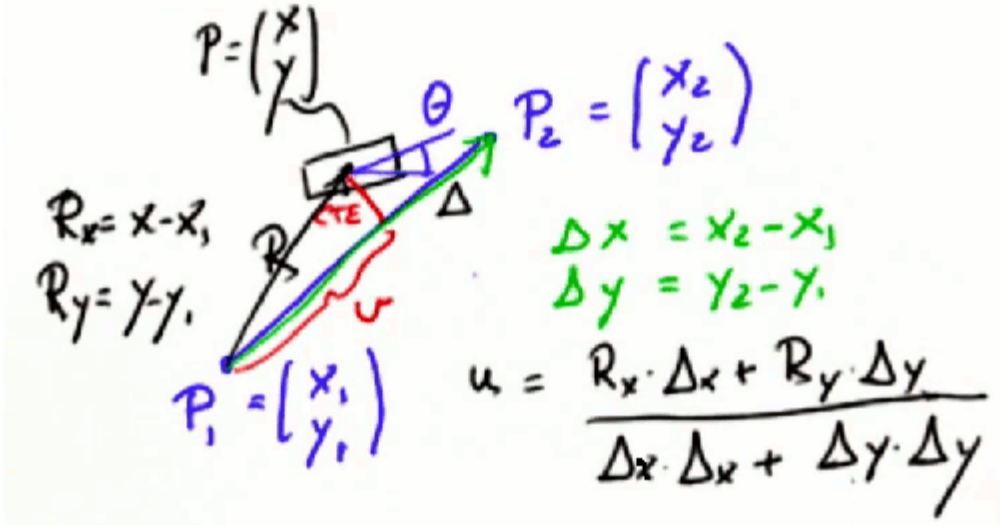
Вот трудная часть. Ваш путь теперь – последовательность сегментов.
Когда робот едет вдоль прямой (красные прямоугольники), он имеет определенную погрешность.

Однако, когда робот выезжает за конец отрезка (черный прямоугольник), вы должны изменить расстояние до соответствующего (следующего) сегмента траектории.

В дополнение к вычислению ошибки относительно произвольного отрезка, а не только Y-оси, вы также должны определить, когда робот выходит за конец сегмента и и перейти на следующий.
Предположим, что это ваш отрезок, и путь он задается координатами начальной точки P1 и конечной точки P2. Предположим, позиция робота P (x, y), где (x, y) – оценки от фильтра частиц в нашем случае, и ориентация робота – тета.



Обе нужных нам величины – траекторная ошибка, и расстояние на которое продвинулся робот вдоль сегмента – могут быть рассчитаны с использованием скалярного произведения вектором.



В частности, можно обозначить зеленый вектор Δ {Δx, Δy}, где Δx=x2 - x1 и Δy=y2 - y1. Рассмотрим черный вектор R из точки P1 в P(x,y). Для него Rx = x-x1 и Ry = y - y1. Тогда величина u, которая обозначает расстояние, на которое продвинулся робот вдоль сегмента, рассчитывается как:

$$u = \frac{R_x \Delta x + R_y \Delta y}{\Delta x^2 + \Delta y^2}$$

Это скалярное произведение черного вектора R и зеленого вектора Δ.
Если u больше, чем 1, это означает, что робот покинул сегмент должен перейти на следующей.
Наконец, траекторная ошибка, красная линия, задается аналогично:

$$CTE = \frac{R_y \Delta_x - R_x \Delta_y}{\Delta x^2 + \Delta y^2}$$

Включите эти уравнения в код программы.

Изучая код, вы увидите, что переменная называемая индекс была создана для Вас:

```
    index = 0 # index into the path
```

При u > 1, индекс должен инкрементироваться, однако, нужно проверять чтобы он никогда не выходил за длину пути.
Траекторная ошибка должна быть вычислена относительно сегмента с текущим индексом, используя скалярное произведение, описанное ранее.
Вы должны использовать путь, называемый spath. Текущий сегмент можно получить так:

```
spath[index][0] # x coordinate at index
spath[index][1] # y coordinate at index
```

Когда вы запустите программу с реализованным кодом, вы получите действительно хорошую траекторию, по большей части, без столкновений.
В данном примере это потребует около 130 шагов робота. Типичный ответ программы будет выглядеть так: [True, 0, 137]. True значит, что робот на самом деле нашел цель, 0 означает отсутствие столкновений, а 137 означает, что он выполнил 137 шагов.

Ответ:
Вот код, реализующий описанную выше задачу:

```
        # ----------------------------------------
        # compute the CTE

        # start with the present robot estimate
        estimate = filter.get_position()

        ### ENTER CODE HERE
        # some basic vector calculations
        dx = spath[index+1][0] - spath[index][0]
        dy = spath[index+1][1] - spath[index][1]
        drx = estimate[0] -  spath[index][0]
        dry = estimate[1] -  spath[index][1]
        # u is the robot estimate projected into the path segment
        u = (drx * dx + dry * dy)/(dx * dx + dy * dy)
        #the cte is the estimate projected onto the normal of the path
segment
        cte = (dry * dx - drx * dy)/(dx * dx + dy * dy)
        if u > 1:
            index += 1
```

```
        # ----------------------------------------
```

Полный код программы, длинный как питон:

```
# -----------
# User Instructions
#
# Familiarize yourself with the code below. Most of it
# reproduces results that you have obtained at some
# point in this class. Once you understand the code,
# write a function, cte, in the run class that
# computes the crosstrack
# error for the case of a segmented path. You will
# need to include the equations shown in the video.
#

from math import *
import random


# don't change the noise paameters

steering_noise    = 0.1
distance_noise    = 0.03
measurement_noise = 0.3


class plan:

    # --------
    # init:
    #    creates an empty plan
    #

    def __init__(self, grid, init, goal, cost = 1):
        self.cost = cost
        self.grid = grid
        self.init = init
        self.goal = goal
        self.make_heuristic(grid, goal, self.cost)
        self.path = []
        self.spath = []

    # --------
    #
    # make heuristic function for a grid

    def make_heuristic(self, grid, goal, cost):
        self.heuristic = [[0 for row in range(len(grid[0]))]
                          for col in range(len(grid))]
        for i in range(len(self.grid)):
            for j in range(len(self.grid[0])):
                self.heuristic[i][j] = abs(i - self.goal[0]) + \
                    abs(j - self.goal[1])



    # ------------------------------------------------
    #
    # A* for searching a path to the goal
    #
    #

    def astar(self):


        if self.heuristic == []:
            raise ValueError, "Heuristic must be defined to run A*"

        # internal motion parameters
        delta = [[-1,  0], # go up
                 [ 0, -1], # go left
                 [ 1,  0], # go down
                 [ 0,  1]] # do right
```

```
        # open list elements are of the type: [f, g, h, x, y]

        closed = [[0 for row in range(len(self.grid[0]))]
                  for col in range(len(self.grid))]
        action = [[0 for row in range(len(self.grid[0]))]
                  for col in range(len(self.grid))]

        closed[self.init[0]][self.init[1]] = 1


        x = self.init[0]
        y = self.init[1]
        h = self.heuristic[x][y]
        g = 0
        f = g + h

        open = [[f, g, h, x, y]]

        found  = False # flag that is set when search complete
        resign = False # flag set if we can't find expand
        count  = 0


        while not found and not resign:

            # check if we still have elements on the open list
            if len(open) == 0:
                resign = True
                print '###### Search terminated without success'

            else:
                # remove node from list
                open.sort()
                open.reverse()
                next = open.pop()
                x = next[3]
                y = next[4]
                g = next[1]

            # check if we are done

            if x == goal[0] and y == goal[1]:
                found = True
            # print '###### A* search successful'

            else:
                # expand winning element and add to new open list
                for i in range(len(delta)):
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
                    if x2 >= 0 and x2 < len(self.grid) and y2 >= 0 \
                        and y2 < len(self.grid[0]):
                            if closed[x2][y2] == 0 and self.grid[x2][y2] == 0:
                                g2 = g + self.cost
                                h2 = self.heuristic[x2][y2]
                                f2 = g2 + h2
                                open.append([f2, g2, h2, x2, y2])
                                closed[x2][y2] = 1
                                action[x2][y2] = i

            count += 1

        # extract the path



        invpath = []
        x = self.goal[0]
        y = self.goal[1]
        invpath.append([x, y])
        while x != self.init[0] or y != self.init[1]:
            x2 = x - delta[action[x][y]][0]
            y2 = y - delta[action[x][y]][1]
            x = x2
            y = y2
            invpath.append([x, y])

        self.path = []
```

```
        for i in range(len(invpath)):
            self.path.append(invpath[len(invpath) - 1 - i])



    # ----------------------------------------------
    #
    # this is the smoothing function
    #



    def smooth(self, weight_data = 0.1, weight_smooth = 0.1,
               tolerance = 0.000001):

        if self.path == []:
            raise ValueError, "Run A* first before smoothing path"

        self.spath = [[0 for row in range(len(self.path[0]))] \
                      for col in range(len(self.path))]
        for i in range(len(self.path)):
            for j in range(len(self.path[0])):
                self.spath[i][j] = self.path[i][j]

        change = tolerance
        while change >= tolerance:
            change = 0.0
            for i in range(1, len(self.path)-1):
                for j in range(len(self.path[0])):
                    aux = self.spath[i][j]

                    self.spath[i][j] += weight_data * \
                        (self.path[i][j] - self.spath[i][j])

                    self.spath[i][j] += weight_smooth * \
                        (self.spath[i-1][j] + self.spath[i+1][j]
                         - (2.0 * self.spath[i][j]))
                    if i >= 2:
                        self.spath[i][j] += 0.5 * weight_smooth * \
                            (2.0 * self.spath[i-1][j] - self.spath[i-2][j]
                             - self.spath[i][j])
                    if i <= len(self.path) - 3:
                        self.spath[i][j] += 0.5 * weight_smooth * \
                            (2.0 * self.spath[i+1][j] - self.spath[i+2][j]
                             - self.spath[i][j])

            change += abs(aux - self.spath[i][j])







# ----------------------------------------------
#
# this is the robot class
#

class robot:

    # --------
    # init:
    # creates robot and initializes location/orientation to 0, 0, 0
    #

    def __init__(self, length = 0.5):
        self.x = 0.0
        self.y = 0.0
        self.orientation = 0.0
        self.length = length
        self.steering_noise    = 0.0
        self.distance_noise    = 0.0
        self.measurement_noise = 0.0
        self.num_collisions    = 0
        self.num_steps         = 0
```

```
    # --------
    # set:
    # sets a robot coordinate
    #

    def set(self, new_x, new_y, new_orientation):

        self.x = float(new_x)
        self.y = float(new_y)
        self.orientation = float(new_orientation) % (2.0 * pi)


    # --------
    # set_noise:
    # sets the noise parameters
    #

    def set_noise(self, new_s_noise, new_d_noise, new_m_noise):
        # makes it possible to change the noise parameters
        # this is often useful in particle filters
        self.steering_noise     = float(new_s_noise)
        self.distance_noise    = float(new_d_noise)
        self.measurement_noise = float(new_m_noise)

    # --------
    # check:
    #    checks of the robot pose collides with an obstacle, or
    # is too far outside the plane

    def check_collision(self, grid):
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == 1:
                    dist = sqrt((self.x - float(i)) ** 2 +
                                (self.y - float(j)) ** 2)
                    if dist < 0.5:
                        self.num_collisions += 1
                        return False
        return True

    def check_goal(self, goal, threshold = 1.0):
        dist =  sqrt((float(goal[0]) - self.x) ** 2 + (float(goal[1]) - self.y) ** 2)
        return dist < threshold

    # --------
    # move:
    #    steering = front wheel steering angle, limited by max_steering_angle
    #    distance = total distance driven, most be non-negative

    def move(self, grid, steering, distance,
             tolerance = 0.001, max_steering_angle = pi / 4.0):

        if steering > max_steering_angle:
            steering = max_steering_angle
        if steering < -max_steering_angle:
            steering = -max_steering_angle
        if distance < 0.0:
            distance = 0.0


        # make a new copy
        res = robot()
        res.length            = self.length
        res.steering_noise    = self.steering_noise
        res.distance_noise    = self.distance_noise
        res.measurement_noise = self.measurement_noise
        res.num_collisions    = self.num_collisions
        res.num_steps         = self.num_steps + 1

        # apply noise
        steering2 = random.gauss(steering, self.steering_noise)
        distance2 = random.gauss(distance, self.distance_noise)


        # Execute motion
        turn = tan(steering2) * distance2 / res.length
```

```
        if abs(turn) < tolerance:

            # approximate by straight line motion

            res.x = self.x + (distance2 * cos(self.orientation))
            res.y = self.y + (distance2 * sin(self.orientation))
            res.orientation = (self.orientation + turn) % (2.0 * pi)

        else:

            # approximate bicycle model for motion

            radius = distance2 / turn
            cx = self.x - (sin(self.orientation) * radius)
            cy = self.y + (cos(self.orientation) * radius)
            res.orientation = (self.orientation + turn) % (2.0 * pi)
            res.x = cx + (sin(res.orientation) * radius)
            res.y = cy - (cos(res.orientation) * radius)

        # check for collision
        # res.check_collision(grid)

        return res

    # --------
    # sense:
    #

    def sense(self):

        return [random.gauss(self.x, self.measurement_noise),
                random.gauss(self.y, self.measurement_noise)]

    # --------
    # measurement_prob
    #    computes the probability of a measurement
    #

    def measurement_prob(self, measurement):

        # compute errors
        error_x = measurement[0] - self.x
        error_y = measurement[1] - self.y

        # calculate Gaussian
        error = exp(- (error_x ** 2) / (self.measurement_noise ** 2) / 2.0) \
            / sqrt(2.0 * pi * (self.measurement_noise ** 2))
        error *= exp(- (error_y ** 2) / (self.measurement_noise ** 2) / 2.0) \
            / sqrt(2.0 * pi * (self.measurement_noise ** 2))

        return error



    def __repr__(self):
        # return '[x=%.5f y=%.5f orient=%.5f]'  % (self.x, self.y, self.orientation)
        return '[%.5f, %.5f]'  % (self.x, self.y)




# --------------------------------------------------
#
# this is the particle filter class
#

class particles:

    # --------
    # init:
    # creates particle set with given initial position
    #

    def __init__(self, x, y, theta,
                 steering_noise, distance_noise, measurement_noise, N = 100):
        self.N = N
```

```
        self.steering_noise    = steering_noise
        self.distance_noise    = distance_noise
        self.measurement_noise = measurement_noise

        self.data = []
        for i in range(self.N):
            r = robot()
            r.set(x, y, theta)
            r.set_noise(steering_noise, distance_noise, measurement_noise)
            self.data.append(r)


    # --------
    #
    # extract position from a particle set
    #

    def get_position(self):
        x = 0.0
        y = 0.0
        orientation = 0.0

        for i in range(self.N):
            x += self.data[i].x
            y += self.data[i].y
            # orientation is tricky because it is cyclic. By normalizing
            # around the first particle we are somewhat more robust to
            # the 0=2pi problem
            orientation += (((self.data[i].orientation
                              - self.data[0].orientation + pi) % (2.0 * pi))
                            + self.data[0].orientation - pi)
        return [x / self.N, y / self.N, orientation / self.N]

    # --------
    #
    # motion of the particles
    #

    def move(self, grid, steer, speed):
        newdata = []

        for i in range(self.N):
            r = self.data[i].move(grid, steer, speed)
            newdata.append(r)
        self.data = newdata

    # --------
    #
    # sensing and resampling
    #

    def sense(self, Z):
        w = []
        for i in range(self.N):
            w.append(self.data[i].measurement_prob(Z))

        # resampling (careful, this is using shallow copy)
        p3 = []
        index = int(random.random() * self.N)
        beta = 0.0
        mw = max(w)

        for i in range(self.N):
            beta += random.random() * 2.0 * mw
            while beta > w[index]:
                beta -= w[index]
                index = (index + 1) % self.N
            p3.append(self.data[index])
        self.data = p3




# --------
#
```

```
# run:  runs control program for the robot
#


def run(grid, goal, spath, params, printflag = False, speed = 0.1, timeout = 1000):

    myrobot = robot()
    myrobot.set(0., 0., 0.)
    myrobot.set_noise(steering_noise, distance_noise, measurement_noise)
    filter = particles(myrobot.x, myrobot.y, myrobot.orientation,
                       steering_noise, distance_noise, measurement_noise)

    cte  = 0.0
    err  = 0.0
    N    = 0

    index = 0 # index into the path

    while not myrobot.check_goal(goal) and N < timeout:

        diff_cte = - cte


        # ----------------------------------------
        # compute the CTE

        # start with the present robot estimate
        estimate = filter.get_position()

        ### ENTER CODE HERE
        # some basic vector calculations
        dx = spath[index+1][0] - spath[index][0]
        dy = spath[index+1][1] - spath[index][1]
        drx = estimate[0] -  spath[index][0]
        dry = estimate[1] -  spath[index][1]
        # u is the robot estimate projected into the path segment
        u = (drx * dx + dry * dy)/(dx * dx + dy * dy)
        #the cte is the estimate projected onto the normal of the path segment
        cte = (dry * dx - drx * dy)/(dx * dx + dy * dy)
        if u > 1:
            index += 1


        # ----------------------------------------


        diff_cte += cte

        steer = - params[0] * cte - params[1] * diff_cte

        myrobot = myrobot.move(grid, steer, speed)
        filter.move(grid, steer, speed)

        Z = myrobot.sense()
        filter.sense(Z)

        if not myrobot.check_collision(grid):
            print '##### Collision ####'

        err += (cte ** 2)
        N += 1

        if printflag:
            print myrobot, cte, index, u

    return [myrobot.check_goal(goal), myrobot.num_collisions, myrobot.num_steps]


# ------------------------------------------------
#
# this is our main routine
#

def main(grid, init, goal, steering_noise, distance_noise, measurement_noise,
         weight_data, weight_smooth, p_gain, d_gain):

    path = plan(grid, init, goal)
    path.astar()
```

```
    path.smooth(weight_data, weight_smooth)
    return run(grid, goal, path.spath, [p_gain, d_gain])



# ------------------------------------------------
#
# input data and parameters
#


# grid format:
#   0 = navigable space
#   1 = occupied space

grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 1, 1, 0],
        [0, 1, 0, 1, 0, 0],
        [0, 0, 0, 1, 0, 1],
        [0, 1, 0, 1, 0, 0]]


init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1]


steering_noise    = 0.1
distance_noise    = 0.03
measurement_noise = 0.3

weight_data       = 0.1
weight_smooth     = 0.2
p_gain            = 2.0
d_gain            = 6.0


print main(grid, init, goal, steering_noise, distance_noise, measurement_noise,
           weight_data, weight_smooth, p_gain, d_gain)



def twiddle(init_params):
    n_params   = len(init_params)
    dparams    = [1.0 for row in range(n_params)]
    params     = [0.0 for row in range(n_params)]
    K = 10

    for i in range(n_params):
        params[i] = init_params[i]


    best_error = 0.0;
    for k in range(K):
        ret = main(grid, init, goal,
                   steering_noise, distance_noise, measurement_noise,
                   params[0], params[1], params[2], params[3])
        if ret[0]:
            best_error += ret[1] * 100 + ret[2]
        else:
            best_error += 99999
    best_error = float(best_error) / float(k+1)
    print best_error

    n = 0
    while sum(dparams) > 0.0000001:
        for i in range(len(params)):
            params[i] += dparams[i]
            err = 0
            for k in range(K):
                ret = main(grid, init, goal,
                           steering_noise, distance_noise, measurement_noise,
                           params[0], params[1], params[2], params[3], best_error)
                if ret[0]:
                    err += ret[1] * 100 + ret[2]
                else:
                    err += 99999
```

```
            print float(err) / float(k+1)
            if err < best_error:
                best_error = float(err) / float(k+1)
                dparams[i] *= 1.1
            else:
                params[i] -= 2.0 * dparams[i]
                err = 0
                for k in range(K):
                    ret = main(grid, init, goal,
                               steering_noise, distance_noise, measurement_noise,
                               params[0], params[1], params[2], params[3], best_error)
                    if ret[0]:
                        err += ret[1] * 100 + ret[2]
                    else:
                        err += 99999
                print float(err) / float(k+1)
                if err < best_error:
                    best_error = float(err) / float(k+1)
                    dparams[i] *= 1.1
                else:
                    params[i] += dparams[i]
                    dparams[i] *= 0.5
        n += 1
        print 'Twiddle #', n, params, ' -> ', best_error
    print ' '
    return params


#twiddle([weight_data, weight_smooth, p_gain, d_gain])
```

Fun с параметрами
Найдите минутку, чтобы играть с этими параметрами модели:

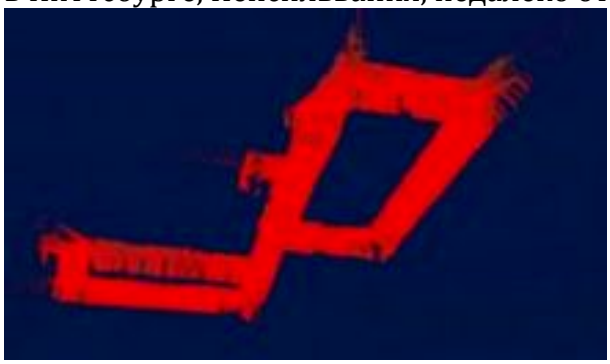
```
weight_data       = 0.1
weight_smooth     = 0.2
p_gain            = 2.0
d_gain            = 6.0
```

Измените параметры так, как вы хотите и попытайтесь найти значения, которые будут производить меньше столкновений в среднем и которые могут позволить роботу достичь цели быстрее.
В примере значения получаются путем приближенного исследования без применения twiddle. Вы можете найти применение twiddle трудным, потому что алгоритм может никогда не сойтись, и вам придется реализовать тайм-аут как-то. Но это весело играть с этими параметрами и попытаться найти лучшее решение, чем предложено. Вы можете использовать twiddle или любой другой метод, который хотите.

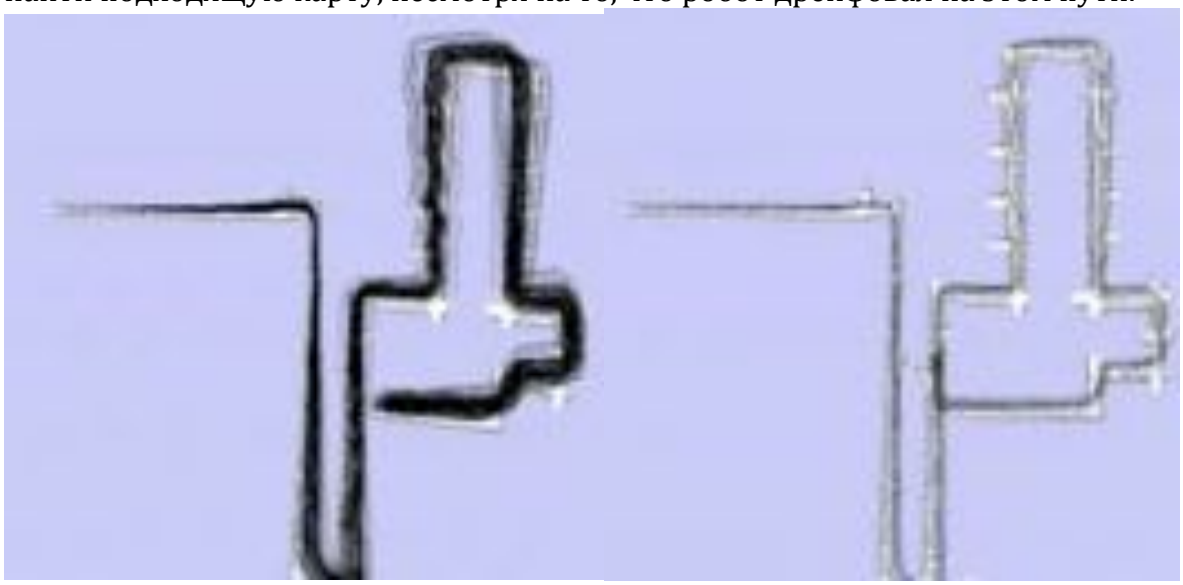
Подведение итогов
На этом мы завершаем новый материал. Даже если вы еще не сдали итоговый экзамен, тем не менее, поздравления с продвижением так далеко!
Вы многому научились, и вы должны чувствовать себя вправе программировать роботов лучше, чем раньше.
В этом классе вы изучали очень простой набор методов, которые каждый должен знать для программирование роботов.
Будь то хирургический робот, будь то робот-горничная или интеллектуальные роботы бытовые или даже летающие роботы – все эти роботы имеют задачи локализации, оценки состояния, планирование и задачи управления.
Спасибо, что вы здесь!

SLAM

SLAM является способом картографирования. Это аббревиатура от Simultaneous Localization And Mapping, одновременная локализация и картография.
Помните, в теме локализации, мы предполагали что карта является заданной. Это не всегда так, и интересно понять, как робот может создавать эти карты.
На снимках ниже вы можете увидеть 3D-карту закрытых подземных угольных шахт в Питтсбурге, Пенсильвания, недалеко от Университета Карнеги-Меллона.



Ряд различных методов для создания карты был разработан в течение последних десяти лет. Что все эти методы имеют общего – это процесс, построения модели окружающей среды, а также учет того, что робот вносит неопределенность, когда он движется.
В этом примере, когда робот совершил круг и вернулся в току, которую он проходил ранее, можно увидеть, как технология картографирования способна учесть это и найти подходящую карту, несмотря на то, что робот дрейфовал на этом пути.

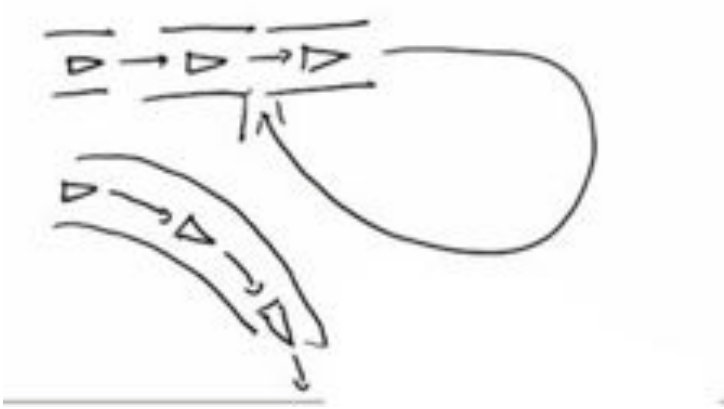


Ключевым моментом в создании карты является то, что сам робот может потерять представление, где он, в силу неопределенности своего движения. В локализации вы можете решить эту проблему, используя существующую карту, но теперь у Вас нет карты - Вы строите ее! Вот когда SLAM вступает в игру.

SLAM не от хлопанья робота (slamming a robot), это означает, одновременная локализация и картографирование.

Это огромное поле исследований. В этом блоке вы узнаете о методе Graph SLAM, который на сегодняшний день является наиболее простым для понимания.
С помощью этого метода можно свести задачу картографирования к паре интуитивно понятных операций сложения с большой матрицей и вектором.

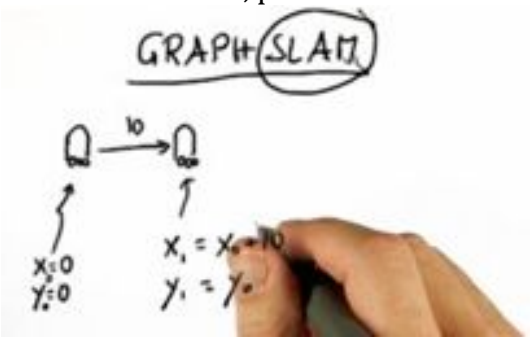
Q6-10: является ли локализация необходимой
Вот краткий тест.
При картографировании окружающей среды с помощью мобильного робота, неточность в движении робота заставляет нас также выполнять локализацию?
а. да
б. нет
Ответ на Q6-10 A6-10: является ли локализация необходимой
И ответ будет положительным. Почти во всех случаях картографирования есть неточность движения, и эта неопределенность может расти с течением времени. С этим нужно что-то делать, в противном случае карта выглядит очень плохо.
Вот пример. Предположим, робот работает в коридоре и видит своими датчикам окружающие стены. Если этот робот имеет проблемы дрейфа, из-за неточности в своем движении он считает окружающую среду иной, чем он есть на самом деле.
На изображении ниже верхняя траектория реальная, а нижняя – построенная роботом с дрейфом в движении.



Эта проблема может быть неразличима на первый взгляд, но если робот когда-либо возвращается на то же место, нет никакой возможности исправить карту. Так что хорошая техника SLAM должна быть в состоянии учесть не только тот факт, что окружающая среда является неопределенной, но и что сам робот работает неточно. Это делает реализацию SLAM трудной.

Q6-11: Graph SLAM
Graph SLAM является одним из многих методов SLAM, но его, безусловно, легче всего объяснить.
Предположим, имеем робота с начальным положением (x0 = 0, y0 = 0).
Оставляем вещи простыми, предполагая, что робот обладает идеальным компасом, так что вам не придется рассматривать направления движения.
Теперь предположим, что робот движется вправо, а именно в направлении x, на 10. В идеальном мире вы бы знали, что новым положением является (x1 = x0 + 10, y0 = y1).

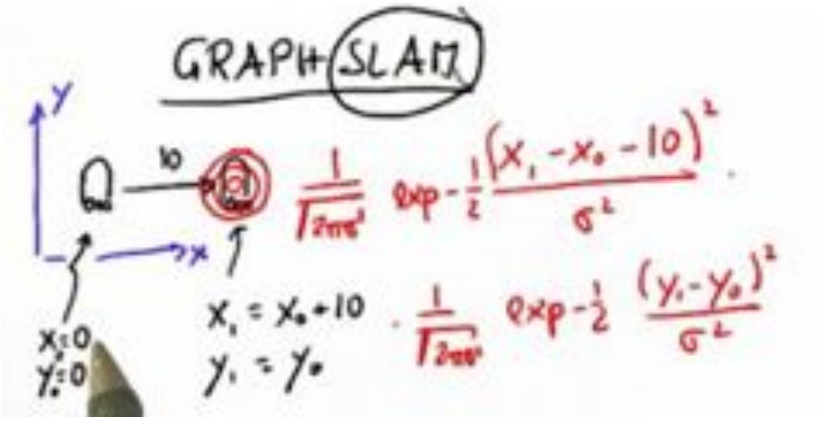
Но, как вы узнали ранее, положение является фактически неопределенным. Вместо того, чтобы принять, что робот проехал 10 единиц вправо точно, вы знаете, что фактическое местоположение является гауссовским распределением вокруг (10, 0). Вы также знаете, робота может быть где-то в другом месте.



Напомним, математику для Гауссиана. Вот как это выглядит для переменной x: вместо того, чтобы писать x1 = x0 + 10, это уравнение выражает гауссиан с пиком когда x1 и x0 + 10 одинаковы:

$$\frac{1}{\sqrt{2\pi\sigma^2}}\ exp\ -\ \frac{1}{2}\ \frac{(x_1 - x_0 - 10)^2}{\sigma^2}$$

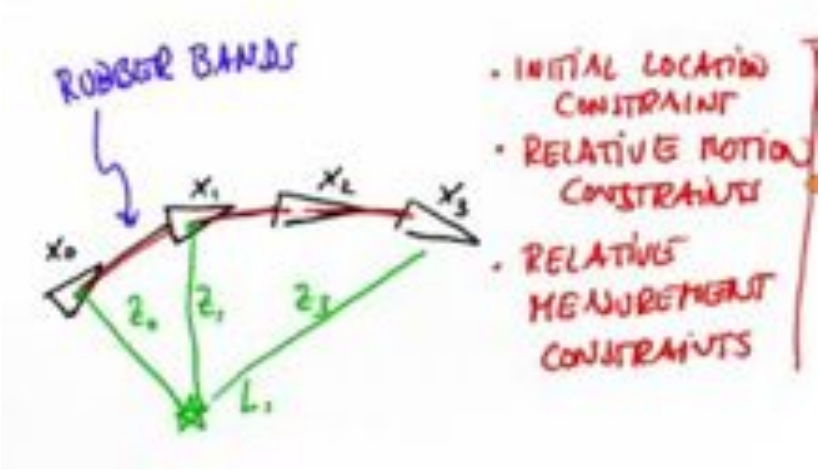
Вычитая одно из другого и подставляя результат как среднее в гауссиан, вы получаете распределение вероятности, которое связывает x1 и x0 . Вы можете сделать то же самое для y. Т.к. нет никакого движения, y0 и y1 должны быть настолько близки, насколько это возможно. Произведение этих двух гауссианов представляет собой ограничение. Вероятность положение (x1, y1) должно быть максимальным при условии, что начальная позиция (x0=0, y0=0).



Graph SLAM определяет вероятности, используя последовательность таких ограничений.
Рассмотрим робота движущегося в некотором пространстве, где каждое местоположение характеризуется вектором $x_0, x_1, x_2, \ldots$. Graph SLAM собирает начальное расположение и много относительных ограничений, которые связывают каждое положение робота с предыдущим, которые называются относительными ограниченимия движения робота (relative robot motion constraints).
Эти ограничения, как резиновые жгуты. В ожидании резиновые жгуты – это точно движение робота, но в действительности им, возможно, придется деформироваться немного, чтобы сделать карту более правильной.

На местности вы можете найти особые точки (ориентиры, landmarks). Landmarks видны из какого-либо положения робота с некоторым относительным измерением. Все эти измерения также являются относительными ограничениями, очень

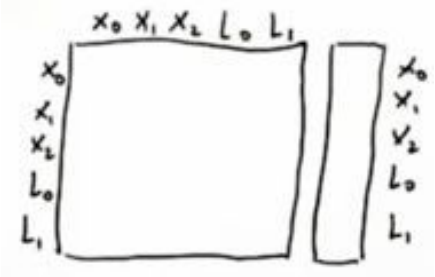
похожими на те, что рассмотрены раньше. Опять они описываются Гауссианами и накладывают относительные ограничения измерения. Такое ограничение вводится каждый раз, когда робот видит ориентир (landmark). Итак, Graph SLAM собирает такие ограничения (что безумно легко) и расслабляет набор резиновых жгутов, так чтобы найти наиболее вероятную конфигурацию пути робота с учетом расположения ориентиров. Это процесс картографирования.



Предположим, у вас есть 6 положений робота (следовательно есть 5 движений) и 8 измерений расстояний до ориентиров. Как много ограничений у вас есть?
Ответ на Q6-11 A6-11: Graph SLAM
И ответ 14. Существует одно ограничение начального положение, 5 движений (между 6 положениями) и 8 ограничений измерений относительно ориентиров.

Дополнительный материал: Для более глубокого понимания Graph SLAM вы можете прочитать эту статью - The GraphSLAM Algorithm with Applications to Large-Scale Mapping of Urban Structure by Sebastian Thrun and Michael Montemerlo, published in THE INTERNATIONAL JOURNAL OF ROBOTICS RESEARCH / May–June 2006.

B6-12: Реализация ограничений
Для реализации Graph SLAM вводятся матрица Omega и вектор Xi. Матрица Omega квадратная и помечена всеми положениями и всеми ориентирами, предполагая, что ориентиры различимы.

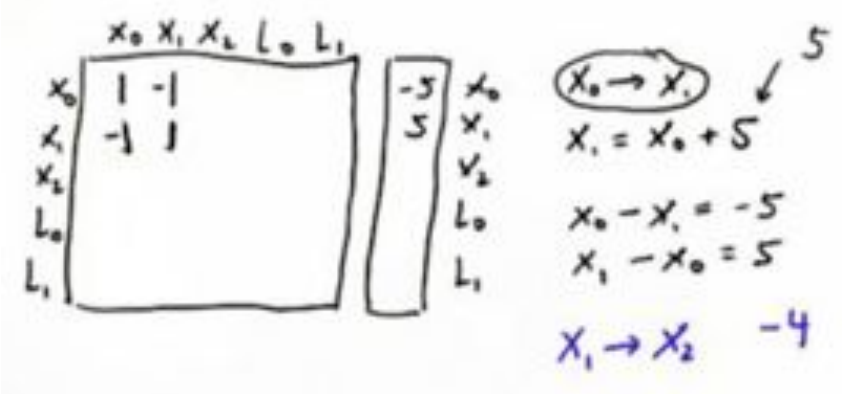


Каждый раз, когда вы делаете наблюдения, например между двумя положениями, они становятся небольшими дополнениями локально в четырех элементах матрицы, определенными этими положениями.

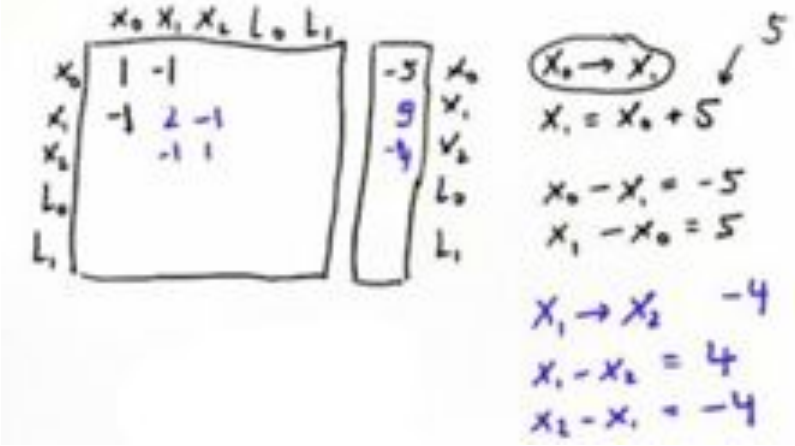
Например, робот перемещается от x0 в x1, и поэтому он считает, что $x_1 = x_0 + 5$. Это вводится в матрицу двумя способами:
1. Уравнение изменяется таким образом, что обе x были сгруппированы вместе и член соответствующий начальному положению имеел положительный знак: X0 - X1 = -5. Затем коэффициенты +1 и -1 для x0 и x1 соответственно добавляются в матрицу, которая изначально содержит все нули. Ограничение -5 добавляется в соответствующий элемент в векторе.

2. Вы делаете то же для уравнения с положительным знаком для второго положения: X1 - X0 = 5.

Иными словами, ограничение движения, которое связывает x0 и x1 движением на 5, модифицирует матрицу постепенно, добавляя значения для всех элементов, которые соответствуют x0 и x1. Ограничение написано дважды, и в обоих случаях вы должны убедиться, что диагональные элементы положительны, а соответствующие недиагональные элементы в том же ряду отрицательны.

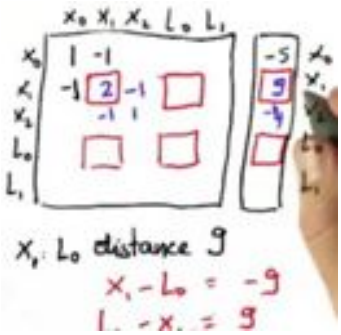


Предположим, что роботов движется от x1 к x2 и движение -4, т.е. он движется в противоположном направлении.
Каковы будут новые значения матрицы и вектора? Подсказка: это движение влияет только на значения, которые находятся в подматрице, образованной x1 и x2 в Омеге и соответствующие записи в Кси. Кроме того, помните, что вы должны сложить значения.
Ответ 6-12: Реализация ограничений

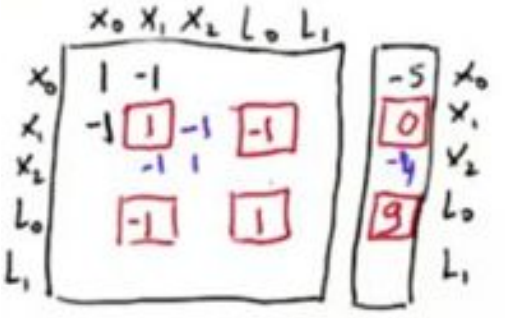


Q6-13: Добавление ориентира
Приготовьтесь к другому заданию! Предположим, что в робот, находясь в x_1, видит ориентир L_0 на расстоянии 9. Это означает накладывание относительного ограничения (связи) между x1 и L0. Матричные элементы для таких ограничений не образуют подматрицы; они разбросаны друг от друга. Измените значения в матрице и векторе чтобы указать, что положение ориентира L_0 больше x1 на 9.

Ответ на Q6-13 A6-13: Добавление ориентира
Преобразуя уравнения точно так, как и раньше, можно получить следующие уравнения: x1 – L0 = -9 и L0 – x1 = 9.

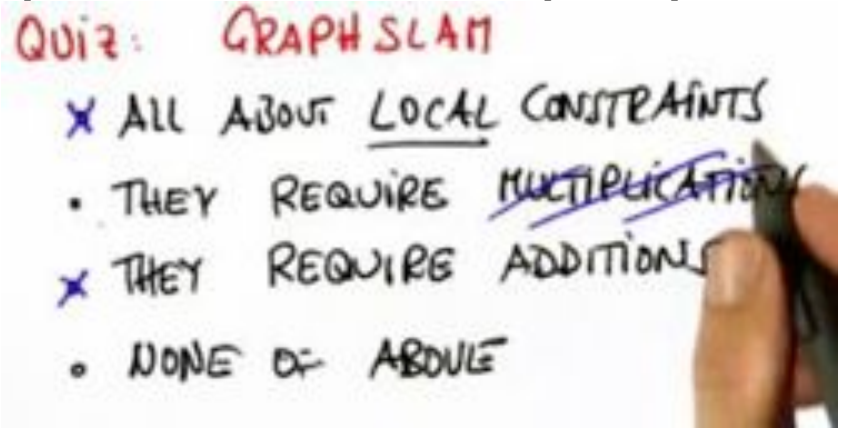


Прибавим +1 и -1 к диагональному и вне- диагональному элементам матрицы соответственно и добавим -9 и +9 и к элементам вектора.
Правильный ответ:



B6-14: SLAM Соберем все, что вы узнали о Graph SLAM до сих пор. Пожалуйста, ответьте на все вопросы, которые относятся к Graph SLAM:
a. Graph SLAM оперирует локальными ограничениями
б. Graph SLAM требует умножения
c. Graph SLAM требует сложений
d. Ни один из указанных

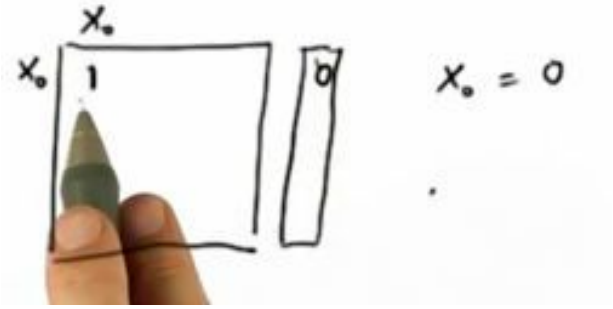
Ответ на Q6-14 A6-14: SLAM
Правильными ответами являются первый и третий.



Graph SLAM действительно имеет дело с локальными ограничениями (связями). Вы видите, что каждое движение, потому что любое движение связывает 2 положения робота, и любое измерение связывает 2 положения робота и ориентира.
Эти ограничения вносятся в соответствующие матрицу и вектор операцией сложения. Умножение здесь неправильный вариант. Это очень просто!

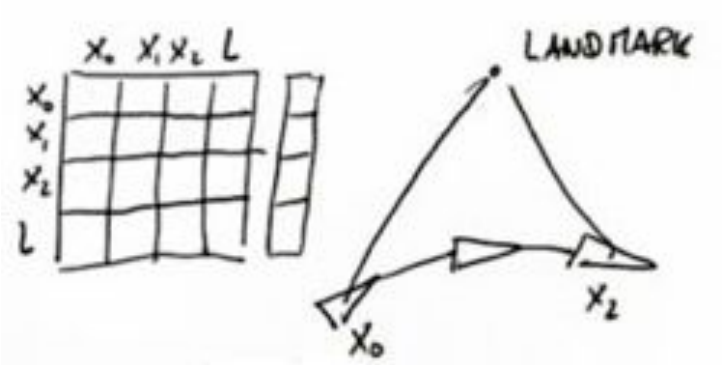
B6-15: Matrix Модификация

Итак, вы узнали, как учесть ограничения (связи) движения и измерения в Graph SLAM. Есть еще одна вещь для рассмотрения: исходное положение. Как вы можете учесть это ограничения в Graph SLAM? Предположим, что ваша начальная координата, в одномерном мире $x_0 = 0$. На рисунке ниже показан способ учета этого в матрице и векторе Graph SLAM.



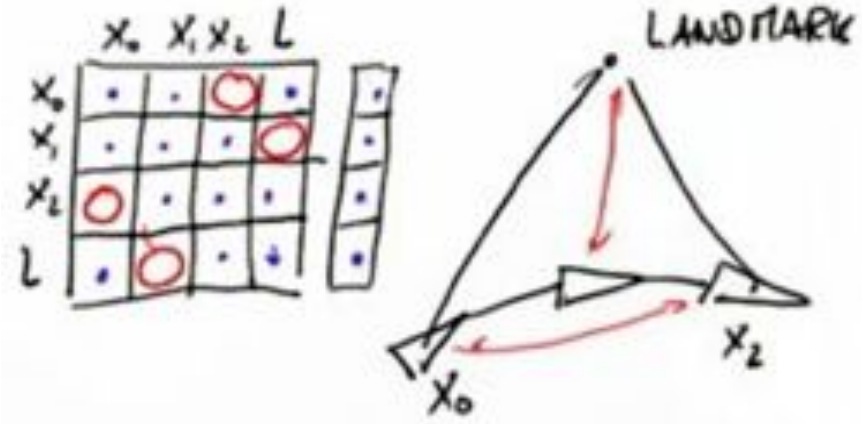
Вы просто добавляете 1 к элементу матрицы, соответствующему x_0, а также прибавляете начальную координату к соответствующему элементу вектора.

Для следующего вопроса, рассмотрим робот, который перемещается от x_0 к x_2. Робот видит ориентир L, когда он находится в x_0 и x_2. Но он не в состоянии увидеть его из x1. Это показано на рисунке ниже.



Отметьте элементы, которые будут изменены в матрице и векторе без расчета реальных значений.

Ответ на Q6-15A6-15: Matrix Модификация
И ответ отмечен синими точками на рисунке ниже!



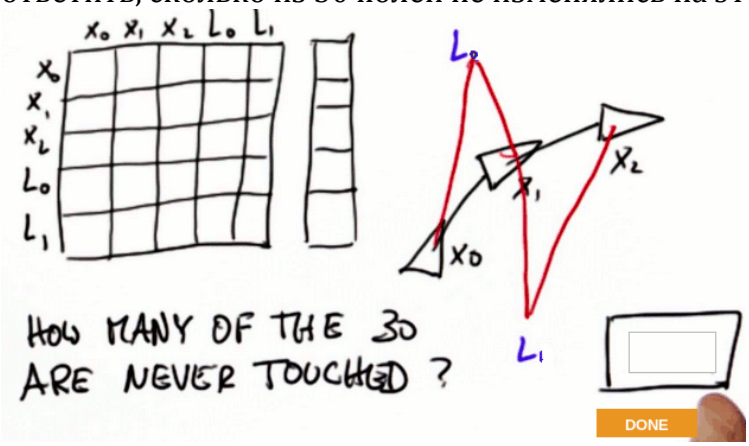
Обратите внимание, что почти каждый элемент матрицы и вектора модифицированы. Только поля, равные нулю (отмечены красным цветом), соответствуют связи движения между x0 и x1 и связи измерения между x1 и L, как показано красными стрелками.
Ниже приведен список полей, которые меняет каждое ограничение (связь):

Initial pose:	matrix[0][0], vector[0]
Movement $x_0 \rightarrow x_1$:	matrix[0][0], matrix[0][1], matrix[1][0], matrix[1][1], vector[0], vector[1]
Movement $x_1 \rightarrow x_2$:	matrix[1][1], matrix[1][2], matrix[2][1], matrix[2][2], vector[1], vector[2]
Measurement $x_0 \rightarrow L$:	matrix[0][0], matrix[0][3], matrix[3][0], matrix[3][3], vector[0], vector[3]
Measurement $x_2 \rightarrow L$:	matrix[2][2], matrix[2][3], matrix[3][2], matrix[3][3], vector[2], vector[3]

Следует отметить, что matrix[0][2], matrix[2][0], matrix[1][3] и matrix[3][1] не находятся в этом списке. Это означает, что нет никаких прямых связей между x_0 и x_2, так как нет информации о прямом движении между ними, а также нет прямой связи между x1 и L, так как отсутствует сенсорная информация, которая связала бы x1 и L

B6-16: Нетронутые поля

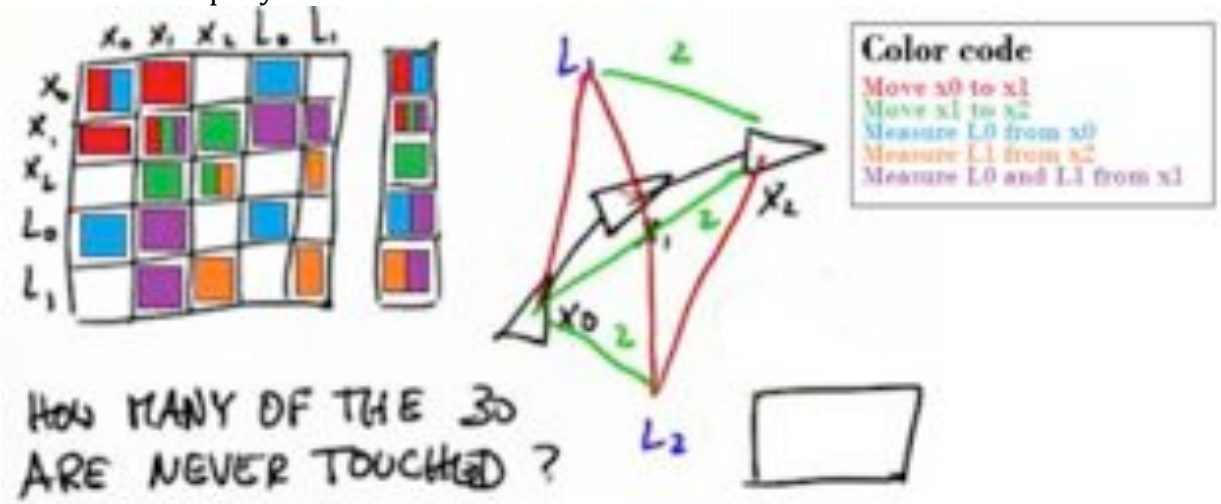
Для этой задачи рассматривается такая же ситуация как и раньше, за исключением того, что на этот раз робот видит два ориентира. Ориентир L0 виден из положений x0 и x1, в то время как ориентир L1 видно из позиции x1 и x2. Можете ли вы ответить, сколько из 30 полей не изменялись на этот раз?

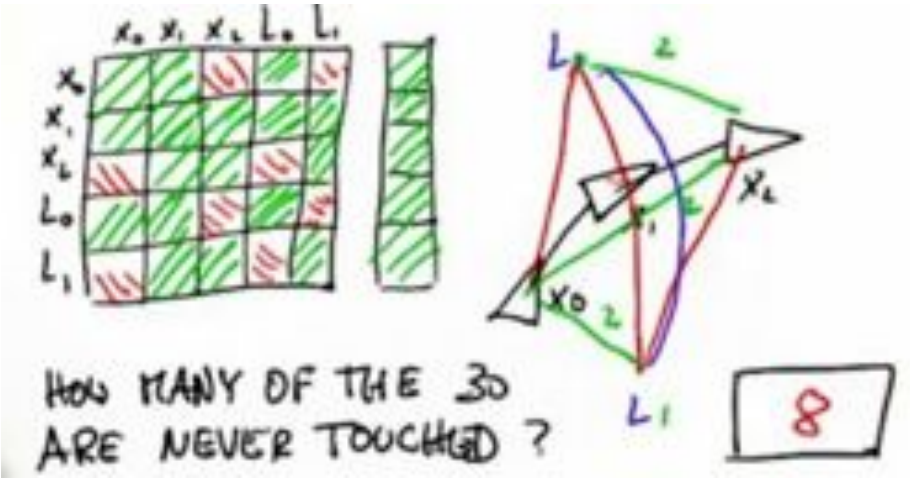


Ответ на Q6-16 A6-16: Нетронутые поля

И ответ ... 6? Нет, ответ 8. Так что если вы получили правильный ответ, поздравляю!

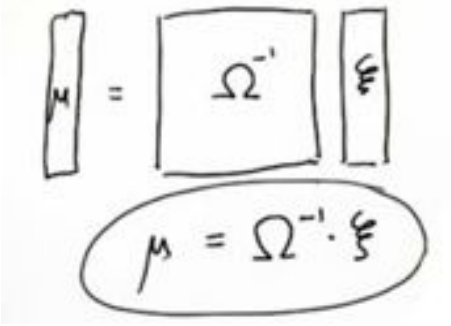
Пояснения на рисунке



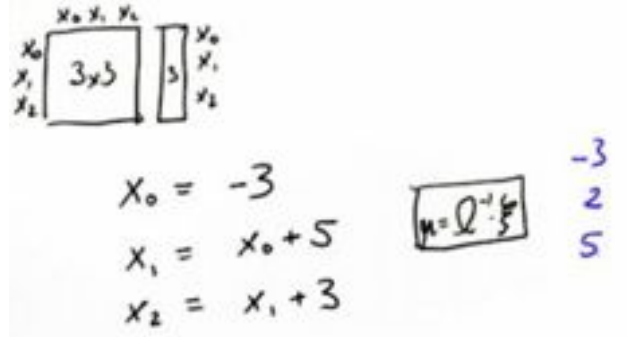


Omega и Xi B6-17: Вы можете использовать матрицу Omega и вектор Xi, которые вы строили, чтобы легко вычислить вектор mu, который будет содержать оценку положения робота (это часть SLAM касающаяся локализации), а также оценку расположения ориентиров (это часть SLAM касающаяся картографирования).

Для этого расчета, необходимо только инвертировать матрицу Омега, что вы можете сделать используя функцию inverse() класса матрицы, и умножить это на вектор Xi. Результатом будет вектор mu! Этот расчет графически показан на рисунке ниже.



Для следующего задания, вы должны построить матрицу Omega, вектор Xi с использованием initial_position и заданных значений move1 и move2. Затем вы должны использовать формулу представленную выше для вычислений и вернуть mu.
На рисунке ниже показан пример, где initial_position = -3, move1 = 5 и move2 = 3. Соответствующий вектор mu показан синим цветом.

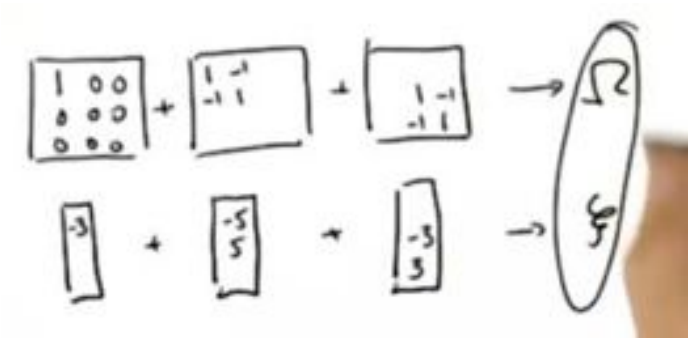


Ответ на Q6-17

Вот что нужно сделать:

```
    Omega = matrix( [[1,0,0], [0,0,0], [0,0,0]] )
    Xi = matrix( [[initial_pos],[0],[0]] )
```

```
    Omega += matrix( [[1,-1,0],[-1,1,0],[0,0,0]] )
    Xi += matrix( [[-move1],[move1],[0]] )
    Omega += matrix( [[0,0,0],[0,1,-1],[0,-1,1]] )
    Xi += matrix( [[0],[-move2],[move2]] )

    mu = Omega.inverse()*Xi
```



Полный код программы:

```
# -----------
# User Instructions
#
# Write a function, doit, that takes as its input an
# initial robot position, move1, and move2. This
# function should compute the Omega and Xi matrices
# discussed in lecture and should RETURN the mu vector
# (which is the product of Omega.inverse() and Xi).
#
# Please enter your code at the bottom.



from math import *
import random


#========================================================================
#
# SLAM in a rectolinear world (we avoid non-linearities)
#
#
#========================================================================


# ----------------------------------------------
#
# this is the matrix class
# we use it because it makes it easier to collect constraints in GraphSLAM
# and to calculate solutions (albeit inefficiently)
#

class matrix:

    # implements basic operations of a matrix class

    # ------------
    #
    # initialization - can be called with an initial matrix
    #

    def __init__(self, value = [[]]):
        self.value = value
        self.dimx  = len(value)
        self.dimy  = len(value[0])
        if value == [[]]:
            self.dimx = 0

    # ------------
    #
    # makes matrix of a certain size and sets each element to zero
```

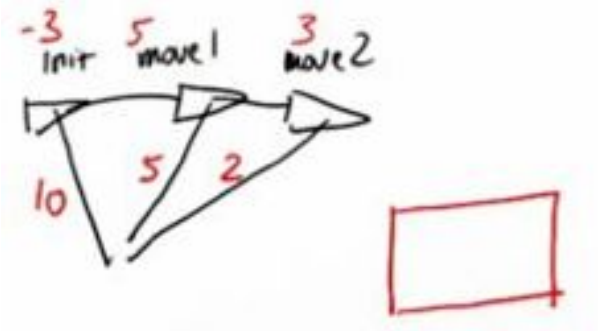
```
    #

    def zero(self, dimx, dimy = 0):
        if dimy == 0:
            dimy = dimx
        # check if valid dimensions
        if dimx < 1 or dimy < 1:
            raise ValueError, "Invalid size of matrix"
        else:
            self.dimx  = dimx
            self.dimy  = dimy
            self.value = [[0.0 for row in range(dimy)] for col in range(dimx)]

    # ------------
    #
    # makes matrix of a certain (square) size and turns matrix into identity matrix
    #


    def identity(self, dim):
        # check if valid dimension
        if dim < 1:
            raise ValueError, "Invalid size of matrix"
        else:
            self.dimx  = dim
            self.dimy  = dim
            self.value = [[0.0 for row in range(dim)] for col in range(dim)]
            for i in range(dim):
                self.value[i][i] = 1.0
    # ------------
    #
    # prints out values of matrix
    #


    def show(self, txt = ''):
        for i in range(len(self.value)):
            print txt + '['+ ', '.join('%.3f'%x for x in self.value[i]) + ']'
        print ' '

    # ------------
    #
    # defines elmement-wise matrix addition. Both matrices must be of equal dimensions
    #


    def __add__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError, "Matrices must be of equal dimension to add"
        else:
            # add if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] + other.value[i][j]
            return res

    # ------------
    #
    # defines elmement-wise matrix subtraction. Both matrices must be of equal dimensions
    #

    def __sub__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError, "Matrices must be of equal dimension to subtract"
        else:
            # subtract if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] - other.value[i][j]
            return res

    # ------------
```

```
    #
    # defines multiplication. Both matrices must be of fitting dimensions
    #


    def __mul__(self, other):
        # check if correct dimensions
        if self.dimy != other.dimx:
            raise ValueError, "Matrices must be m*n and n*p to multiply"
        else:
            # multiply if correct dimensions
            res = matrix()
            res.zero(self.dimx, other.dimy)
            for i in range(self.dimx):
                for j in range(other.dimy):
                    for k in range(self.dimy):
                        res.value[i][j] += self.value[i][k] * other.value[k][j]
        return res


    # ------------
    #
    # returns a matrix transpose
    #


    def transpose(self):
        # compute transpose
        res = matrix()
        res.zero(self.dimy, self.dimx)
        for i in range(self.dimx):
            for j in range(self.dimy):
                res.value[j][i] = self.value[i][j]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[ 1,  2,  3,  4,  5],
    #                   [ 6,  7,  8,  9, 10],
    #                   [11, 12, 13, 14, 15]])
    #
    #       l.take([0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 3, 4],
    #        [11, 13, 14]]
    #
    #
    # take is used to remove rows and columns from existing matrices
    # list1/list2 define a sequence of rows/columns that shall be taken
    # is no list2 is provided, then list2 is set to list1 (good for symmetric matrices)
    #

    def take(self, list1, list2 = []):
        if list2 == []:
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError, "list invalid in take()"

        res = matrix()
        res.zero(len(list1), len(list2))
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[i][j] = self.value[list1[i]][list2[j]]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[1, 2, 3],
    #                   [4, 5, 6]])
    #
```

```
#       l.expand(3, 5, [0, 2], [0, 2, 3])
#
# results in:
#
#       [[1, 0, 2, 3, 0],
#        [0, 0, 0, 0, 0],
#        [4, 0, 5, 6, 0]]
#
# expand is used to introduce new rows and columns into an existing matrix
# list1/list2 are the new indexes of row/columns in which the matrix
# elements are being mapped. Elements for rows and columns
# that are not listed in list1/list2
# will be initialized by 0.0.
#

def expand(self, dimx, dimy, list1, list2 = []):
    if list2 == []:
        list2 = list1
    if len(list1) > self.dimx or len(list2) > self.dimy:
        raise ValueError, "list invalid in expand()"

    res = matrix()
    res.zero(dimx, dimy)
    for i in range(len(list1)):
        for j in range(len(list2)):
            res.value[list1[i]][list2[j]] = self.value[i][j]
    return res

# ------------
#
# Computes the upper triangular Cholesky factorization of
# a positive definite matrix.
# This code is based on http://adorio-research.org/wordpress/?p=4560

def Cholesky(self, ztol= 1.0e-5):
    res = matrix()
    res.zero(self.dimx, self.dimx)

    for i in range(self.dimx):
        S = sum([(res.value[k][i])**2 for k in range(i)])
        d = self.value[i][i] - S
        if abs(d) < ztol:
            res.value[i][i] = 0.0
        else:
            if d < 0.0:
                raise ValueError, "Matrix not positive-definite"
            res.value[i][i] = sqrt(d)
        for j in range(i+1, self.dimx):
            S = sum([res.value[k][i] * res.value[k][j] for k in range(i)])
            if abs(S) < ztol:
                S = 0.0
            res.value[i][j] = (self.value[i][j] - S)/res.value[i][i]
    return res

# ------------
#
# Computes inverse of matrix given its Cholesky upper Triangular
# decomposition of matrix.
# This code is based on http://adorio-research.org/wordpress/?p=4560

def CholeskyInverse(self):
    res = matrix()
    res.zero(self.dimx, self.dimx)

    # Backward step for inverse.
    for j in reversed(range(self.dimx)):
        tjj = self.value[j][j]
        S = sum([self.value[j][k]*res.value[j][k] for k in range(j+1, self.dimx)])
        res.value[j][j] = 1.0/ tjj**2 - S/ tjj
        for i in reversed(range(j)):
            res.value[j][i] = res.value[i][j] = \
                -sum([self.value[i][k]*res.value[k][j] for k in \
                      range(i+1,self.dimx)])/self.value[i][i]
    return res

# ------------
#
# comutes and returns the inverse of a square matrix
```

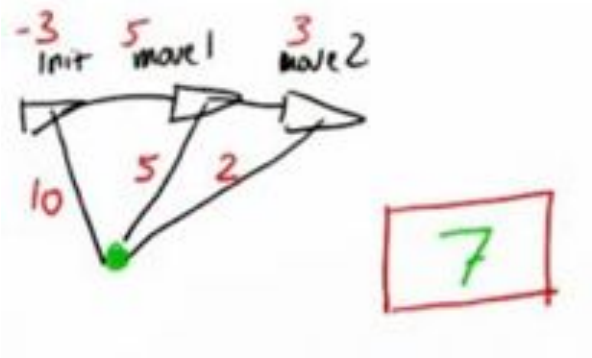
```
    #


    def inverse(self):
        aux = self.Cholesky()
        res = aux.CholeskyInverse()
        return res

    # ------------
    #
    # prints matrix (needs work!)
    #


    def __repr__(self):
        return repr(self.value)



# ########################################################################
# ########################################################################
# ########################################################################


"""
    For the following example, you would call doit(-3, 5, 3):
    3 robot positions
    initially: -3
    moves by 5
    moves by 3



    which should return a mu of:
    [[-3.0],
    [2.0],
    [5.0]]
    """
def doit(initial_pos, move1, move2):
    #
    #
    # Add your code here.
    Omega = matrix( [[1,0,0], [0,0,0], [0,0,0]] )
    Xi = matrix( [[initial_pos],[0],[0]] )
    Omega += matrix( [[1,-1,0],[-1,1,0],[0,0,0]] )
    Xi += matrix( [[-move1],[move1],[0]] )
    Omega += matrix( [[0,0,0],[0,1,-1],[0,-1,1]] )
    Xi += matrix( [[0],[-move2],[move2]] )

    mu = Omega.inverse()*Xi
    #
    #
    return mu

doit(-3, 5, 3)
```

B6-18: Положения ориентиров Landmark
В этом задании, робот начинает в первоначальном положении -3, где он видит ориентир на расстоянии 10. Затем он перемещается на 5 и видит ориентир на расстоянии 5. Наконец, он перемещается на 3 и измеряет расстояние до ориентира 2.



Какова лучшая оценка положения ориентира?
Мы рассматриваем одномерный мир в этом примере.

Ответ на Q6-18 A6-18: Позиция Landmark
И ответ 7. Вы можете проверить этот результат напрямую, потому что пока нет никаких погрешностей движения или измерения.
В первом положении у вас может возникнуть соблазн сказать, что в действительности существует два возможных положения ориентира, который отстоят на 10 единиц от x0 = -3, а именно -13 и 7. Однако, когда вы перейдете к x1 и x2, единственный вариант, который согласуется с последующими измерениями – 7.



Расчеты, используемые для получения этого вывода приведены в таблице ниже. Они же записаны в Omega и Кси .

$x_0 = -3$	$x_1 = x_0 + move1 = -3 + 5 = 2$	$x_2 = x_1 + move2 = 2 + 3 = 5$
$d_0 = \lvert x_l - x_0 \rvert = 10$	$d_1 = \lvert x_l - x_1 \rvert = 5$	$d_2 = \lvert x_l - x_2 \rvert = 2$

B6-19: Expand
Теперь вы будете расширять ваши матрицу и вектор для размещения ориентиров. В частности, вы будете использовать функцию expand. Например, вы можете dspdfnm Omega.expand для преобразования матрицы 3x3 в матрицу 4x4 которая включает в себя ориентир.

Ответ на Q6-19
A6-19: Развернуть
Сначала используйте команду expand чтобы увеличить размеры Omega и Xi

```
    Omega = Omega.expand(4, 4, [0,1,2], [0,1,2])
    Xi = Xi.expand(4, 1, [0, 1, 2], [0])
```

Теперь добавьте ограничения измерений:

```
    Omega += matrix([[1., 0., 0., -1.],[0., 0., 0., 0.],[0., 0., 0., 0.],[- 1., 0., 0., 1.]])
    Xi += matrix([[-Z0], [0.], [0.], [Z0]])
    Omega += matrix([[0., 0., 0., 0.],[0., 1., 0., -1.],[0., 0., 0., 0.],[0.,-1., 0., 1.]])
    Xi += matrix([[0.], [-Z1], [0.], [Z1]])
    Omega += matrix([[0., 0., 0., 0.],[0., 0., 0., 0.],[0., 0., 1., -1.],[0., 0., -1., 1.]])
    Xi += matrix([[0.], [0.], [-Z2], [Z2]])
```

Приводим полный код функции doit, остальное (определение класса матрицы) - можно взять из предыдущей программы:

```
def doit(initial_pos, move1, move2, Z0, Z1, Z2):
    Omega = matrix([[1.0, 0.0, 0.0],
                    [0.0, 0.0, 0.0],
                    [0.0, 0.0, 0.0]])
    Xi = matrix([[initial_pos],
                 [0.0],
                 [0.0]])
```

```
    Omega += matrix([[1.0, -1.0, 0.0],
                     [-1.0, 1.0, 0.0],
                     [0.0, 0.0, 0.0]])
    Xi += matrix([[-move1],
                  [move1],
                  [0.0]])

    Omega += matrix([[0.0, 0.0, 0.0],
                     [0.0, 1.0, -1.0],
                     [0.0, -1.0, 1.0]])
    Xi += matrix([[0.0],
                  [-move2],
                  [move2]])

    #
    #
    # Add your code here.
    Omega = Omega.expand(4, 4, [0,1,2], [0,1,2])
    Xi = Xi.expand(4, 1, [0, 1, 2], [0])
    Omega += matrix([[1., 0., 0., -1.],[0., 0., 0., 0.],[0., 0., 0., 0.],[- 1., 0.,
0., 1.]])
    Xi += matrix([[-Z0], [0.], [0.], [Z0]])
    Omega += matrix([[0., 0., 0., 0.],[0., 1., 0., -1.],[0., 0., 0., 0.],[0.,-1., 0.,
1.]])
    Xi += matrix([[0.], [-Z1], [0.], [Z1]])
    Omega += matrix([[0., 0., 0., 0.],[0., 0., 0., 0.],[0., 0., 1., -1.],[0., 0., -1.,
1.]])
    Xi += matrix([[0.], [0.], [-Z2], [Z2]])
    #
    #

    Omega.show('Omega: ')
    Xi.show('Xi:    ')
    mu = Omega.inverse() * Xi
    mu.show('Mu:    ')

    return mu

doit(-3, 5, 3, 10, 5, 2)
```

Омега и Кси до expand:

```
Omega: [2.000, -1.000, 0.000]
Omega: [-1.000, 2.000, -1.000]
Omega: [0.000, -1.000, 1.000]

Xi:    [-8.000]
Xi:    [2.000]
Xi:    [3.000]
```

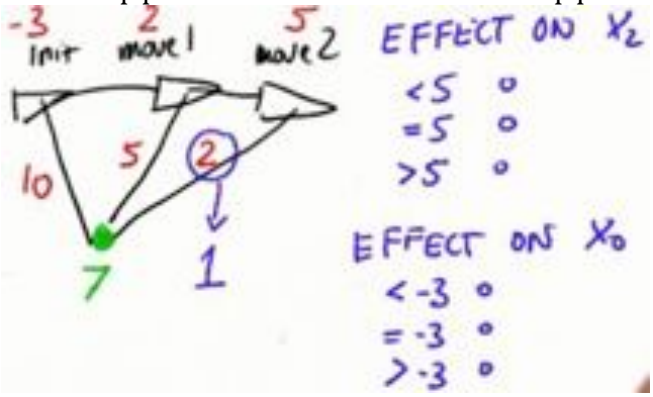
И после:

```
Omega: [3.000, -1.000, 0.000, -1.000]
Omega: [-1.000, 3.000, -1.000, -1.000]
Omega: [0.000, -1.000, 2.000, -1.000]
Omega: [-1.000, -1.000, -1.000, 3.000]

Xi:    [-18.000]
Xi:    [-3.000]
Xi:    [1.000]
Xi:    [17.000]
```

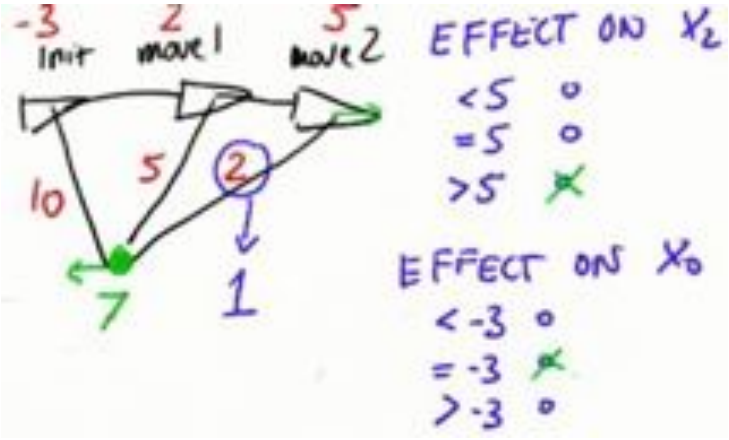
Вектор mu:

```
Mu:     [-3.000]
Mu:     [2.000]
Mu:     [5.000]
Mu:     [7.000]
```

B6-20: Введение шума
Рассмотрим движение робота снова. Предположим, вы изменили последнее измерение от 2 до 1.
Какой эффект от этого на x2? Какой эффект на x0? Подсказка: попробуйте это в коде!

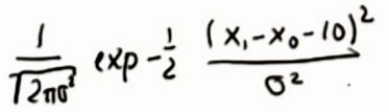


Ответ на Q6-20 A6-20: Введение шума
До изменений, результат был [-3, 2, 5, 7] . После смены вы получите [-3, 2,125, 5,5, 6,875]

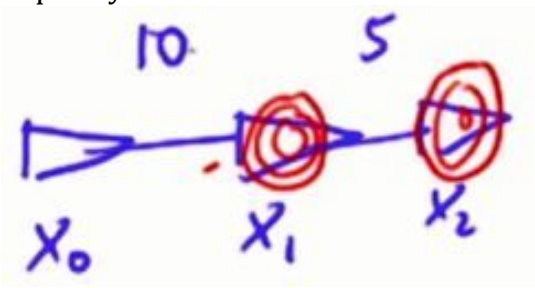


Что это значит?
Исходное положение не изменится. Ничто не изменит тот факт, что эта координата = -3. Изменятся две другие координаты, но не начальная.
Вы можете понять влияние изменения на x2 представляя резиновый жгут. Изменяя натяжение жгута до 1, вы ожидаете что положения x2 и ориентира переместятся ближе друг к другу.

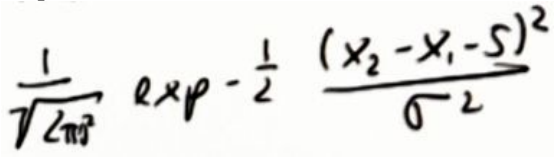
B6-21: Уверенный Измерения
Вот представление о том, почему это работает:
Предположим, у вас есть две позиции робота x0 и x1. Вторая позиция сдвинута на 10 вправо, с гауссовым шумом. Неопределенность выглядит следующим образом:



Здесь есть постоянная, экспоненты и выражение, которое будет сведено к минимуму, когда $(x_1 - x_0)$ = 10, хотя оно может отклоняться от ровно 10. Гауссиан максимален, когда это уравнение выполняется.
Теперь модель второго движения, с еще большем гауссианом по отношению к первому.



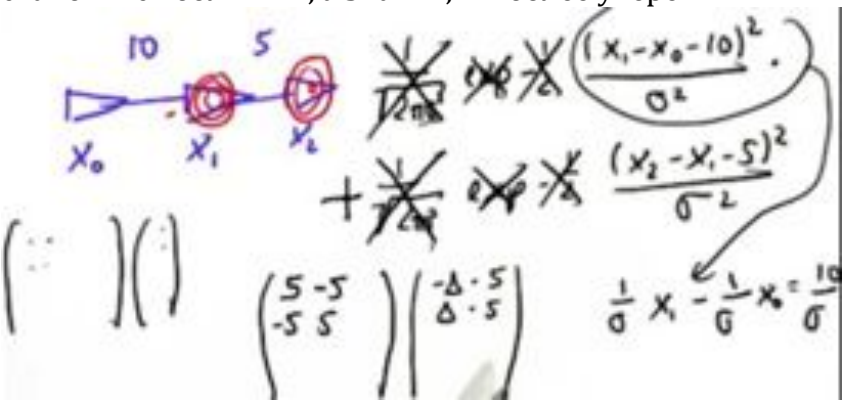
Ограничение конечно очень похоже.



Суммарная вероятность является произведением двух. Если вы хотите увеличить произведение, вот несколько трюков. Постоянная часть не имеет никакого отношения к максимизации, так что вы можете убрать ее. Вы также можете отказаться от экспоненты, если вы запишите сумму в ее показателе и будете рассматривать далее только ее. И теперь вы можете также убрать -1/2 внутри того, что было внутри экспоненты.
В итоге у вас получится уравнение.

$$x_1/\sigma - x_0/\sigma = 10/\sigma$$

Член $1/\sigma$ представляет вашу уверенность в точности. Для небольшого σ, $1/\sigma$ становится большим, а значит, вы более уверены.



Таким образом, для более точного измерения мы должны умножить значения в нашей матрице на некоторое число (например, 5, вместо 1 ранее).
Для следующего задания измените код так, как будто последнее измерение имеет очень высокое доверие с коэффициентом 5. Вы должны получить [-3, 2,179, 5,714, 6,821] в качестве ответа. Вы видите в этом результате, что разница между последней точкой и ориентиром очень близка к измеренной разности 1,0, потому что у вас есть относительно высокое доверие по сравнению с другими измерениями и движениями.

Ответ на Q6-21

Все, что нужно было сделать, это умножить все элементы прибавочной матрицы и вектора для второго положения:

```
    Omega += matrix([[0.0, 0.0, 0.0, 0.0],
                     [0.0, 0.0, 0.0, 0.0],
                     [0.0, 0.0, 5.0, -5.0],
                     [0.0, 0.0, -5.0, 5.0]])
    Xi    += matrix([[0.0],
                     [0.0],
                     [-Z2*5],
                     [Z2*5]])
```

Реализация SLAM
B6-22: Реализация SLAM
Каждый раз, когда есть ограничения (начальное положение, движение или измерения) возьмите его и добавьте в Omega и XI. Кроме того, умножьте на коэффициент $1/\sigma$ представляющий его достоверность.
Просто вычислите $\mu = \Omega^{-1}\xi$ и получите в результате путь и карту. Amazing!

В этом задании, у вас есть обобщенная среда со следующими параметрами:

```
num_landmarks      = 5        # number of landmarks
N                  = 20       # time steps
world_size         = 100.0    # size of world
measurement_range  = 50.0     # range at which we can sense landmarks
motion_noise       = 2.0      # noise in robot motion
measurement_noise  = 2.0      # noise in the measurements
distance           = 20.0     # distance by which robot (intends to)
move each iteratation
```

В коде есть функция make_data. Это функция, которая, учитывая ваши параметры окружающей среды, возвращает последовательность движений и последовательность измерений.
Вы будете писать функцию slam, которая принимает эти данные и некоторые параметры среды и возвращает последовательность пути робота и оценочные положения ориентиров.
Отметим, что существует начальное ограничение, которое помещает робота в [50., 50] - центр мира.

Ответ на Q6-22

Полный код программы:

```
# ------------
# User Instructions
#
# In this problem you will implement SLAM in a 2 dimensional
# world. Please define a function, slam, which takes five
# parameters as input and returns the vector mu. This vector
# should have x, y coordinates interlaced, so for example,
# if there were 2 poses and 2 landmarks, mu would look like:
#
#  mu =  matrix([[Px0],
#                [Py0],
#                [Px1],
#                [Py1],
#                [Lx0],
#                [Ly0],
#                [Lx1],
```

```
#               [Ly1]])
#
# data - This is the data that is generated with the included
#        make_data function. You can also use test_data to
#        make sure your function gives the correct result.
#
# N -    The number of time steps.
#
# num_landmarks - The number of landmarks.
#
# motion_noise - The noise associated with motion. The update
#                strength for motion should be 1.0 / motion_noise.
#
# measurement_noise - The noise associated with measurement.
#                     The update strength for measurement should be
#                     1.0 / measurement_noise.
#
#
# Enter your code at line 509

# --------------
# Testing
#
# Uncomment the test cases at the bottom of this document.
# Your output should be identical to the given results.


from math import *
import random


#=============================================================
#
# SLAM in a rectolinear world (we avoid non-linearities)
#
#
#=============================================================


# ----------------------------------------------
#
# this is the matrix class
# we use it because it makes it easier to collect constraints in GraphSLAM
# and to calculate solutions (albeit inefficiently)
#

class matrix:

    # implements basic operations of a matrix class

    # ------------
    #
    # initialization - can be called with an initial matrix
    #

    def __init__(self, value = [[]]):
        self.value = value
        self.dimx  = len(value)
        self.dimy  = len(value[0])
        if value == [[]]:
            self.dimx = 0

    # ------------
    #
    # makes matrix of a certain size and sets each element to zero
    #

    def zero(self, dimx, dimy):
        if dimy == 0:
            dimy = dimx
        # check if valid dimensions
        if dimx < 1 or dimy < 1:
            raise ValueError, "Invalid size of matrix"
        else:
            self.dimx  = dimx
            self.dimy  = dimy
            self.value = [[0.0 for row in range(dimy)] for col in range(dimx)]
```

```
    # ------------
    #
    # makes matrix of a certain (square) size and turns matrix into identity matrix
    #

    def identity(self, dim):
        # check if valid dimension
        if dim < 1:
            raise ValueError, "Invalid size of matrix"
        else:
            self.dimx  = dim
            self.dimy  = dim
            self.value = [[0.0 for row in range(dim)] for col in range(dim)]
            for i in range(dim):
                self.value[i][i] = 1.0
    # ------------
    #
    # prints out values of matrix
    #

    def show(self, txt = ''):
        for i in range(len(self.value)):
            print txt + '['+ ', '.join('%.3f'%x for x in self.value[i]) + ']'
        print ' '

    # ------------
    #
    # defines elmement-wise matrix addition. Both matrices must be of equal dimensions
    #

    def __add__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError, "Matrices must be of equal dimension to add"
        else:
            # add if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] + other.value[i][j]
            return res

    # ------------
    #
    # defines elmement-wise matrix subtraction. Both matrices must be of equal dimensions
    #

    def __sub__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimx != other.dimx:
            raise ValueError, "Matrices must be of equal dimension to subtract"
        else:
            # subtract if correct dimensions
            res = matrix()
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] - other.value[i][j]
            return res

    # ------------
    #
    # defines multiplication. Both matrices must be of fitting dimensions
    #

    def __mul__(self, other):
        # check if correct dimensions
        if self.dimy != other.dimx:
            raise ValueError, "Matrices must be m*n and n*p to multiply"
        else:
            # multiply if correct dimensions
            res = matrix()
            res.zero(self.dimx, other.dimy)
            for i in range(self.dimx):
                for j in range(other.dimy):
                    for k in range(self.dimy):
                        res.value[i][j] += self.value[i][k] * other.value[k][j]
```

```
        return res


    # ------------
    #
    # returns a matrix transpose
    #

    def transpose(self):
        # compute transpose
        res = matrix()
        res.zero(self.dimy, self.dimx)
        for i in range(self.dimx):
            for j in range(self.dimy):
                res.value[j][i] = self.value[i][j]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[ 1,  2,  3,  4,  5],
    #                   [ 6,  7,  8,  9, 10],
    #                   [11, 12, 13, 14, 15]])
    #
    #       l.take([0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 3, 4],
    #        [11, 13, 14]]
    #
    #
    # take is used to remove rows and columns from existing matrices
    # list1/list2 define a sequence of rows/columns that shall be taken
    # is no list2 is provided, then list2 is set to list1 (good for
    # symmetric matrices)
    #

    def take(self, list1, list2 = []):
        if list2 == []:
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError, "list invalid in take()"

        res = matrix()
        res.zero(len(list1), len(list2))
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[i][j] = self.value[list1[i]][list2[j]]
        return res

    # ------------
    #
    # creates a new matrix from the existing matrix elements.
    #
    # Example:
    #       l = matrix([[1, 2, 3],
    #                   [4, 5, 6]])
    #
    #       l.expand(3, 5, [0, 2], [0, 2, 3])
    #
    # results in:
    #
    #       [[1, 0, 2, 3, 0],
    #        [0, 0, 0, 0, 0],
    #        [4, 0, 5, 6, 0]]
    #
    # expand is used to introduce new rows and columns into an existing matrix
    # list1/list2 are the new indexes of row/columns in which the matrix
    # elements are being mapped. Elements for rows and columns
    # that are not listed in list1/list2
    # will be initialized by 0.0.
    #

    def expand(self, dimx, dimy, list1, list2 = []):
        if list2 == []:
```

```
            list2 = list1
        if len(list1) > self.dimx or len(list2) > self.dimy:
            raise ValueError, "list invalid in expand()"

        res = matrix()
        res.zero(dimx, dimy)
        for i in range(len(list1)):
            for j in range(len(list2)):
                res.value[list1[i]][list2[j]] = self.value[i][j]
        return res

    # ------------
    #
    # Computes the upper triangular Cholesky factorization of
    # a positive definite matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560
    #

    def Cholesky(self, ztol= 1.0e-5):

        res = matrix()
        res.zero(self.dimx, self.dimx)

        for i in range(self.dimx):
            S = sum([(res.value[k][i])**2 for k in range(i)])
            d = self.value[i][i] - S
            if abs(d) < ztol:
                res.value[i][i] = 0.0
            else:
                if d < 0.0:
                    raise ValueError, "Matrix not positive-definite"
                res.value[i][i] = sqrt(d)
            for j in range(i+1, self.dimx):
                S = sum([res.value[k][i] * res.value[k][j] for k in range(i)])
                if abs(S) < ztol:
                    S = 0.0
                res.value[i][j] = (self.value[i][j] - S)/res.value[i][i]
        return res

    # ------------
    #
    # Computes inverse of matrix given its Cholesky upper Triangular
    # decomposition of matrix.
    # This code is based on http://adorio-research.org/wordpress/?p=4560
    #

    def CholeskyInverse(self):

        res = matrix()
        res.zero(self.dimx, self.dimx)

        # Backward step for inverse.
        for j in reversed(range(self.dimx)):
            tjj = self.value[j][j]
            S = sum([self.value[j][k]*res.value[j][k] for k in range(j+1, self.dimx)])
            res.value[j][j] = 1.0/ tjj**2 - S/ tjj
            for i in reversed(range(j)):
                res.value[j][i] = res.value[i][j] = \
                    -sum([self.value[i][k]*res.value[k][j] for k in \
                          range(i+1,self.dimx)])/self.value[i][i]
        return res

    # ------------
    #
    # comutes and returns the inverse of a square matrix
    #
    def inverse(self):
        aux = self.Cholesky()
        res = aux.CholeskyInverse()
        return res

    # ------------
    #
    # prints matrix (needs work!)
    #
    def __repr__(self):
        return repr(self.value)
```

```
# --------------------------------------------------
#
# this is the robot class
#
# our robot lives in x-y space, and its motion is
# pointed in a random direction. It moves on a straight line
# until is comes close to a wall at which point it turns
# away from the wall and continues to move.
#
# For measurements, it simply senses the x- and y-distance
# to landmarks. This is different from range and bearing as
# commonly studied in the literature, but this makes it much
# easier to implement the essentials of SLAM without
# cluttered math
#

class robot:

    # --------
    # init:
    #   creates robot and initializes location to 0, 0
    #

    def __init__(self, world_size = 100.0, measurement_range = 30.0,
                 motion_noise = 1.0, measurement_noise = 1.0):
        self.measurement_noise = 0.0
        self.world_size = world_size
        self.measurement_range = measurement_range
        self.x = world_size / 2.0
        self.y = world_size / 2.0
        self.motion_noise = motion_noise
        self.measurement_noise = measurement_noise
        self.landmarks = []
        self.num_landmarks = 0


    def rand(self):
        return random.random() * 2.0 - 1.0

    # --------
    #
    # make random landmarks located in the world
    #

    def make_landmarks(self, num_landmarks):
        self.landmarks = []
        for i in range(num_landmarks):
            self.landmarks.append([round(random.random() * self.world_size),
                                   round(random.random() * self.world_size)])
        self.num_landmarks = num_landmarks


    # --------
    #
    # move: attempts to move robot by dx, dy. If outside world
    #       boundary, then the move does nothing and instead returns failure
    #

    def move(self, dx, dy):

        x = self.x + dx + self.rand() * self.motion_noise
        y = self.y + dy + self.rand() * self.motion_noise

        if x < 0.0 or x > self.world_size or y < 0.0 or y > self.world_size:
            return False
        else:
            self.x = x
            self.y = y
            return True


    # --------
    #
    # sense: returns x- and y- distances to landmarks within visibility range
    #        because not all landmarks may be in this range, the list of measurements
    #        is of variable length. Set measurement_range to -1 if you want all
    #        landmarks to be visible at all times
    #
```

```
    def sense(self):
        Z = []
        for i in range(self.num_landmarks):
            dx = self.landmarks[i][0] - self.x + self.rand() * self.measurement_noise
            dy = self.landmarks[i][1] - self.y + self.rand() * self.measurement_noise
            if self.measurement_range < 0.0 or abs(dx) + abs(dy) <= self.measurement_range:
                Z.append([i, dx, dy])
        return Z

    # --------
    #
    # print robot location
    #

    def __repr__(self):
        return 'Robot: [x=%.5f y=%.5f]'  % (self.x, self.y)

######################################################

# --------
# this routine makes the robot data
#

def make_data(N, num_landmarks, world_size, measurement_range, motion_noise,
              measurement_noise, distance):


    complete = False

    while not complete:

        data = []

        # make robot and landmarks
        r = robot(world_size, measurement_range, motion_noise, measurement_noise)
        r.make_landmarks(num_landmarks)
        seen = [False for row in range(num_landmarks)]

        # guess an initial motion
        orientation = random.random() * 2.0 * pi
        dx = cos(orientation) * distance
        dy = sin(orientation) * distance

        for k in range(N-1):

            # sense
            Z = r.sense()

            # check off all landmarks that were observed
            for i in range(len(Z)):
                seen[Z[i][0]] = True

            # move
            while not r.move(dx, dy):
                # if we'd be leaving the robot world, pick instead a new direction
                orientation = random.random() * 2.0 * pi
                dx = cos(orientation) * distance
                dy = sin(orientation) * distance

            # memorize data
            data.append([Z, [dx, dy]])

        # we are done when all landmarks were observed; otherwise re-run
        complete = (sum(seen) == num_landmarks)

    print ' '
    print 'Landmarks: ', r.landmarks
    print r


    return data

######################################################

# --------------------------------
#
# print the result of SLAM, the robot pose(s) and the landmarks
```

```
#

def print_result(N, num_landmarks, result):
    print
    print 'Estimated Pose(s):'
    for i in range(N):
        print '    ['+ ', '.join('%.3f'%x for x in result.value[2*i]) + ', ' \
            + ', '.join('%.3f'%x for x in result.value[2*i+1]) +']'
    print
    print 'Estimated Landmarks:'
    for i in range(num_landmarks):
        print '    ['+ ', '.join('%.3f'%x for x in result.value[2*(N+i)]) + ', ' \
            + ', '.join('%.3f'%x for x in result.value[2*(N+i)+1]) +']'

# --------------------------------
#
# slam - retains entire path and all landmarks
#

############### ENTER YOUR CODE BELOW HERE ###################
def slam(data, N, num_landmarks, motion_noise, measurement_noise):
    #
    #
    # Add your code here!
    #
    #
    #set the dimension of the filter
    dim = 2 * (N + num_landmarks)
    #make the constraint information matrix and vector
    Omega = matrix()
    Omega.zero(dim,dim)
    Omega.value[0][0] = 1.0
    Omega.value[1][1] = 1.0
    Xi = matrix()
    Xi.zero(dim, 1)
    Xi.value[0][0] = world_size / 2
    Xi.value[1][0] = world_size / 2
    for k in range(len(data)):
        #n is the index of the robots pose in the matrix/vector
        n=k*2
        measurement = data[k][0]
        motion = data[k][1]
        # integrate measurements
        for i in range(len(measurement)):
            #m is the index of the landmark coordinate in the matrix/vector
            m = 2 * (N + measurement[i][0])
            # update the information matrix according to measurement
            for b in range(2):
                Omega.value[n+b][n+b] +=  1.0 / measurement_noise
                Omega.value[m+b][m+b] +=  1.0 / measurement_noise
                Omega.value[n+b][m+b] += -1.0 / measurement_noise
                Omega.value[m+b][n+b] += -1.0 / measurement_noise
                Xi.value[n+b][0]      += -measurement[i][1+b] / measurement_noise
                Xi.value[m+b][0]      +=  measurement[i][1+b] / measurement_noise
        # update the information matrix according to motion
        for b in range(4):
            Omega.value[n+b][n+b]     +=  1.0 / motion_noise
        for b in range(2):
            Omega.value[n+b  ][n+b+2] += -1.0 / motion_noise
            Omega.value[n+b+2][n+b  ] += -1.0 / motion_noise
            Xi.value[n+b  ][0]        += -motion[b] / motion_noise
            Xi.value[n+b+2][0]        +=  motion[b] / motion_noise
    mu = Omega.inverse() * Xi
    return mu # Make sure you return mu for grading!



############### ENTER YOUR CODE ABOVE HERE ###################

# ------------------------------------------------------------------------
# ------------------------------------------------------------------------
# ------------------------------------------------------------------------
#
# Main routines
#


num_landmarks      = 5        # number of landmarks
```

```
N                   = 20       # time steps
world_size          = 100.0    # size of world
measurement_range   = 50.0     # range at which we can sense landmarks
motion_noise        = 2.0      # noise in robot motion
measurement_noise   = 2.0      # noise in the measurements
distance            = 20.0     # distance by which robot (intends to) move each iteratation

data = make_data(N, num_landmarks, world_size, measurement_range, motion_noise,
measurement_noise, distance)
result = slam(data, N, num_landmarks, motion_noise, measurement_noise)
print_result(N, num_landmarks, result)

# -------------
# Testing
#
# Uncomment one of the test cases below to compare your results to
# the results shown for Test Case 1 and Test Case 2.

test_data1 = [[[[1, 19.457599255548065, 23.8387362100849], [2, -13.195807561967236,
11.708840328458608], [3, -30.0954905279171, 15.387879242505843]], [-12.2607279422326, -
15.801093326936487]], [[[2, -0.4659930049620491, 28.088559771215664], [4, -17.866382374890936, -
16.384904503932]], [-12.2607279422326, -15.801093326936487]], [[[4, -6.202512900833806, -
1.823403210274639]], [-12.2607279422326, -15.801093326936487]], [[[4, 7.412136480918645,
15.388585962142429]], [14.008259661173426, 14.274756084260822]], [[[4, -7.526138813444998, -
0.4563942429717849]], [14.008259661173426, 14.274756084260822]], [[[2, -6.299793150150058,
29.047830407717623], [4, -21.93551130411791, -13.21956810989039]], [14.008259661173426,
14.274756084260822]], [[[1, 15.796300959032276, 30.65769689694247], [2, -18.643708219834820,
17.380022987031367]], [14.008259661173426, 14.274756084260822]], [[[1, 0.40311325410337906,
14.169429532679855], [2, -35.069349468466235, 2.4945558982439957]], [14.008259661173426,
14.274756084260822]], [[[1, -16.71340983241936, -2.777000269543834]], [-11.006096015782283,
16.699276945166858]], [[[1, -3.611096830835776, -17.954019226763958]], [-19.693482634035977,
3.4880856845730480]], [[[1, 18.398273354362416, -22.705102332550947]], [-19.693482634035977,
3.4880856845730480]], [[[2, 2.789312482883833, -39.737201931213240]], [12.849049222879723, -
15.326510824972983]], [[[1, 21.26897046581808, -10.121029799040915], [2, -11.917698965880655, -
23.177116626020970], [3, -31.81167947898398, -16.7985673023331]], [12.849049222879723, -
15.326510824972983]], [[[1, 10.48157743234859, 5.692957082575485], [2, -22.314884735549350, -
5.389184118551409], [3, -40.81803984305378, -2.4703329790238118]], [12.849049222879723, -
15.326510824972983]], [[[0, 10.591050242096598, -39.20517989671130], [1, -3.5675572049297553,
22.849456408289125], [2, -38.39251065320351, 7.288990306029511]], [12.849049222879723, -
15.326510824972983]], [[[0, -3.6225556479370766, -25.580068652355120]], [-7.8874682868419965, -
18.379005523261092]], [[[0, 1.9784503557879374, -6.5025974151499]], [-7.8874682868419965, -
18.379005523261092]], [[[0, 10.050665232782423, 11.026385307998742]], [-17.829193597782980,
9.062000642947142]], [[[0, 26.526838150174818, -0.22563393232425621], [4, -33.703039368866520,
2.8803398410136770]], [-17.829193597782980, 9.062000642947142]]]
#test_data2 = [[[[0, 26.543274387283322, -6.262538160312672], [3, 9.937396825799755, -
9.128540360867689]], [18.92765331253674, -6.460955043986683]], [[[0, 7.706544739722961, -
3.758467215445748], [1, 17.03954411948937, 31.705489938553438], [3, -11.61731288777497, -
6.649640967164160]], [18.92765331253674, -6.460955043986683]], [[[0, -12.35130507136378,
2.585119104239249], [1, -2.563534536165313, 38.22159657838369], [3, -26.961236804740935, -
0.4802312626141525]], [-11.167066095509824, 16.592065417497455]], [[[0, 1.4138633151721272, -
13.912454837810632], [1, 8.087721200818589, 20.51845934354381], [3, -17.091723454402302, -
16.521500551709707], [4, -7.414211721400232, 38.09191602674439]], [-11.167066095509824,
16.592065417497455]], [[[0, 12.886743222179561, -28.703968411636318], [1, 21.660953298391387,
3.4912891084614914], [3, -6.401401414569506, -32.321583037341625], [4, 5.034079343639034,
23.102207946092893]], [-11.167066095509824, 16.592065417497455]], [[[1, 31.126317672358578, -
10.036784369535214], [2, -38.70878528420893, 7.4987265861424595], [4, 17.977218575473767,
6.150889254289742]], [-6.595520680493778, -18.88118393939265]], [[[1, 41.82460922922086,
7.847527392202475], [3, 15.711709540417502, -30.34633659912818]], [-6.595520680493778, -
18.88118393939265]], [[[0, 40.18454208294434, -6.710999804403755]], [3, 23.019508919299156, -
10.121108672906040]], [-6.595520680493778, -18.88118393939265]], [[[3, 27.18579315312821,
8.067219022708391]], [-6.595520680493778, -18.88118393939265]], [[], [11.492663265706092,
16.368221988386210]], [[[3, 24.57154567653098, 13.461499960708197]], [11.492663265706092,
16.368221988386210]], [[[0, 31.61945290413707, 0.4272295085799329], [3, 16.973922991589910, -
5.274596836133088]], [11.492663265706092, 16.368221988386210]], [[[0, 22.407381798735177, -
18.035000683792590], [1, 29.642444125196995, 17.3794951934614], [3, 4.7969752441371645, -
21.075053616399690], [4, 14.726069092569372, 32.75999422300078]], [11.492663265706092,
16.368221988386210]], [[[0, 10.705527984670137, -34.589764174299596], [1, 18.587723367956030, -
0.2010970816478776 5], [3, -4.839806195049413, -39.92208742305105], [4, 4.188248101654540,
14.146847823548889]], [11.492663265706092, 16.368221988386210]], [[[1, 5.878492140223764, -
19.955352450942357], [4, -7.059505455306587, -0.9740849280550585]], [19.628527845173146,
3.836781806574670]], [[[1, -11.150789592446378, -22.736641053247872], [4, -28.832815721158255, -
3.9462962046291388]], [-19.841703647091965, 2.5113335861604362]], [[[1, 8.64427397916182, -
20.286336970889053], [4, -5.036917727942285, -6.311739993868336]], [-5.946642674882207, -
19.095482211697870]], [[[0, 7.151866679283043, -39.561032326163690], [1, 16.015354013733680, -
3.780995345194027], [4, -3.048013318321370, 13.697362774960865]], [-5.946642674882207, -
19.095482211697870]], [[[0, 12.872879480504395, -19.707592098123207], [1, 22.236710716903136,
16.331770792606406], [3, -4.841206109583004, -21.246044358512420], [4, 4.271111632235520,
32.253097486141840]], [-5.946642674882207, -19.095482211697870]]]
```

```
##  Test Case 1
##
##  Estimated Pose(s):
##      [49.999, 49.999]
##      [37.971, 33.650]
##      [26.183, 18.153]
##      [13.743, 2.114]
##      [28.095, 16.781]
##      [42.383, 30.900]
##      [55.829, 44.494]
##      [70.855, 59.697]
##      [85.695, 75.540]
##      [74.010, 92.431]
##      [53.543, 96.451]
##      [34.523, 100.078]
##      [48.621, 83.951]
##      [60.195, 68.105]
##      [73.776, 52.932]
##      [87.130, 38.536]
##      [80.301, 20.506]
##      [72.797, 2.943]
##      [55.244, 13.253]
##      [37.414, 22.315]
##
##  Estimated Landmarks:
##      [82.954, 13.537]
##      [70.493, 74.139]
##      [36.738, 61.279]
##      [18.696, 66.057]
##      [20.633, 16.873]


##  Test Case 2
##
##  Estimated Pose(s):
##      [49.999, 49.999]
##      [69.180, 45.664]
##      [87.742, 39.702]
##      [76.269, 56.309]
##      [64.316, 72.174]
##      [52.256, 88.151]
##      [44.058, 69.399]
##      [37.001, 49.916]
##      [30.923, 30.953]
##      [23.507, 11.417]
##      [34.179, 27.131]
##      [44.154, 43.844]
##      [54.805, 60.919]
##      [65.697, 78.544]
##      [77.467, 95.624]
##      [96.801, 98.819]
##      [75.956, 99.969]
##      [70.199, 81.179]
##      [64.053, 61.721]
##      [58.106, 42.626]
##
##  Estimated Landmarks:
##      [76.778, 42.885]
##      [85.064, 77.436]
##      [13.546, 95.649]
##      [59.448, 39.593]
##      [69.262, 94.238]



### Uncomment the following three lines for test case 1 ###

result = slam(test_data1, 20, 5, 2.0, 2.0)
print_result(20, 5, result)
#print result


### Uncomment the following three lines for test case 2 ###

#result = slam(test_data2, 20, 5, 2.0, 2.0)
#print_result(20, 5, result)
#print result
```

Поздравляю Вы сделали это!