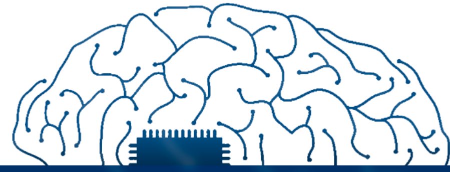
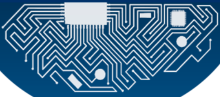


СПбГЭТУ «ЛЭТИ»
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ



А.Ю. Филатов

Задачи по SLAM-алгоритмам

Фрагмент конспекта

СПбГЭТУ «ЛЭТИ», 2022 г.

1.1 Цель работы

Изучить основы работы SLAM, используя расширенный фильтр Калмана.

1.2 Общие положения

Для выполнения работы необходимо теорию SLAM алгоритмов, алгоритм их работы.

В работе требуется реализовать программу на языке Python.

Проверка работы осуществляется на moodle.

1.3 Порядок выполнения

Требуется решить задачу, в которой необходимо написать правильную последовательность вызова функций, которые определяют новые координаты робота, используя расширенный фильтр Калмана.

Описание подготовленных функций, которые нужно использовать при решение задачи:

- **pred_ekf** – функция, которая предсказывает новое положение робота.

  На вход принимает:

  1. Нынешнее положение робота
  2. Ковариация для положения робота
  3. Вектор движения

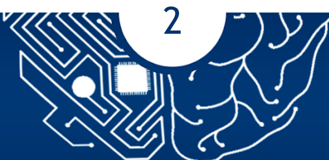
- **update_ekf** – функция, которая обновляет положение робота в пространстве. Обновляет данные об опорных точках(landmarks).

  На вход принимает:

  1. Положение робота в пространстве
  2. Ковариация для положения робота
  3. Измерение, с сенсора
  4. Матрица, отвечающая за опорные точки (landmarks)

  На выход выдает:

  1. Новое положение робота
  2. Ковариация для положения робота



3. Новую матрицу, отвечающую за опорные точки

- **update_mu** – возвращает новые обработанные координаты робота с учетом старых и новых координат робота. Необходимо вызывать после предсказания и обновления положения робота.
  На вход принимает:
  1. Старые координаты робота
  2. Новые координаты робота

  На выходе получается скорректированные координаты робота
- **update_prob** – возвращает новую вероятностную матрицу, в которой описаны все координаты landmarks. На вход принимает:
  1. Старую вероятностную матрицу с landmarks
  2. Новую вероятностную матрицу с landmarks

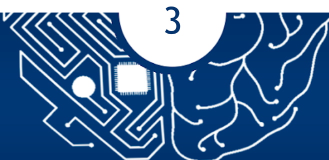
На выходе получается обновленная матрица, которая содержит информацию о landmarks.

1.4 Результат работы

Пройдена проверка на moodle.

1.5 Контрольные вопросы и задачи

1. Опишите работы ekf-slam?
2. Что такое опорные точки? Приведите пример.



2.1 Цель работы

Изучить SLAM алгоритм, основанный на фильтре частиц.

2.2 Общие положения

Для выполнения работы необходимо изучить теорию по SLAM алгоритмам и разобраться в том, как они работают.

В задании требуется реализовать программу на языке Python.

Проверка работы осуществляется на платформе moodle.

2.3 Порядок выполнения

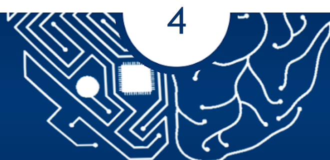
Требуется реализовать функцию, в которой нужно написать правильную последовательность вызова уже реализованных функций в соответствии с fast-slam алгоритмом. Также нужно будет реализовать одну функцию, которая отвечает за обновление состояния частиц.

Описание функций, которые надо реализовать:

- **update_with_observation** - функция, которую необходимо реализовать. Обновляет массив частиц на основе наблюдений

  Сигнатура функции.
  - particles - массив частиц. Частица - объект класса Particle.
  - z - массив наблюдений
    z[0] - x-вые координаты наблюдений
    z[1] - y-вые координаты наблюдений
    z[2] - id наблюдения.

    Пример массива:
    ```
    [[ 9.2494106   17.57994024  15.55470995  7.60171905  16.39453923]
     [-0.28770964  0.61160783   1.37903837   2.24327946  2.22198108]
     [ 0.          1.           4.           6.          7.        ]]
    ```

Шаги реализации функции:

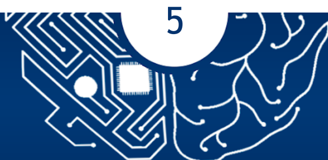
Основная идея: перебирая все наблюдения, обновить информацию о каждом наблюдении у каждой частицы, тем самым получить более точные данные частиц.

1. Перебрать все частицы и проверить, новое это для них наблюдение или нет.
2. Сравнить расстояние между наблюдением, информация о котором хранится в частице с тем наблюдением, которое мы рассматриваем. Точность сравнения - 0.01.
3. Если расстояние меньше, тогда это новое наблюдение. Добавим информацию о нём в частицу с помощью функции add_new_landmark. Если расстояние больше, тогда это известное наблюдение, нужно обновить вес частицы и информацию о данном наблюдении у частицы.
4. Рассчитаем новый вес частицы. Применим функцию compute_weight, умножим полученный результат на старый вес частицы. Полученный результат будет новым весом частицы.
5. Затем обновим информацию о данном наблюдении у частицы с помощью функции update_landmark.
6. Применив вышестоящие действия для всех частиц и всех наблюдений вернём из функции массив частиц.

- **solution** - функция, в которой нужно последовательно вызвать уже реализованные функции в соответствии с алгоритмом Fast-Slam

Сигнатура функции.

- particles - массив с начальным состоянием частиц
- movements - массив с движениями робота
- observations - массив наблюдений, полученных при каждом движении



Идея функции.

Вам нужно будет использовать массивы movements и observations. В цикле получать из них движение и наблюдение, а затем обновлять частицы и получать состояние робота. На каждом шаге состояние нужно будет записать в массив, чтобы при проверке вашего решения подсчитать отклонение от настоящих значений робота.

Функции, которые нужно последовательно вызвать в функции: update_with_observation, predict_particles, resampling, calc_final_state.

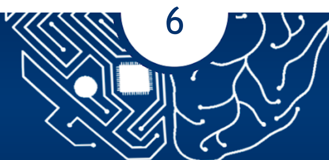
Описание данных функций:

- **update_with_observation** - обновляет массив частиц на основе наблюдений
  Аргументы:
  - particles - массив частиц
  - u - массив наблюдений

- **predict_particles** - по текущему набору частиц и движению робота рассчитывает то, как изменятся частицы, если робот совершит данное движение
  Аргументы:
  - particles - массив частиц
  - u - движение робота

- **resampling** - производит отсеивание частиц на основе данных об их весе. Возвращает обновленный массив частиц.
  Аргументы:
  - particles - массив частиц.

- **calc_final_state** - рассчитывает положение робота на основе информации о частицах. Возвращает положение робота.
  Аргументы:
  - particles - массив частиц.



Код уже реализованных функций можно посмотреть в Приложении А.

2.4 Результат работы

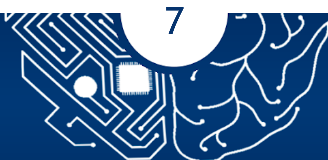
Пройдена проверка на moodle.

2.5 Контрольные вопросы и задачи

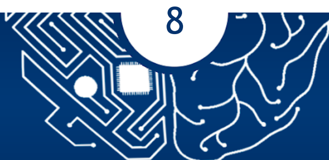
1. Опишите алгоритм работы fast-slam.
2. Опишите, почему алгоритм так назван. Он действительно быстрее работает?

2.6 Приложение А. Код функций

```
# Fast SLAM covariance
Q = np.diag([3.0, np.deg2rad(10.0)]) ** 2
R = np.diag([1.0, np.deg2rad(20.0)]) ** 2

#  Simulation parameter
Q_sim = np.diag([0.3, np.deg2rad(2.0)]) ** 2
R_sim = np.diag([0.5, np.deg2rad(10.0)]) ** 2
OFFSET_YAW_RATE_NOISE = 0.01

DT = 0.1  # time tick [s]
SIM_TIME = 10.0  # simulation time [s]
MAX_RANGE = 20.0  # maximum observation range
M_DIST_TH = 2.0  # Threshold of Mahalanobis distance for data association.
STATE_SIZE = 3  # State size [x,y,yaw]
LM_SIZE = 2  # LM state size [x,y]
N_PARTICLE = 100  # number of particle
NTH = N_PARTICLE / 1.5  # Number of particle for re-sampling


class Particle:

      def __init__(self, n_landmark):
      self.w = 1.0 / N_PARTICLE
      self.x = 0.0
      self.y = 0.0
      self.yaw = 0.0
```

```
        # landmark x-y positions
        self.lm = np.zeros((n_landmark, LM_SIZE))
        # landmark position covariance
        self.lmP = np.zeros((n_landmark * LM_SIZE, LM_SIZE))


def normalize_weight(particles):
        sum_w = sum([p.w for p in particles])

        try:
        for i in range(N_PARTICLE):
                particles[i].w /= sum_w
        except ZeroDivisionError:
                for i in range(N_PARTICLE):
                        particles[i].w = 1.0 / N_PARTICLE

                return particles

        return particles


def calc_final_state(particles):
        xEst = np.zeros((STATE_SIZE, 1))

        particles = normalize_weight(particles)

        for i in range(N_PARTICLE):
        xEst[0, 0] += particles[i].w * particles[i].x
        xEst[1, 0] += particles[i].w * particles[i].y
        xEst[2, 0] += particles[i].w * particles[i].yaw
        xEst[2, 0] = pi_2_pi(xEst[2, 0])

        return xEst


def predict_particles(particles, u):
        for i in range(N_PARTICLE):
        px = np.zeros((STATE_SIZE, 1))
        px[0, 0] = particles[i].x
        px[1, 0] = particles[i].y
        px[2, 0] = particles[i].yaw
        ud = u + (np.random.randn(1, 2) @ R ** 0.5).T  # add noise
        px = motion_model(px, ud)
        particles[i].x = px[0, 0]
        particles[i].y = px[1, 0]
        particles[i].yaw = px[2, 0]
```

```
    return particles


def add_new_landmark(particle, z, Q_cov):
    r = z[0]
    b = z[1]
    lm_id = int(z[2])

    s = math.sin(pi_2_pi(particle.yaw + b))
    c = math.cos(pi_2_pi(particle.yaw + b))

    particle.lm[lm_id, 0] = particle.x + r * c
    particle.lm[lm_id, 1] = particle.y + r * s

    # covariance
    dx = r * c
    dy = r * s
    d2 = dx ** 2 + dy ** 2
    d = math.sqrt(d2)
    Gz = np.array([[dx / d, dy / d],
                   [-dy / d2, dx / d2]])
    particle.lmP[2 * lm_id:2 * lm_id + 2] = np.linalg.inv(
    Gz) @ Q_cov @ np.linalg.inv(Gz.T)

    return particle


def compute_jacobians(particle, xf, Pf, Q_cov):
    dx = xf[0, 0] - particle.x
    dy = xf[1, 0] - particle.y
    d2 = dx ** 2 + dy ** 2
    d = math.sqrt(d2)

    zp = np.array(
    [d, pi_2_pi(math.atan2(dy, dx) - particle.yaw)]).reshape(2, 1)

    Hv = np.array([[-dx / d, -dy / d, 0.0],
                   [dy / d2, -dx / d2, -1.0]])

    Hf = np.array([[dx / d, dy / d],
                   [-dy / d2, dx / d2]])

    Sf = Hf @ Pf @ Hf.T + Q_cov

    return zp, H
```