Mobile Robot Systems
Assignment 1

Zach Lambert Pembroke College Feb 2021

1 Exercise 1 - Kinematic simulator

(1a) Differential drive equation of motion

Robot state $\boldsymbol{x} = \begin{bmatrix} x & y & \theta \end{bmatrix}^T$, for position (x, y) and yaw θ.

The control inputs are linear velocity $u = 0.25$ and angular velocity $\omega = \cos(t)$, which give the equation of motion:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} u\cos(\theta) \\ u\sin(\theta) \\ \omega \end{bmatrix}$$

(1b) Euler's method

Euler's method uses a first-order approximation of the series expansion of $\boldsymbol{x}(t)$:

$$\boldsymbol{x}(t+\delta t) = \boldsymbol{x}(t) + \delta t \left.\frac{\partial \boldsymbol{x}}{\partial t}\right|_{t,\boldsymbol{x}(t)} + O(\delta t^2)$$

$$\boldsymbol{x}(t+\delta t) \approx \boldsymbol{x}(t) + \delta t \left.\frac{\partial \boldsymbol{x}}{\partial t}\right|_{t,\boldsymbol{x}(t)}$$

which introduces a truncation error $O(\delta t^2)$.

(c) Effect of step-size on accuracy of simulation with Euler's method

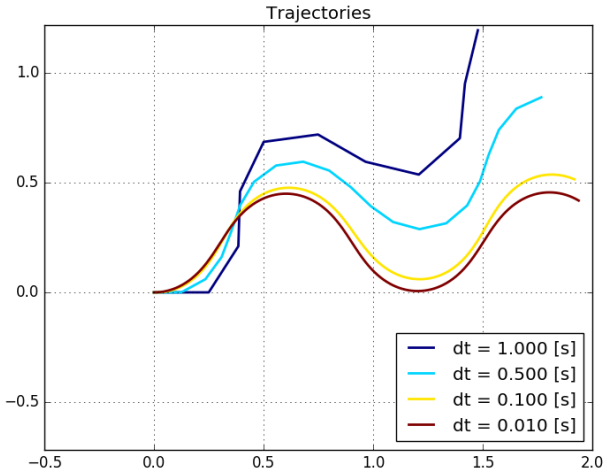

Figure 1: Simulation results with Euler's method for various step sizes.

Because the truncation error of Euler's method is $O(\delta t^2)$, accuracy improves with decreasing step size, as shown in Figure 1. However, smaller step size increases computation time. Therefore, step size should only be selected as small as is necessary. When the signals involved are changing more slowly, with smaller frequency components and smaller higher order derivatives, step sizes can be smaller.

(1d) The Runge-Kutta method

The Runge-Kutta method (RK4) is a fourth-order numerical integration method. Euler's method is a first-order method, so it takes into account first-order terms, but truncates second-order and higher terms, giving a local truncation error $O(\delta t^2)$. RK4 takes into account all terms up to and including fourth-order terms, giving a local truncation error $O(\delta t^5)$.

RK4 uses the following approximation:

$$\boldsymbol{x}(t+\delta t) \approx \delta t \times \frac{1}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)$$

where

$$\begin{aligned}
\boldsymbol{k}_1 &= \boldsymbol{f}(t, \boldsymbol{x}) \\
\boldsymbol{k}_2 &= \boldsymbol{f}(t + \frac{\delta t}{2}, \boldsymbol{x} + \boldsymbol{k}_1 \frac{\delta t}{2}) \\
\boldsymbol{k}_3 &= \boldsymbol{f}(t + \frac{\delta t}{2}, \boldsymbol{x} + \boldsymbol{k}_2 \frac{\delta t}{2}) \\
\boldsymbol{k}_4 &= \boldsymbol{f}(t + \delta t, \boldsymbol{x} + \boldsymbol{k}_3 \delta t)
\end{aligned}$$

and $\boldsymbol{f}(t, \boldsymbol{x})$ is the derivative of x with respect to time, evaluated for a given time t and state $\boldsymbol{x}$:

$$\boldsymbol{f}(t, \boldsymbol{x}) = \left.\frac{\partial \boldsymbol{x}}{\partial t}\right|_{t,\boldsymbol{x}}$$

(1e) Comparing Euler's method and RK4

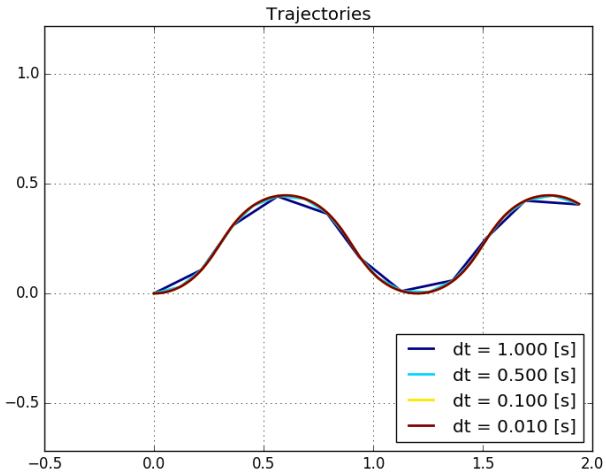

Figure 2: Simulation results with RK4 for various step sizes.

RK4 is far more accurate than Euler's method for the same fixed time-step, as shown in Figure 2. A time step of $\delta t = 1$ with RK4 gives similar accuracy to $\delta t = 0.01$ with Euler's method, since it takes into account the higher derivatives.

(1f) Simulating a perception-action loop running at 1Hz

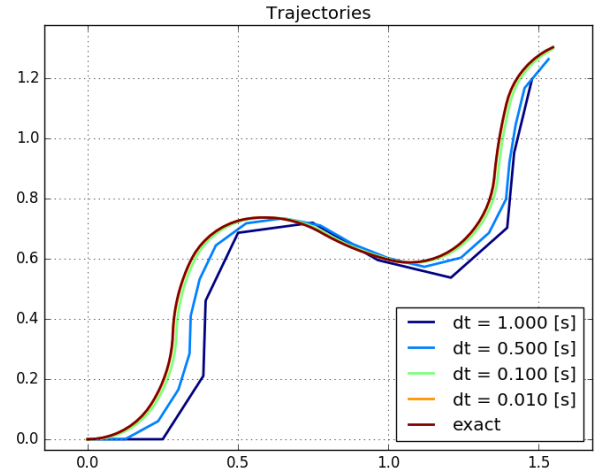

Figure 3: Simulation results with $u = \cos(\lfloor t \rfloor)$ for a 1Hz perception-action loop, using Euler's method.

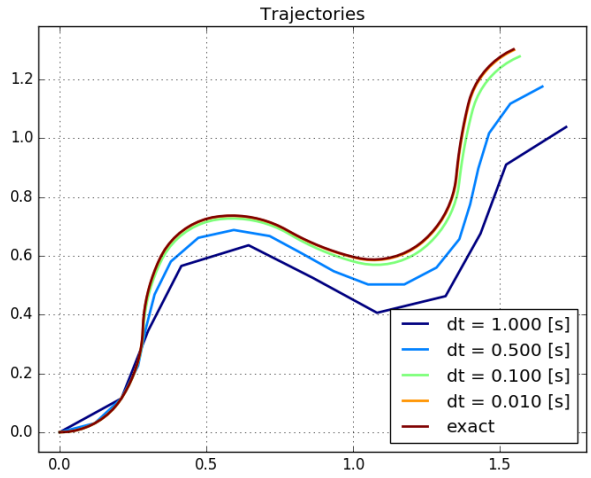

Figure 4: Simulation results with $u = \cos(\lfloor t \rfloor)$ for a 1Hz perception-action loop, using RK4.

Figure 3 and Figure 4 show the simulation with a perception-action loop running at 1 Hz. The exact solution is shown for comparison. With $\omega = \cos(t)$, the angular velocity varies continuously and with relatively small state derivatives. However, with $\omega = \cos(\lfloor t \rfloor)$, there is a step change in angular velocity at $t = 1, 2, \ldots$, which gives large derivatives at these points, introducing errors.

This can be seen in the RK4 simulation, where the simulation results deviate at about $x = 0.25$ for $t = 1$, where the step change in ω causes the larger step sizes to introduce errors.

(1g) Adaptive numerical integration methods

Previous methods used a fixed step size δt. Adaptive methods vary the step size based on the precision required for a particular point in the simulation. When a small step size isn't needed, if velocity is zero for example, step size can be increased to reduce simulation time. When the velocity is changing rapidly, higher precision is required to maintain the same accuracy, so step size is reduced.

Adaptive methods estimate the truncation error for a given step size δt and iteratively update δt until a suitable truncation error is achieved. This can be applied to Euler's method[1] and RK4[2].

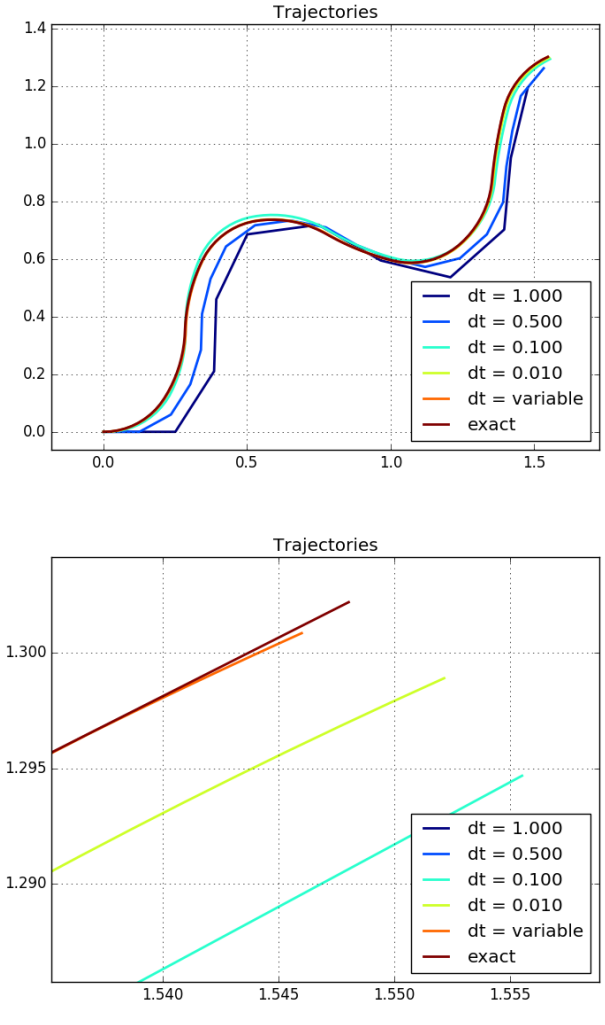

Figure 5: Simulation results using Euler's method and an adaptive step size, with an overall plot and focus on the end states.

Figure 6: Simulation results using RK4 and an adaptive step size, with an overall plot and focus on the end states.

Figure 6 shows the performance of Euler's method with an adaptive step size. The variable step size is highly accurate, reaching practically the same end state, while being just as fast as the fixed step sizes shown.

Small step sizes were required when there were step changes in the angular velocity, at $t = 1, 2, \ldots$, allowing it to reach step sizes as small as 10^{-6} to maintain accuracy, while using larger step sizes for other parts of the simulation to save time.

To give comparable accuracy with a fixed step size, a fixed step size would have to be selected as the minimum used by the adaptive integrator and would be much slower.

Using an adaptive step size with RK4 performs even better, with the final state coinciding with the exact final state. It was more accurate than using a fixed step size of $\delta t = 0.001$, while being much quicker to simulate.

Again, the smaller step sizes were required when the angular velocity underwent step changes.

2 Exercise 2 - Obstacle avoidance

(2a) Implementing a Braitenberg controller for obstacle avoidance

The Braitenberg controller implemented sets the linear velocity u (m/s) and angular velocity ω (rad/s) as linear combinations of the inverses of the distance measurements. These are labelled $d_0, \ldots, d_4$ corresponding to the left, front left, front, front right and right sensors.

$$\boldsymbol{v} = W\boldsymbol{x} + \boldsymbol{v}_0$$

for

$$\boldsymbol{x} = \begin{bmatrix} \frac{1}{d_0} & \cdots & \frac{1}{d_4} \end{bmatrix}^T \quad \boldsymbol{v} = \begin{bmatrix} u & \omega \end{bmatrix}^T$$

The controller should:

- Move at $u = 0.2$ m/s if there are no obstacles in front, and slowing down to $u = 0$ at 0.5m from a wall.
- Rotate away from obstacles on the left or right, moving away more quickly from obstacles detected by the front left and front right sensors.

This used:

$$\boldsymbol{v}_0 = \begin{bmatrix} 0.2 & 0 \end{bmatrix}^T$$
$$W = \begin{bmatrix} 0 & 0 & -0.1 & 0 & 0 \\ -0.5 & -1 & 0 & 1 & 0.5 \end{bmatrix}$$

With no obstacles in front, $d_2 = \infty$, $x_2 = 0$ so the robot moves with $u = 0.2$. If $d_2 = 0.5$, $x_2 = 2$, so $v = 0.2 - 2 \times 0.1 = 0$.

The weights for ω cause the robot to rotate away from the side with smaller distances and therefore larger inputs x_i, with larger weights for the front right and front left sensors.

The inverse distances were used instead of a sigmoid function since this removed the sensitivity to hyperparameters (W), which only affected how close the robot had to get to an obstacle before slowing down or turning away from it.

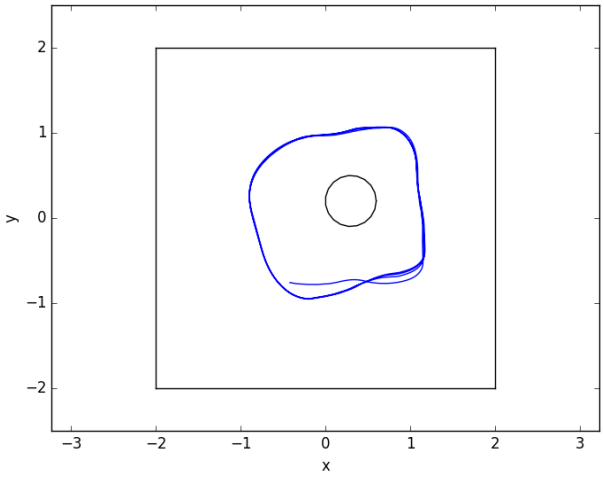

Figure 7: Robot trajectory when using the Braitenberg obstacle avoidance controller.

Figure 7 shows a trajectory of the robot using the Braitenberg controller. It follows a closed loop around the environment avoiding obstacles and trying to stay equidistant from obstacles on either side.

(2b) Implementing a rule-based controller for obstacle avoidance

The rule-based controller uses similar concepts to the Braitenberg controller, with with adjustments.

Firstly, $u = 0.2$, unless the front distance is less thatn $1m$, where it starts to reduce the velocity until $u = 0$ at $d_2 = 0.5$. This allows the robot to move more quickly in general, while stopping before hitting obstacles in front.

Secondly, only the front and left sensors are taken into account to set angular velocity, allowing smoother motion for maintaining an equal distance between obstacles. However, if the front right or front left distances go below a threshold (in this case, 2m), then ω was set to move away from this.

Finally, when the front distance is small and ω is small, this means the robot has reached a dead end. In this case, the controller offsets ω if it is exactly equal to zero, and scales the angular velocity by the inverse of the front distance. This allows it to rotate out of a dead end and start moving back the way it came.

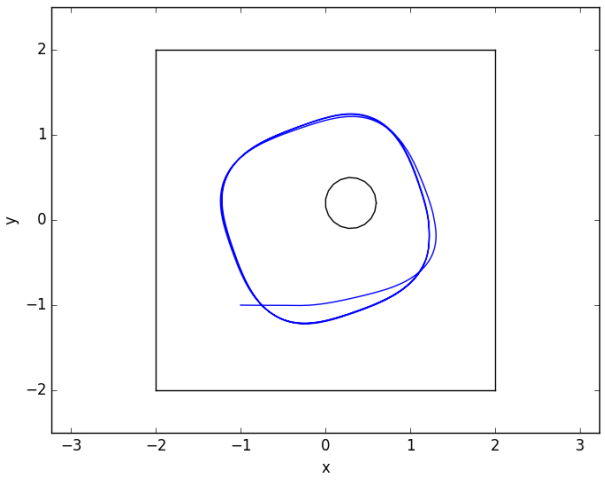

Figure 8: Robot trajectory when using the rule-based obstacle avoidance controller.

Figure 8 shows the trajectory when using a rule-based controller. The trajectory is similar to the Braitenberg controller, but is smoother.

(2c) Comparing performance of the controllers

Both controllers used similar concepts, so behaved similarly and were robust to changes in the environment.

The only exception was dead-ends, where the Braitenberg controller would come to a stop, whereas the rule-based controller would move back the way it came.

(2d) Effect of sensor noise on performance

To edit the sensor noise, the sensor noise standard deviation for the laser plugin in the turtlebot3 gazebo urdf was edited. Because the distance measurements are averaged into five bins, this smoothed out the noise, making small noise insignificant. Even at significant noise levels, both controllers were relatively unaffected, since neither used sharp decision boundaries. However, around a standard deviation of 0.5m, the distance measurements became unreliable, causing the controllers to fail.

3 Exercise 3 - Localisation

(3a) Gazebo setup

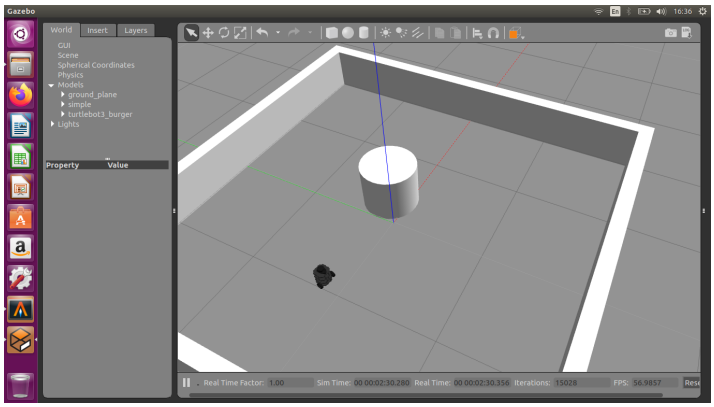

Figure 9: Screenshot of setup for robot simulation with Gazebo for localisation.

(3b) Implementing *Particle.___init___*

A particle is initialised by setting its weight to $w = 1$ and randomly selecting a valid start pose. Sampling from uniform distributions:

$$
\begin{aligned}
x &\sim U[-W + R, W - R] \\
y &\sim U[-W + R, W - R] \\
\theta &\sim U[-\pi, \pi]
\end{aligned}
$$

where the robot has radius R, and the environment has a size $2W \times 2W$ centered at $(0, 0)$.

Additionally, the position (x, y) is checked for collision with the cylinder and resampled if there is a collision. A collision occurs when the distance between the cylinder and robot is less than the sum of the robot radius and cylinder radius.

(3c) Implementing *Particle.move*

The pose at time-step t is represented by the vector:

$$
\boldsymbol{x} = \begin{bmatrix} x & y & \theta \end{bmatrix}^T
$$

The discrete-time state equation for the robot is:

$$
\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{\delta x}_k + \boldsymbol{w}_k
$$

$\boldsymbol{w}_k$ is white noise with standard deviation equal to 30% of $\boldsymbol{\delta x}$ and $\boldsymbol{\delta x}$ is determined by the linear velocity u_k and

angular velocity ω_k:

$$\begin{aligned}\boldsymbol{\delta x}_k &= \delta t \times R(\theta_k)\boldsymbol{v}_k \\ \boldsymbol{v}_k &= \begin{bmatrix} u_k & 0 & \omega_k \end{bmatrix}^T \\ R(\theta_k) &= \begin{bmatrix} \cos(\theta_k) & -\sin(\theta_k) & 0 \\ \sin(\theta_k) & \cos(\theta)k) & 0 \\ 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

The *Particle.move* function samples $\boldsymbol{x}_{k+1}$ according to a normal distribution with mean $\boldsymbol{x}_k + \boldsymbol{\delta x}_k$ and the covariance of $\boldsymbol{w}_k$. A 30% standard deviation was chosen instead of the suggested 10% as this was found to perform better.

To allow the particle filter to handle the robot being kidnapped, or starting in an unknown, a probability of kidnap $\epsilon = 0.2$ was used. With ϵ probability, instead of using sampling x_{k+1}, the new particle state was randomly sampled over all valid states. A probability of 0.2 was chosen since it meant the particle filter more quickly found the valid state. When a cluster formed around the valid state, when particles were moved to a random state, they would have a low weight and not be re-sampled.

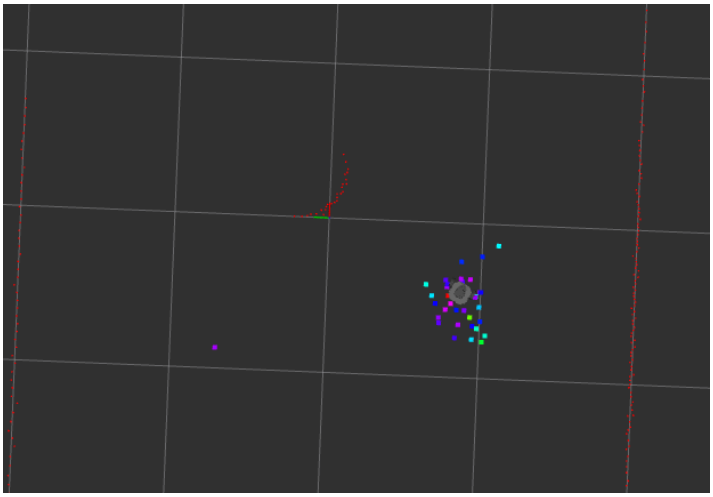

Figure 10: Screenshot of setup for visualising the robot and particle filter point cloud using Rviz.

Figure 10 shows a screenshot of the localisation in Rviz, with a cluster of particles formed around the robot.

(3d) Implementing *Particle.compute_weight*

The distance sensor readings y follow a normal distribution with mean equal to the true distance, μ and standard deviation $\sigma = 0.8$. However, if the measurement is above 3.5, this will return an infinite reading.

For a set of measurements $\{y_i\}_{i=1}^N$, each contributes w_i to the weight:

$$w = \prod_{i=1}^{N} w_i$$

If $y \neq \infty$, weight is assigned to the probability density:

$$w_i = p(y_i) = \mathcal{N}(\mu_i, \sigma_i)$$

However, if an infinite measurement $y_i = \infty$ is received, weight is assigned to probability *mass* that $y_i > 3.5$:

$$w_i = p(y_i > 3.5) = p\left(z > \frac{3.5 - \mu_i}{\sigma_i}\right) = 1 - \phi\left(\frac{3.5 - \mu_i}{\sigma_i}\right)$$

where $\phi(z)$ is the cumulative density function for the unit normal distribution.

The function uses the ray trance function to compute the expected distance for the current state estimate.

(3e) Localisation convergence

In general, localisation always succeeds. However, due to simple geometry of the environment, several trajectories of the robot give the same distance measurements. For example, if moving towards a wall in any corner region of the box.

However, if a cluster formed about a similar trajectory, it would break up whenever unexpected distance measurements were received, whereas a cluster that formed around the true state wouldn't receive unexpected distance measurements and would persist.

Once one particle was randomly placed near the correct state, it would be assigned a high weight, and particles would quickly accumulate.

(3f) Robustness to kidnapping

Kidnapping has the same effect as initialising the robot with unknown starting state. So long as the kidnap probability was sufficiently high, once a particle found a nearby state, a cluster would form around it, converging to the corrects state.

(3g) Particle filter performance

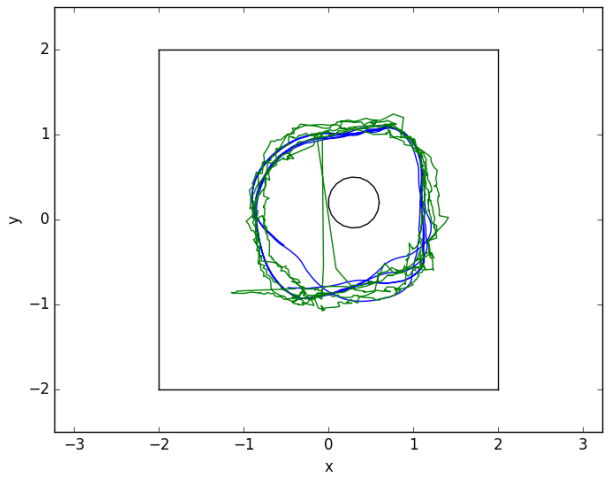

Figure 11: Trajectory of state estimate and true estimate for the particle filter.

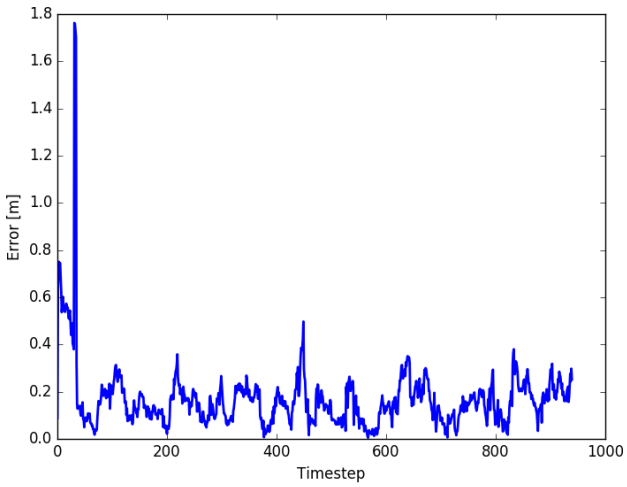

Figure 12: State estimate error over time for the particle filter.

Figure 11 shows the trajectory of the state estimate and true state. At the start, the state estimate has larger errors, while still searching for the correct solution. However, once converged on the correct solution, it would reliably maintain this state, with an average error of about 0.2m, as shown in Figure 12.

(3h) Comparing the particle filter to an Extended Kalman Filter

The main advantage of a Kalman filter is increased resolution and decreased computation time. A single state estimate is maintained, which is updated by combining the prior estimate based on previous measurements and the likelihood of the new state based on new measurements. This gives the optimal state estimation for a linear system with Gaussian noise, which performs well so long as the system model is accurate.

However, the Kalman filter relies on finding a new state estimate from the old state estimate, so can't handle starting from an unknown state.

Perhaps a practical localisation method would use a Kalman filter after converging to a solution with small errors, while using a particle filter when it needs to do a wider search of possible states when the current state is unknown or inaccurate.

4 References

[1] University of British Columbia (2021)
Variable step size methods, Available at:
`http://www.math.ubc.ca/~feldman/math/vble.pdf`
(Accessed: 01 Feb 2021)

[2] Oklahoma State University (2021)
Runge-Kutta Method, Available at:
`https://math.okstate.edu/people/yqwang/teaching/math4513_fall11/Notes/rungekutta.pdf`
(Accessed: 01 Feb 2021)