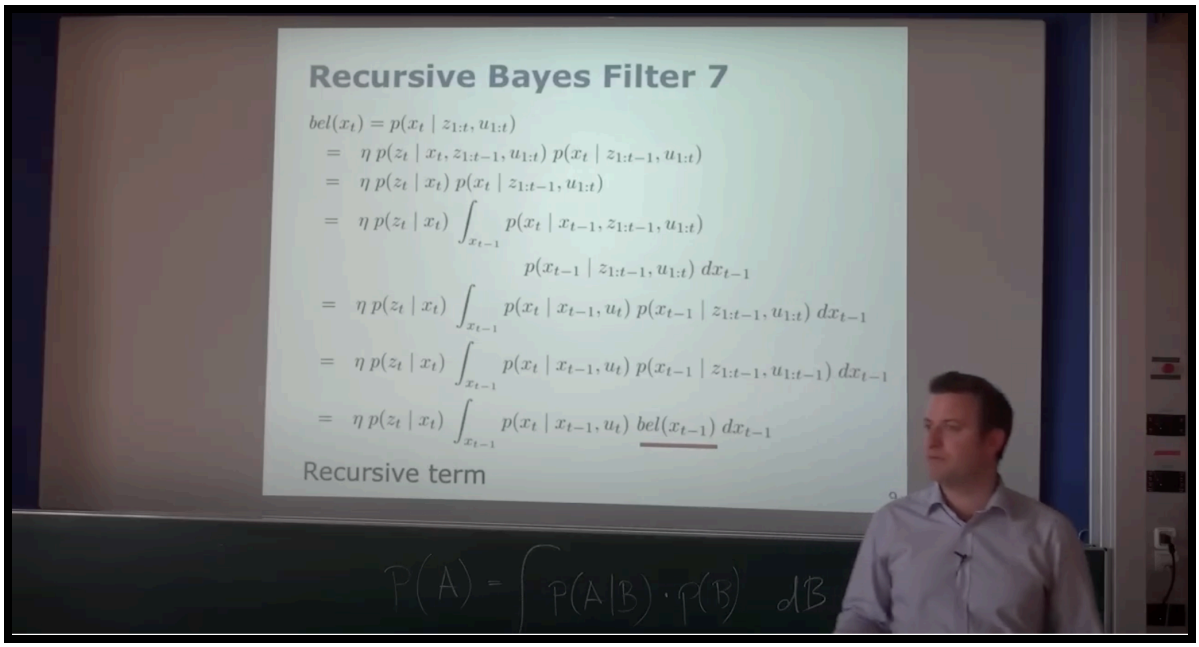
My imaginary robotics project is about a robot which is placed on the floor, and tasked with mapping out a room in the fastest time. Therefore, I analyzed the SLAM folder within the repository. I specifically analyzed the file titled "fast_slam1.py" and wrote comments on it.

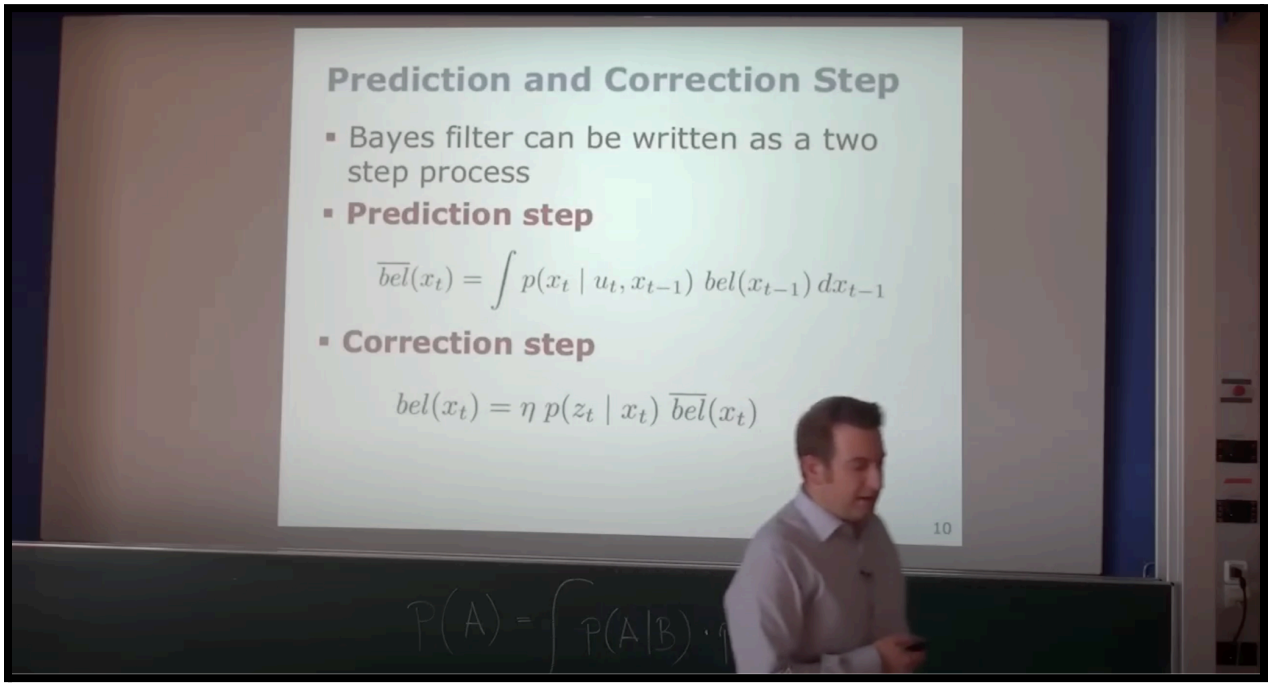
The initial section of the file includes the "import"s. Math imports mathematical functions, "matplotlib.pyplot" imports graphing, "numpy" imports numerical operations and utils.angle imports angle operations.

Next, we have the initializations of certain variables. As there are too many of these to write about, I have selected a few to go over. The first variable is "DT," which is given the value of 0.1. This means that between each calculations of the robot, there are 0.1 seconds. SIM_TIME is the total simulation time (for my project, SIM_TIME would be determined by the robot itself, as soon as the mapping finishes.) In this instance, it is 50 seconds.

For the other sections of the file, because I do not fully understand the workings of FastSLAM, but because I have learned of the Bayes' filter, and as "FastSLAM is based on the theory of Bayesian filtering," (Kim) I will try to explain how the code fits into the theory of Bayesian filtering.



(Stachnis, https://youtu.be/5Pu558YtjYM?si=2tsFCo-68BCzYIk0)



(Stachnis, https://youtu.be/5Pu558YtjYM?si=2tsFCo-68BCzYIk0)

The prediction step, as seen on the slide, is represented with the function defined between the lines 89 and 99, which is provided below:

```
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
```

The method is called within the function named “fast_slam1,” which is also provided:

```
def fast_slam1(particles, u, z):
    particles = predict_particles(particles, u)

    particles = update_with_observation(particles, z)

    particles = resampling(particles)

    return particles
```

As fastSLAM utilizes particle filters, the bel(x_t) is represented by a set of particles, where each particle has a particle has a state (x, y yaw) and a weight. We see that a variable named “px” is initialized to the current states of the particles in the code provided at the top of this page. Then, noise is added (if I am not mistaken, this is not explicitly expressed in the equation for the Bayes’ filter. After that, the motion_model function is called, which will be explained in the next paragraph:

The part of the formula written as $p(x_t \mid x_{t-1}, u_t)$ is represented by the motion_model, which is written as:

```
def motion_model(x, u):
    F = np.array([[1.0, 0, 0],
                  [0, 1.0, 0],
                  [0, 0, 1.0]])

    B = np.array([[DT * math.cos(x[2, 0]), 0],
                  [DT * math.sin(x[2, 0]), 0],
                  [0.0, DT]])

    x = F @ x + B @ u

    x[2, 0] = pi_2_pi(x[2, 0])

    return x
```

As it can be seen, the method takes in 2 parameters: x, which represents the position, and u, which represents the controls. As it can be seen, the next position of the particle is calculated using this line of code: x = F @ x + B @ u

The last part of the equation, which is the correction step, is written on the second page. The part of the correction step written as $p(z_t|x_t)$ is represented by the function named compute_weight:

```
def compute_weight(particle, z, Q_cov):
    lm_id = int(z[2])
    xf = np.array(particle.lm[lm_id, :]).reshape(2, 1)
    Pf = np.array(particle.lmP[2 * lm_id:2 * lm_id + 2])
    zp, Hv, Hf, Sf = compute_jacobians(particle, xf, Pf, Q_cov)

    dx = z[0:2].reshape(2, 1) - zp
    dx[1, 0] = pi_2_pi(dx[1, 0])
```

```
    try:
        invS = np.linalg.inv(Sf)
    except np.linalg.linalg.LinAlgError:
        print("singular")
        return 1.0

    num = np.exp(-0.5 * (dx.T @ invS @ dx))[0, 0]
    den = 2.0 * math.pi * math.sqrt(np.linalg.det(Sf))

    w = num / den

    return w
```

Basically, this function calculates the likelihood of the observation (z_t) given the particle's state (x_t). As I do not have an in-depth understanding of fastSLAM, I will not be able to elucidate the usage of other functions within this method which are not directly represented by the Bayes' filter, such as compute_jacobians.

Lastly, I can mention the function titled normalize_weight, which represents the normalizing factor as it makes sure the weight of all the particles add up to one.

Note: there is no integration in the code. This is explained by ChatGPT as "Instead of integrating over a continuous space, we perform a discrete sum over all particles."

-Can