Part I

Assignment 1

1 Dynamical Systems and Motion Control

1.1 Exercise 1

1.1.1 Part (a)

The robot is described by the state $\mathbf{x} = [x, y, \theta]^T$, the differential equations that describe the motion of the system are

$$\dot{x} = ucos(\theta)$$
$$\dot{y} = usin(\theta)$$
$$\dot{\theta} = \omega = cos(t)$$

These can also be expressed as explicit functions of time

$$\dot{x} = ucos[sin(t)]$$
$$\dot{y} = usin[sin(t)]$$
$$\dot{\theta} = cos(t)$$

1.1.2 Part (b)

Euler's method is a first order integration method to calculate the state in one time step from the current state and it's derivatives. This works using a Taylor expansion of the function about it's current state and approximating the next step by considering only the first derivative. The method essentially calculates the next step by moving a fixed amount along the tangent of the true path at the current step, and thus a small step size is required such that the gradient doesn't change too much between steps. If we define a function f such that

$$f : \mathbf{x} \to \dot{\mathbf{x}}$$

with the derivative of $\mathbf{x}$ described in part (a). Euler integration then gives

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t f(\mathbf{x}, t)$$

In python however it is easier to implement the update for each state separately so

$$x_{k+1} = x_k + ucos(\theta_k)\Delta t$$
$$y_{k+1} = y_k + usin(\theta_k)\Delta t$$
$$\theta_{k+1} = \theta_k + cos(t)\Delta t$$

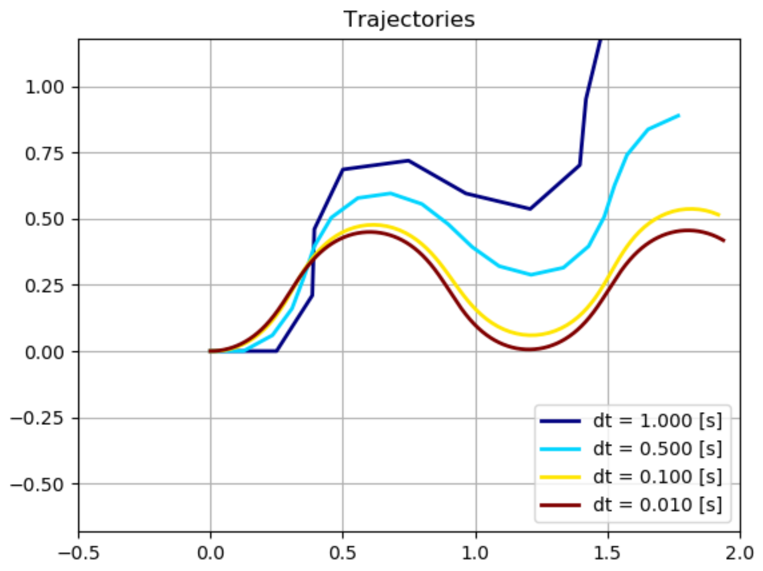


Figure 1: Kinematics with Euler integration for a selection of time steps

1.1.3 Part (c)

Running the kinematic simulation with $\Delta t = 1, 0.5, 0.1, 0.01$ gives the plot shown in figure 1. We see that for small time steps the Euler method is very unstable, this is because the gradient of the states change too much between steps k and $k+1$, and so $\mathbf{x}_{k+1}$ cannot be accurately inferred from $\mathbf{x}_k$ using the gradient alone. These errors then accumulate leading to the divergence seen in 1. The advantage to a large step size however is computation time; for a fixed simulation time, a larger time step requires fewer computations. If the time step is increased then the computation time decreases by the ratio of the previous and new time steps. So the time step should only be made as small as is necessary for the particular application.

1.1.4 Part (d)

Classical Runge-Kutta (RK4) is a fourth order integration method (while Euler was first order), it works by making four approximations $\{k_i\}_{i=0}^4$. The first approximation k_1 is simply the Euler approximation, k_2 is then the gradient evaluated a half time step along the k_1 predicted path, k_3 is then the gradient at a half time step along the k_2 predicted path, and finally k_4 is the gradient at a full time step along the k_3 predicted path. The next step is then calculated with a weighted sum of these four approximations. With the same f as previously, the full process is as follows

$$\mathbf{k}_1 = f(\mathbf{x}_k, t)$$

$$\mathbf{k}_2 = f(\mathbf{x}_k + \frac{\mathbf{k}_1}{2}\Delta t, t + \frac{\Delta t}{2})$$
$$\mathbf{k}_3 = f(\mathbf{x}_k + \frac{\mathbf{k}_2}{2}\Delta t, t + \frac{\Delta t}{2})$$
$$\mathbf{k}_4 = f(\mathbf{x}_k + \mathbf{k}_3\Delta t, t + \Delta t)$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)\Delta t$$

1.1.5 Part (e)

Again running the kinematic simulation with $\Delta t = 1, 0.5, 0.1, 0.01$ gives the plot shown in figure 2. RK4 integration is clearly more stable than the Euler method, with the plots following the general path even for a larger step size. One reason for this is the RK4 method computes $\mathbf{x}_{k+1}$ using both $\mathbf{x}_k$ and $\mathbf{x}_{k+0.5}$ so it is not relying on only the gradient of the initial state to infer the next. It's accuracy is also attributed to the fact that RK4 is a fourth order method, meaning its local error (error each step) scales with Δt^5, and its total accumulated error scales with Δt^4. The Euler method on the other hand is only a first order method, and so it's local error scales with Δt^2 and the total error scales with Δt, these errors are very significant in this case. It is worth noting however that the Euler method requires fewer steps, and thus is more computationally efficient than RK4. It also exactly predicts any linear path (as it is a first order method) and so for near linear applications using RK4 would be unnecessarily computationally expensive and Euler integration would be preferred.

1.1.6 Part (f)

We can simulate a 1Hz perception-action loop by letting $\omega = cos(\lfloor t \rfloor)$. This is as if the robot can only receive information about ω once per second. This means that $\omega = 1$ for a full second at the start of the simulation, meaning that even though there is a balance between clockwise and anticlockwise 'spin' of the robot across the full 10 seconds, there is an initial 'upwards' trajectory that can be seen in figures 3 and 4. Another issue that can arise is that if the time step is greater than the time period of the perception-action loop (1 second), we could under-sample the ω values leading to large error. We see this error in figure 3 for Δt equal to 2s and 1.5s, the RK4 integration gives a somewhat reasonable path for the $\Delta t = 1.5$ case, but runs into the same errors for $\Delta t = 2$, we expect RK4 to be somewhat more robust to these issues than Euler.

1.1.7 Part (g) [bonus]

We can attempt to resolve the under-sampling issue by adopting an adaptive integration method. The adaptive integration process performs an integration step as normal, and then compares this to the result had it computed the integration over two separate steps. If there is significant change, it will recursively

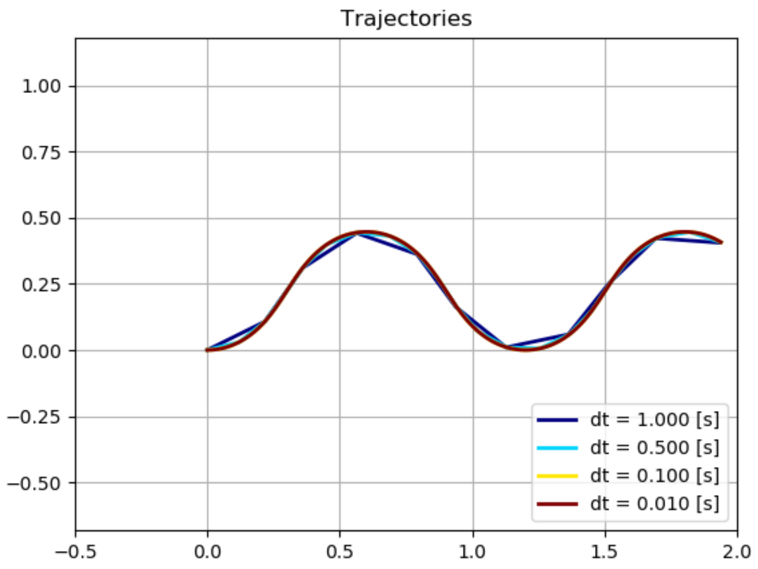


Figure 2: Kinematics with RK4 integration for a selection of time steps

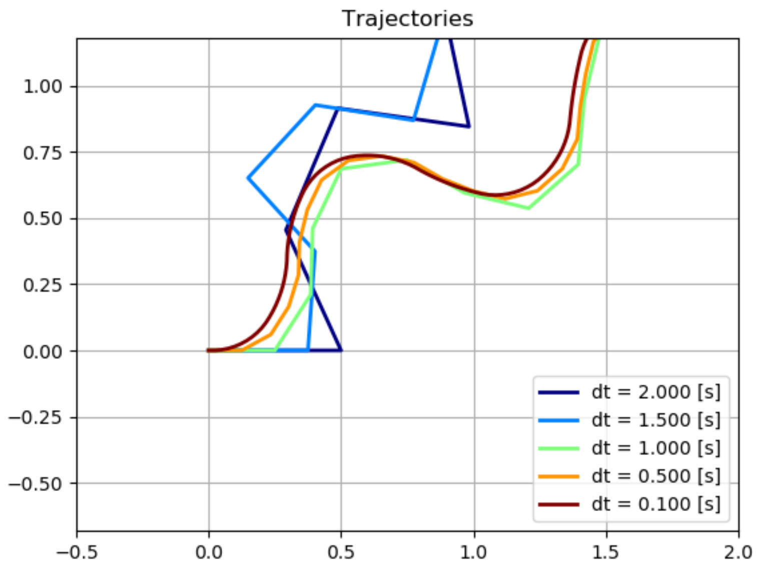


Figure 3: Kinematics with Euler integration and a perception-action loop of 1Hz for a selection of time steps

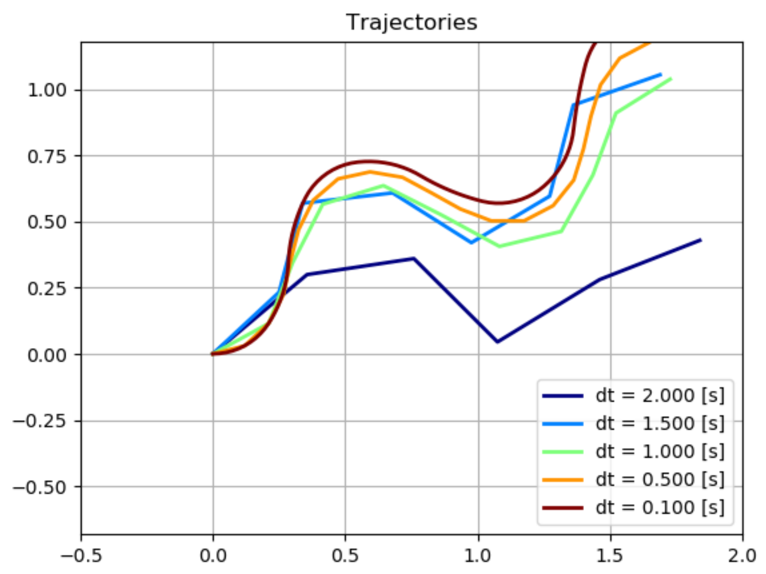


Figure 4: Kinematics with RK4 integration and a perception-action loop of 1Hz for a selection of time steps

compute those two steps separately (recursively meaning that if those subsections would be better computed in two parts it will do so again). This approach achieves two things

1. Ensures the step size is small in parts of the integration where this is necessary
2. Does not require a small step size in parts of the integration where this is less crucial, thus is computationally more efficient than just reducing the step size

This eliminates the effects of under-sampling because in the regions where under-sampling leads to a decrease in accuracy we simply use a smaller step size. Figures 5 and 6 show the effects of this approach.

1.2 Exercise 2

1.2.1 Part (a)

A Braitenberg controller is a memoryless controller with actuator outputs that are purely functions of the current input. One implementation of this is an artificial neural network with a single neuron for each output, leading to the output of actuator i, y_i, from sensor inputs $\mathbf{u}$ being given by

$$y_i = \phi(\mathbf{a}_i^T \mathbf{u} + c_i)$$

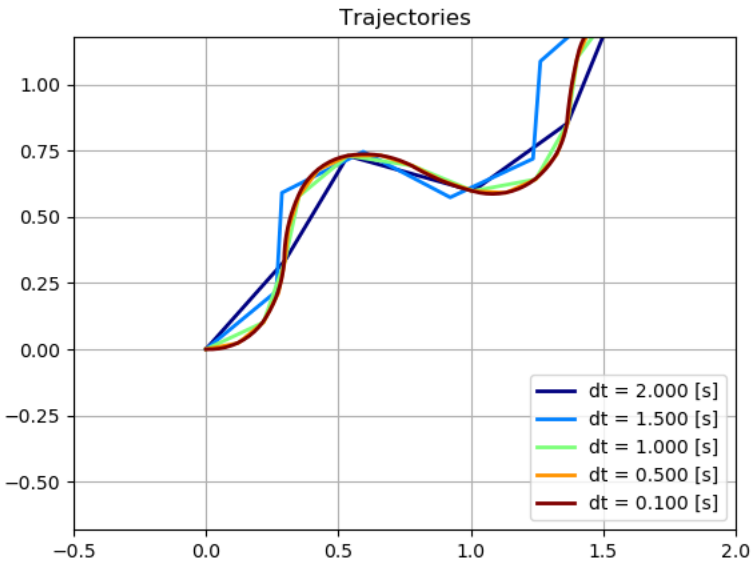


Figure 5: Adaptive integration used with the Euler method

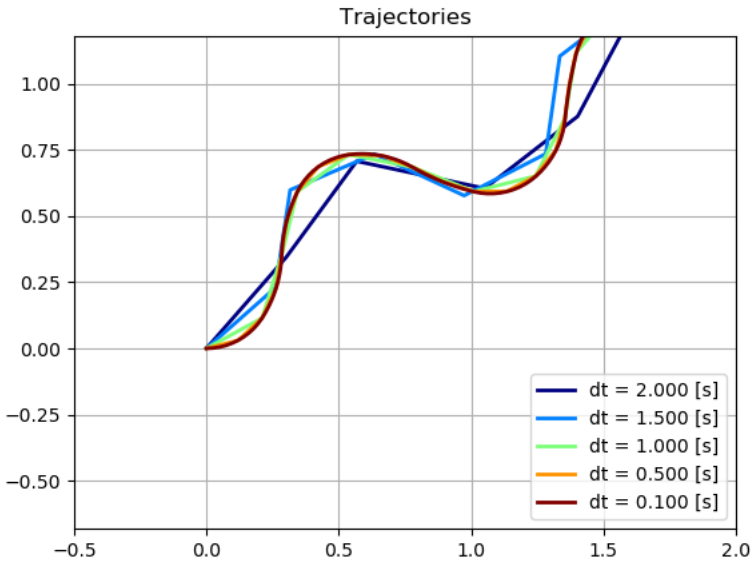


Figure 6: Adaptive integration used with RK4

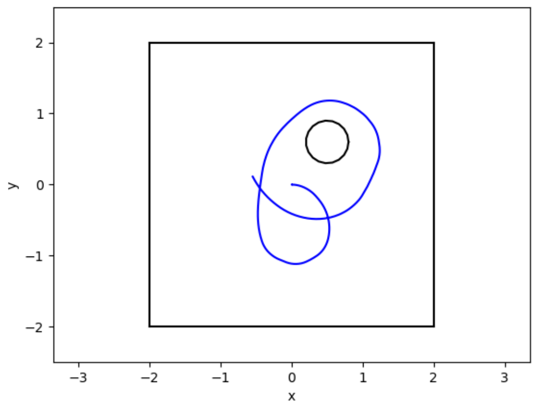


Figure 7: Path of robot with a Braitenberg controller

where c_i is a bias, and ϕ is a non-linear activation function. In this case $\phi(x) = \tanh(x)$, creating saturation of the actuator outputs. The sensor inputs are $\mathbf{u} = [\text{front, front\_right, front\_left, right, left}]^T$. This network will not be trained, but instead the weights set intuitively based on obstacle avoidance. I set the weights for the forward velocity output to be $\mathbf{a}_1 = [2, 0.5, 0.5, 0, 0]$ with a bias $c_1 = -2$, the idea being that if the robot is very close to an obstacle, the bias will cause it to move backwards, but generally the sensor values are large enough to encourage it to move forwards, also more emphasis is on the front sensor so as not to discourage the robot to move forwards if it was moving through a tight gap, which would lead to small values for front_left and front_right. The value of -2 is chosen for the bias because $\tanh(-2) \approx -1$ which saturates the activation function. I set the weights for the angular velocity to be $\mathbf{a}_2 = [0, 0.5, -0.5, 0.4, -0.4]$, creating an anti-symmetry between left and right sensors, so that if the robot is closer to an obstacle on the right it turns left and vice versa, there was no bias for the angular velocity (because we do not want the robot to spin by default). I also scaled the activation outputs by the maximum value i.e. $u = u_{max}\phi(x_1)$ and $\omega = \omega_{max}\phi(x_2)$ as the activation function saturates to a value of 1. The final path is shown in figure 7, we observe the robot avoiding obstacles as expected, and it always favours the areas of the arena with the most space in the forward 45^o direction, as these weights for the angular velocity node are the largest.

1.2.2 Part (b)

Alternatively to a Braitenberg controller, where the outputs are continuous over an input domain, we can use `if-else` statements to create a state-based model of the robot, with different actions in different scenarios. The code for this is shown in figure 9, there are three main states for the robot. The first state is if

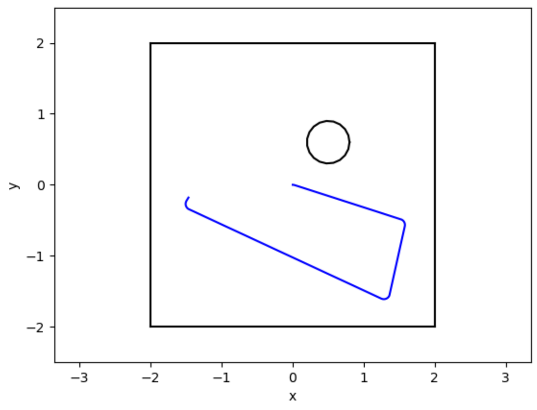
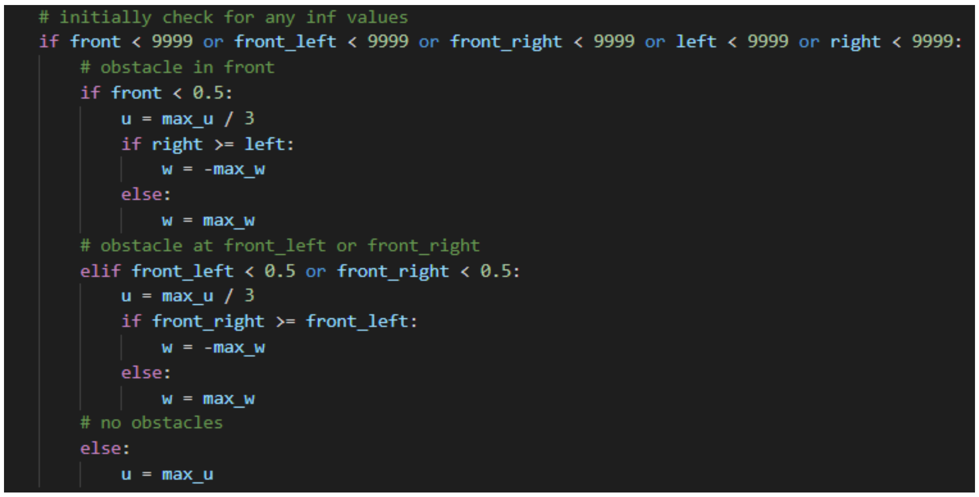


Figure 8: Path of robot with a rule based controller



```
# initially check for any inf values
if front < 9999 or front_left < 9999 or front_right < 9999 or left < 9999 or right < 9999:
    # obstacle in front
    if front < 0.5:
        u = max_u / 3
        if right >= left:
            w = -max_w
        else:
            w = max_w
    # obstacle at front_left or front_right
    elif front_left < 0.5 or front_right < 0.5:
        u = max_u / 3
        if front_right >= front_left:
            w = -max_w
        else:
            w = max_w
    # no obstacles
    else:
        u = max_u
```

Figure 9: Snapshot of rule based code

it has an obstacle in front of it (front < 0.5), in this case the robot reduces its speed and veers to the left or right depending on which has more space (based on its left/right sensors). The second case is if front > 0.5, but there is an obstacle nearby in the front_left or front_right direction, the robot veers to the left/right in this situation, depending on where said obstacle is, in the scenario where the robot is moving through the middle of a narrow gap this would lead to a repeated left/right motion as it moves through the gap. Finally, if there are no apparent obstacles in front of the robot, it will simply move forwards. The final path is shown in figure 8, we see the robot initially veer away from the cylinder, and then move towards several walls, veering away from them each time - this is the behaviour we expect.

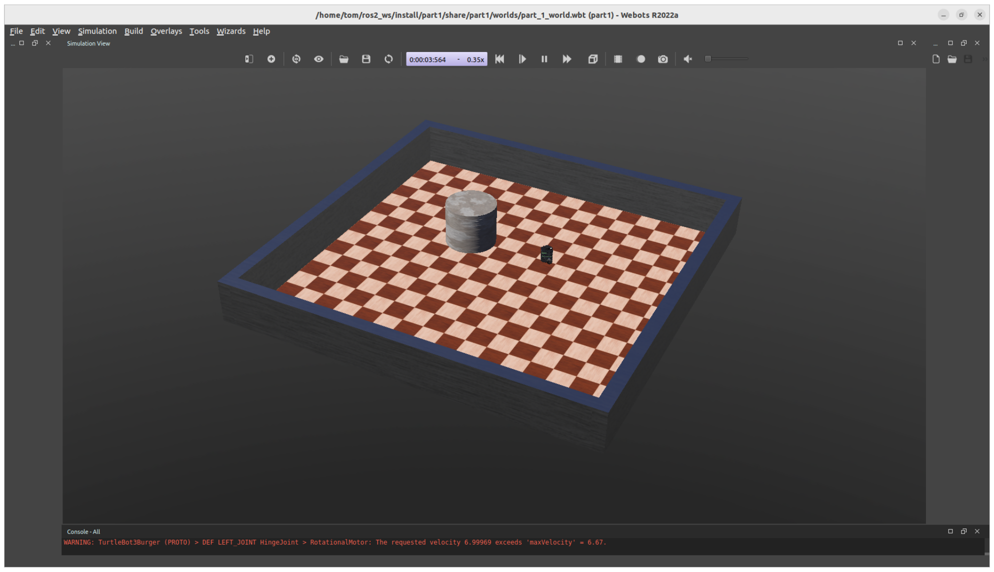
1.2.3 Part (c)

The Braitenberg controller is somewhat robust, and follows a smooth trajectory, it veers way from obstacles in time before collisions, and even moves backwards when necessary. There are scenarios however where it can get stuck, such as if it approaches an obstacle 'head-on' and has equal sensor values for left and right and also for front_left and front_right, as it will not turn due to the symmetry of the weights for the angular velocity, and it continues to move backwards and forwards around the point where the bias and the network output balance out. A potential solution to this issue would be a small bias on the angular velocity, or to add some noise to the anti-symmetric weights. This would however, make the robot always favour a particular direction of turning - so may not be desirable.

The rule based controller on the other hand is more robust, it very rarely ends up stuck at a particular point in the arena. However its movement is less smooth, this is due to the discontinuous nature of the transfer function from sensor input to actuator output. A particular scenario when the movement isn't smooth is when it is moving through a tight gap, it rotates left and right repeatedly and while it does move through the gap properly it does so with a jagged path due to its repeated switching between two states.

1.2.4 Part (d)

The Braitenbeg controller has a continuous transfer function from sensor input to actuator output, and as such it is not very sensitive to noise, there will be no drastic differences in output for small input changes. However the rule-based controller has lots of discontinuity and is thus sensitive to noise, changes in input values could lead to completely different behaviours to what is expected when readings are near to the discontinuous regions. Generally however, the behaviour we want for the robot based on general decisions based on vicinity to objects, and so were the behaviour state to change due to noise, it would only be in areas of the arena where that behaviour may well be applicable anyway. So perfect sensors would not have too much of an effect on the robot behaviour.


/home/tom/ros2_ws/install/part1/share/part1/worlds/part_1_world.wbt (part1) - Webots R2022a
File Edit View Simulation Build Overlays Tools Wizards Help
Simulation View
0:00:03:564 - 0.35x
Console - All

```
WARNING: TurtleBot3Burger (PROTO) > DEF LEFT_JOINT HingeJoint > RotationalMotor: The requested velocity 6.99969 exceeds 'maxVelocity' = 6.67.
```

Figure 10: Webots setup

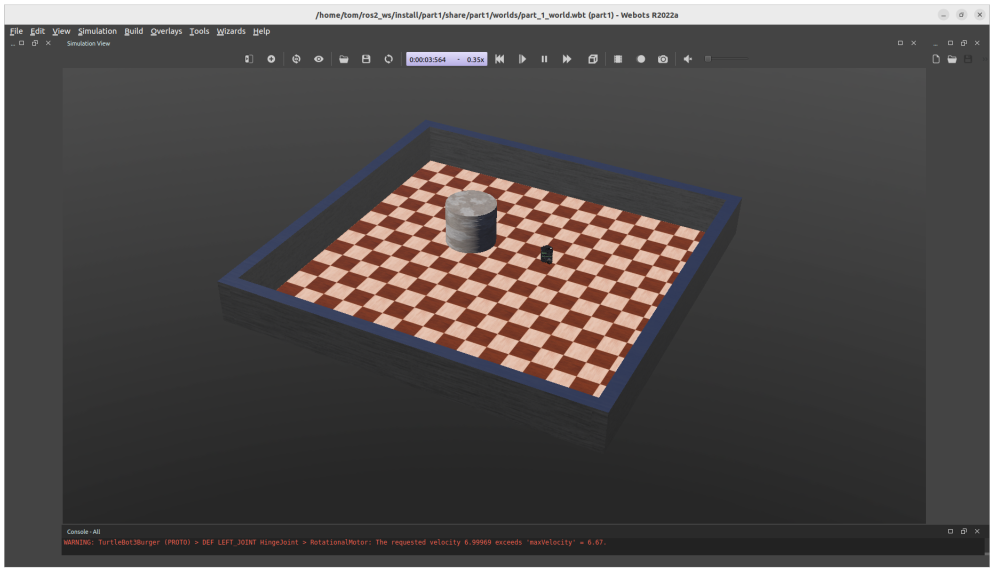
1.3 Exercise 3

1.3.1 Part (a)

For this exercise a Braitenberg controller is used, with the same algorithm and parameters as in Exercise 2. The webots setup is shown in 10

1.3.2 Part (b)

For a particle filter, we want to initialise particles randomly in the arena, but we must ensure that they are valid (i.e. they are inside the arena, and not in any obstacles). This is achieved in the `is_valid` method in the `Particle` class. There are two checks to ensure a particle is valid, first if the particle is within the cylinder, we compute the relative position to the centre of the circle, find its magnitude, and check if it is less than the circle radius. $||\mathbf{r}_{particle} - \mathbf{r}_{cylinder}|| < r_{circle} = 0.3$. If this condition is met, we return `False` from the method. The next check is if the particle is within the arena boundaries, the x and y coordinates can be checked separately for the condition $|x| < x_{wall}$, where x_{wall} is the WALL_OFFSET variable. If this condition is met we return `False`. If neither of the previous conditions are met, the particle is valid and we return `True`. Now in the `Particle` constructor we can use a while loop to generate a particle in a random position in the arena, and check if it is valid, if so we do nothing, but if not we generate the particle in a new position. The likelihood of a randomly initialised particle being invalid is small, so it is unlikely this loop will run any more than twice, so more sophisticated approaches are not


/home/tom/ros2_ws/install/part1/share/part1/worlds/part_1_world.wbt (part1) - Webots R2022a
File Edit View Simulation Build Overlays Tools Wizards Help
Simulation View
0:00:03:564 - 0.35x
Console - All

```
WARNING: TurtleBot3Burger (PROTO) > DEF LEFT_JOINT HingeJoint > RotationalMotor: The requested velocity 6.99969 exceeds 'maxVelocity' = 6.67.
```

Figure 11: Particles after completion of part (c)

required.

1.3.3 Part (c)

We want to set up a motion model for the particles, this is done in the `move` method. The method takes a `delta_pose` parameter, which is a small movement vector in the local reference frame for the robot, $\Delta = [udt, 0, \omega dt]$. In order to convert this to the particles' global reference frame, we need to know the robot's absolute orientation θ_{true}, which we can get from the `odom` topic. We also want to add some noise to our motion model, so that we are not simply relying on particles being initialised in the correct spot, but they can explore the space to converge on a good value for the robot pose. We use a zero mean Gaussian with standard deviation 10% of the robots' velocity, or angular velocity, which we can again get from the `odom` topic. The update equations in `move` are therefore

$$x_{k+1} = x_k + \Delta x \sin(\theta_{true}) + e_1$$

$$y_{k+1} = y_k + \Delta x \cos(\theta_{true}) + e_2$$

$$\theta_{k+1} = \theta_k + \Delta\theta + e_3$$

Where $e_1, e_2 \sim N(0, (0.1u)^2)$ and $e_3 \sim N(0, (0.1\omega)^2)$.

1.3.4 Part (d)

Were we to start the robot at (1,1) instead of (0,0) we would find that our `is_valid` function would run into issues, because the particle position is relative to the robot as the origin, so particles would be deemed out of the arena and thus invalid when they are not. To combat this we could include in offset in the `is_valid` function, or indeed as an attribute in the `Particle` class.

1.3.5 Part (e)

We have randomly initialised particles, that move in the same way as the robot, but we need to determine which particles are most similar to the true robot pose. This requires a correction step where we assign probabilistic weights to each particle, we do this in the `compute_weight` method. We have measurements from the five sensors from the robot x_t, and the same measurements from the particles z_t, and we want to compute $p(z_t|x_t)$, the probability that the particle sensor values came from the correct position, given the robot sensor values. To do this we assume a Gaussian distribution of sensor values, center it on the robot sensor values x_t and compute the Gaussian pdf at z_t. We multiply this over all 5 sensors to compute $p(\{z_t^{(i)}\}_{i=0}^5|\{x_t^{(i)}\}_{i=0}^5)$ (with some independence assumptions). We can then set this probability as the weight for that particle (note that this is not normalized, but it does not matter). The only caveat is that we set any invalid particles (using the `is_valid` method) to have a weight of zero, so they are removed in the re-sampling step.

Now that the particles have likelihood weights assigned, we can remove unlikely particles in a re-sampling step. We take the weights $\{w_i\}$ of all particles and compute $\mu = E[w]$ and $\sigma = \sqrt{Var[w]}$. Each frame every particle with a weight less than $\mu - \eta\sigma$ where η is a hyperparameter we can tune, is removed. When we remove a particle we re-initialise it at the same point as another particle, we do this randomly weighted on the weights of the other particles. This leads to the particles gradually converging to high-probability poses with regard to the robots' true position.

1.3.6 Part (f)

Repetitions of the experiment lead to consistent convergence of the particle filter, although the convergence is sometimes slow and it struggles to localise when the robot is near an obstacle. Convergence can be slow when there are many particles that are not being re-sampled as they have high weights, but they are not quite in the right position. This means that when low weight particles are re-sampled there is a large range of parameters that they could be re-sampled to, and so it takes a while for them all to converge to the correct value. However quick convergence can lead to convergence on the wrong value, the particles have not had time to explore the space to have a very high probability particle in the correct pose for the robot. There is a balance required for good localisation.

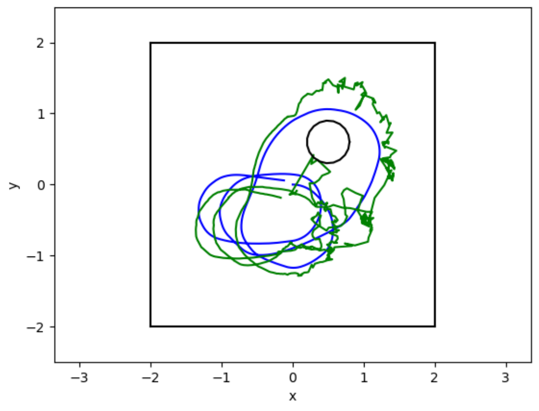


Figure 12: Trajectory of robot with localisation output

1.3.7 Part (g)

Moving the robot while the simulation is paused and replaying, generally means that the particles follow the robot and are still localising properly. However sometimes there can be offsets created by doing this, and if the particles are already converged to a point it can be difficult for them to find the true optimum again. This can be rectified by finding a correct balance between localisation accuracy and convergence speed. This is determined by the hyperparameters, most notably the η value for re-sampling and the noise in the movement function. Too much noise and the convergence will be slow because the particles move around a lot and so can move away from high probability poses easily, however if it is too small the particles are not exploratory enough and it can converge to an incorrect value. I used 30% of the velocity and angular velocity for the standard deviation of movement noise to balance convergence speed and accuracy. Similarly the η value being too large leads to too few re-samples and low-probability points can alter the general movement of the particles, however if it is too high there is not enough time for particles to explore and end up near to the true scope before they are re-sampled, which can lead to convergence to incorrect values. I chose $\eta = \frac{1}{2}$ to find this balance.

1.3.8 Part (h)

The trajectory of the localisation is shown in figure 13, and the errors are shown in figure **??**. The trajectory shows the localisation converges near the robots true position and follows it as it moves through the arena. The error shows that generally the error is less than 0.5m, except when the robot moves near to the cylinder, this is because many particles become invalid and are thus re-sampled, generally reducing the localisation accuracy. A potential solution for this would

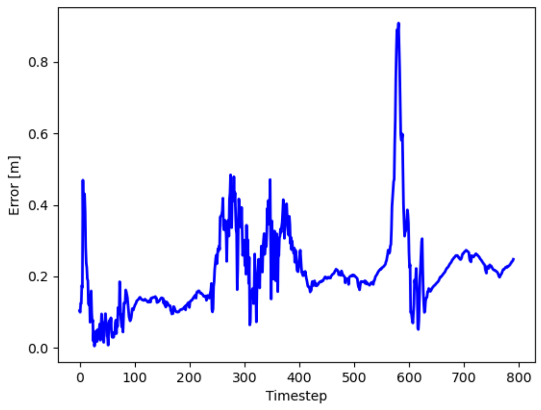


Figure 13: Localisation error

be to only remove particles that are invalid with some probability, as given we expect some error in this localisation we could afford some particles being in invalid positions if it will help the stability of the overall localisation.

1.3.9 Part (i)

A Kalman filter is a potential improvement on our particle filter, most notably because we do not have to deal with a large number of particles and instead perform matrix multiplications which increases our computational efficiency. Another advantage of not having to deal with particles is that we don't run into re-sampling issues, we are dealing with a probability distribution on the position of our robot that is computed using a belief prior, and a measurement likelihood. We can deal with values being invalid (outside of the arena) post-computation by truncating the posterior and re-normalising, this means there will be no issues with particle collapse! The disadvantage of a Kalman filter is that we make Gaussian assumptions on both our prior belief, and our sensor readings. If we have no real knowledge of the sensor noise, then a Gaussian is a reasonable guess, however if we knew the sensor noise to be another non Gaussian distribution then we would not be able to take advantage of that information, because we would not be able to find analytic solutions (whereas we could in a particle filter, because we are dealing with numerics and don't require closed-form solutions).