University of Colorado Boulder

Department of Aerospace Engineering Science

Single and Multi-Agent Localization in Known Environments

Komal Porwal, Joey Caley

Instructor: Dr. Nisar Ahmed

A report submitted in partial fulfilment of the requirements of
the University of Colorado Boulder for the degree of
Master of Science in *Aerospace Engineering*

May 7, 2025

Declaration

I, Komal Porwal and Joey Caley, of the Department of Aerospace Engineering, University of Colorado Boulder, confirm that this is our own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced.

We give consent to a copy of my report being shared with future students as an exemplar.

We give consent for my work to be made available more widely to members of CU Boulder and public with interest in teaching, learning and research.

Komal Porwal, Joey Caley
May 7, 2025

Abstract

Accurate localization is essential for autonomous robotic systems operating in environments without GPS or external tracking technology that identifies and communicates the robot's position. This problem can be solved using information from the environment or using shared information from other robots operating in the environment. This paper explores both single-agent and multi-agent localization strategies using probabilistic inference methods grounded in Bayesian reasoning. We implement a particle filter based on a dynamic Bayesian network (DBN) for a single robot navigating a 2D space with known landmarks. Building on this, we extend the framework to a multi- agent setting where robots share relative measurements using message passing and Gaussian Belief Propagation (GBP), with select agents possessing global state estimates to serve as anchors. Through simulations, we evaluate the performance, accuracy, and scalability of these approaches, highlighting trade-offs and potential advantages of decentralized swarm-based localization systems.

Acknowledgment

We would like to express our sincere gratitude to Professor Nisar Ahmed for his guidance and support throughout this project. His insights and feedback were invaluable to our work. We also acknowledge the University of Colorado Boulder for providing the resources and environment necessary for our research.

Chapter 1

Introduction

Autonomous robotic systems rely heavily on accurate localization to navigate and operate effectively in diverse environments—from structured indoor settings like warehouses and hospitals to unstructured terrains on planetary surfaces. In the provided information about their position and orientation, robots must estimate their own positions using internal sensing and probabilistic reasoning. This makes localization a critical and challenging component of autonomy, especially when measurements are noisy, nonlinear, and dynamically evolving over time. Bayesian reasoning offers a powerful framework for addressing this uncertainty, enabling robots to infer their states through models such as dynamic Bayesian networks (DBNs) Dagum et al. (1992).

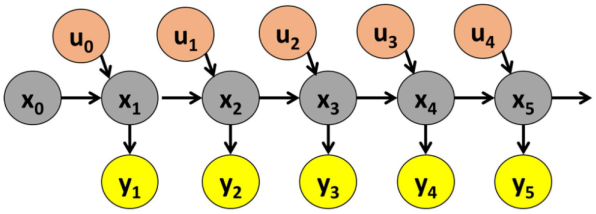

Figure 1.1: An Example of DBN Ahmed (2025)

While single-agent localization is well-studied and often feasible with techniques like particle filters under simplified conditions, real-world deployments often demand more scalable and collaborative solutions. Multi-agent systems or robotic swarms, used in domains ranging from search-and- rescue missions to coordinated drone displays, can enhance localization through shared information. However, this inter- agent communication introduces additional complexity in terms of coordination, data fusion, and consistency.

This research investigates both individual and collaborative localization strategies for autonomous agents. We first implement a single-robot particle filter based on Peter Corke's Robotics Toolbox Corke (2017). We also extend it by adding a linear dynamics and measurement model in addition to his existing nonlinear bicycle and range-bearing sensor to act as a simpler test. We then extend this to a multi-agent system using message-passing, Gaussian belief propagation, and factor graph representations. Our approach simulates a two-layered structure wherein some robots possess highly accurate state estimates from GPS or other sources while others infer their states relative to these anchors. This work aims to evaluate and compare the performance of decentralized estimation methods under varying levels of complexity and sensor availability, ultimately contributing insights toward scalable localization

in swarm robotics.

Project Implementation and Contributions

The project was structured in three progressive levels of complexity, beginning with a foundational implementation and culminating in advanced multi-agent inference and mapping strategies.

In **Level 1: Basic Implementation**, we developed a simulation environment for a single robot and implemented a Sequential Importance Resampling (SIR) particle filter for localization. This phase required formulating the probabilistic model for the robot's motion and sensor observations, and tuning parameters for robust performance.

In **Level 2: Multi-Agent Collaboration**, the system was extended to include multiple robots, where agents with global pose estimates were initially stationary. We implemented relative measurement sharing through message passing and Gaussian belief propagation to improve localization accuracy across the swarm.

Level 3: Advanced Representations introduced greater complexity in both robot behavior and estimation techniques. Robots with global estimates were allowed to move, and those that previously relied purely on swarm navigation were provided with sparse, noisy measurements. We implemented factor graphs for joint state estimation and compared their performance to that of particle filters. To challenge our algorithms further, we introduced an unknown environment and explored landmark averaging techniques for map fusion.

While we collaborated on every aspect of the project—from research and modeling to debugging and visualization—individual strengths played a role in dividing core responsibilities. **Komal** focused on the linear dynamics and single-agent formulation, including deriving the necessary mathematical models and tuning parameters for the particle filter. **Joey** contributed heavily to the multi-agent framework, including implementing relative measurement fusion and developing the factor graph-based inference system.

Chapter 2

Background

2.1 BACKGROUND

Localization is a foundational problem in robotics that involves determining an agent's position and orientation within an environment. In the absence of external tracking systems like GPS or motion capture, autonomous robots must rely on onboard sensors and probabilistic models to estimate their states. This is inherently a problem of inference under uncertainty, where noisy and partial observations must be used to infer the most probable trajectory of the robot over time.

One of the most widely used frameworks for such inference are Bayesian networks, which model conditional dependencies using a directed graph. A popular form for robotics is the dynamic Bayesian network (DBN) which models the temporal evolution of system states and their relationships with control actions and sensor measurements. DBNs are well-suited for robotics due to their ability to represent the Markovian nature of robot motion and the conditional independence between an observation and the state it results from. The resulting framework repeats at each time step, resulting in a consistent structure that it is easier to build inference solutions for. However, exact inference in DBNs becomes computationally infeasible in high-dimensional, nonlinear, and non-Gaussian systems, prompting the need for approximate methods such as Particle Filters (PFs), especially the Sequential Importance Resampling (SIR) variant.

When multiple agents are present in an environment and able to communicate, the resulting multi-agent systems introduce new opportunities and challenges. By sharing information, robots in a swarm can collaboratively enhance their localization accuracy. However, Bayesian networks are not a suitable option for this localization problem as they only flow in one direction. It is not possible to model the loops and interconnected nature of multi-agent localization. However, by using more general graphical models like Markov Random Fields Besag (1974) or Factor Graphs Koller and Friedman (2009) with techniques such as message passing Pearl (1988) and Gaussian Belief Propagation Davison and Ortiz (2022), it is possible to do decentralized data fusion across agents. These approaches allow each agent to maintain a belief over its state that is informed by both its own measurements and those of neighboring agents, creating a robust network of interdependent estimations. Notably, Factor Graphs, popularized in tools like GTSAM Dellaert (n.d.), offer a structured way to represent the joint probability distributions over states and observations in large-scale systems.

Recent work also extends collaborative localization into the domain of Simultaneous Localization and Mapping (SLAM), where the environment itself is unknown. Methods like landmark averaging and map merging become essential for maintaining consistency and accuracy across agents navigating an unfamiliar space.

This research builds upon these foundational concepts to develop and evaluate localization strategies for both single agent systems with a SIR filter and multi-agent systems using Gau. By combining established probabilistic models with modern multi-agent inference techniques, we aim to contribute toward robust, scalable solutions for autonomous robotic navigation.

Chapter 3

Methodology

Our approach to autonomous localization is structured into three progressive levels, each building upon the last to explore increasingly complex and collaborative probabilistic reasoning frameworks. The methodology consists of implementing and evaluating single-agent localization, expanding to basic multi-agent collaboration with factor graphs and GBP, and finally fusing both approaches using and comparing the strengths and weaknesses.

Level 1: Single-Agent Localization Using Particle Filter

The first phase of our work involves localizing a single robot in a two-dimensional (2D) environment using a Particle Filter (PF) based on a Dynamic Bayesian Network (DBN). The robot navigates a known map containing static landmarks, using noisy control inputs and range-bearing measurements to localize itself. The PF approximates the posterior belief $p(x_t|z_{1:t}, u_{1:t})$ over the robot's state x_t at time t, given the history of measurements $z_{1:t}$ and control inputs $u_{1:t}$.

The particle filter operates through three main steps:

1. Sampling (Prediction Step): A set of N particles $\{x_t^{[i]}\}_{i=1}^N$ are sampled based on the robot's motion model:

$$x_t^{[i]} \sim p(x_t|x_{t-1}^{[i]}, u_t) \tag{3.1}$$

The motion model for the bicycle model is typically defined as:

$$x_t = \begin{bmatrix} x_{t-1} + v_t \Delta t \cos(\theta_{t-1}) \\ y_{t-1} + v_t \Delta t \sin(\theta_{t-1}) \\ \theta_{t-1} + \frac{v_t}{L} \tan(\delta_t) \Delta t \end{bmatrix} + \mathcal{N}(0, Q) \tag{3.2}$$

where v_t is the linear velocity, δ_t is the steering angle, θ is the vehicle's heading, L is the wheelbase of the vehicle, and Q is the process noise covariance matrix. The model assumes constant velocity and steering input during the time interval Δt, and includes motion uncertainty modeled as zero-mean Gaussian noise.

2. Weighting (Measurement Update): Each particle is assigned a weight $w_t^{[i]}$ based on the likelihood of the observed measurement z_t given the particle's state:

$$w_t^{[i]} \propto p(z_t|x_t^{[i]}) \tag{3.3}$$

For range and bearing observations to known landmarks, we use:

$$p(z_t|x_t^{[i]}) \propto \exp\left(-\frac{1}{2}(z_t - \hat{z}_t^{[i]})^\top R^{-1}(z_t - \hat{z}_t^{[i]})\right) \tag{3.4}$$

where $\hat{z}_t^{[i]}$ is the expected measurement from particle $x_t^{[i]}$, and R is the measurement noise covariance matrix.

3. Resampling: To address particle degeneracy, we perform resampling where particles are drawn with replacement according to their weights:

$$x_t^{[i]} \sim \text{Discrete}\left(\left\{x_t^{[j]}, w_t^{[j]}\right\}_{j=1}^{N}\right) \tag{3.5}$$

This ensures particles with higher likelihood dominate the next generation.

3.0.1 Multi-Agent Localization

In the second phase, the system is extended to include multiple robots. A subset of these robots are localized to a high degree of accuracy (e.g., through GPS or high quality maps and sensors) while others must localize themselves through inter-agent communication. This is accomplished by connecting the robots in a factor graph, with their positions as the variable nodes in the graph and their relative measurements represented as the factors that connect them. For ease of implementation, only the robot's position is considered for this work, with the heading estimated via the robot's motion.

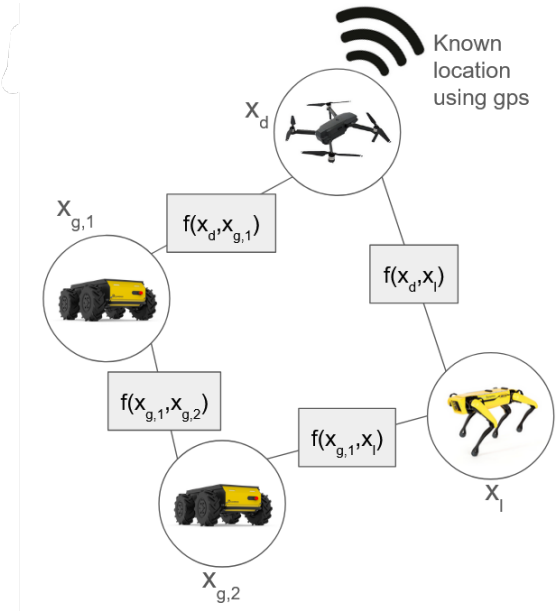

Figure 3.1: An example factor graph

Factor Graph Structure

The probability of any given set of robot positions X is modeled as the product of graph's factors, with each factor represented as an energy-based model based on its subset of X. As a result, solving for the maximum a posteri (MAP) estimate of the variables can be viewed as minimizing the energy of all the factors.

$$\tilde{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad X = \{\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_n\} \tag{3.6}$$

$$p(X) = \prod_i f_i(X_i). \tag{3.7}$$

$$f_i(X_i) \propto e^{-E_i(X_i)}, \quad E_i(X_i) \geq 0 \tag{3.8}$$

For simplicity of notation, x will represent the full position vector in all remaining equations of this section.

To compute the MAP inference of the factor graph based on the observed relative measurements, we solve iteratively using GBP. Each node in the graph is initialized based on its prior estimate of its position. For this work, even if that prior estimate is non-Gaussian, it is approximated with a Gaussian distribution to allow for easier computations.

$$p(x) \sim \mathcal{N}(\mu_x, \Sigma_x) \tag{3.9}$$

Factors are represented similarly, with the residual r of the observed measurement z and the expected measurement of the current state estimate $h(X)$ modeled as a gaussian.

$$z = h(X) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \Sigma_n) \tag{3.10}$$

$$r = z - h(X), \quad r \sim \mathcal{N}(0, \Sigma_n) \tag{3.11}$$

$$p(X) \propto e^{-E(X)}, \quad E(X) = \frac{1}{2}(h(X) - z)^\top \Sigma_n^{-1}(h(X) - z). \tag{3.12}$$

To make it easier to take products of Gaussian probabilities and factors, all Gaussians are represented in the canonical form, Eq. 3.16, rather than their moment form, Eq. 3.14. Rather than the traditional μ and Σ, the gaussian is represented by the information vector η and precision matrix Λ. This form makes it easier to take products of Gaussians, as products can be viewed as summing the information vectors and precision matrices of the Guassians. This makes it much easier to combine the Gaussian messages used in GBP. As previously stated, we can view this as minimizing the energy of this factor.

$$p(x) \propto e^{-E(x)} \tag{3.13}$$

$$E(x) = \frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu) \tag{3.14}$$

$$\eta = \Sigma^{-1}\mu, \quad \Lambda = \Sigma^{-1} \tag{3.15}$$

$$E(x) = \frac{1}{2}x^T \Lambda^{-1} x + \eta^T x \tag{3.16}$$

Message Passing and Gaussian Belief Propagation

To combine information between multiple factors and arrive at the best possible estimate, message passing is employed with GBP used to solve for the best possible estimate. GBP can be viewed as a specific version of the more general Sum Product algorithm operating under the Gaussian assumption discussed previously.

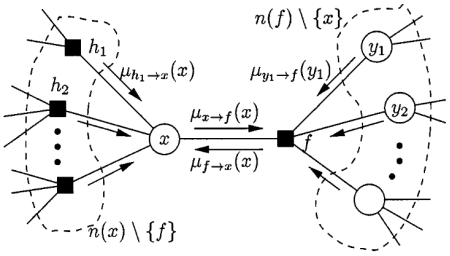

Figure 3.2: Sum product algorithm structure Davison and Ortiz (2022)

GBP works by repeatedly applying the same 3 steps in succession. This process can be done across all variables and factors simultaneously or asynchronously, with variables passing information to other ones sporadically. The "loopy" nature of this process for factor graphs with loops means that there are no guarantees on convergence. However it still generally converges, albeit not always to the correct answer Weiss and Freeman (1999).

1. Variable to Factor Messages To start this process, a variable will take the product of the incoming messages from other factors, with the exception of any messages from the variable it is sending the message to. This describes the variable's belief b_i based on the other information it has seen. This allows information to flow across the graph, with the information obtained from factors informing other factors via improved variable beliefs.

$$m_{x_i \to f_j} = \prod_{s \in \mathcal{N}(i) \setminus j} m_{f_s \to x_i} \tag{3.17}$$

Since we are working with gaussians in canonical form, this process just consists of summing the information vectors and precision matrixes of the incoming messages.

$$\eta_{x_i \to f_j} = \sum_{s \in \mathcal{N}(i) \setminus j} \eta_{f_s \to x_i} \quad \Lambda_{x_i \to f_j} = \sum_{s \in \mathcal{N}(i) \setminus j} \Lambda_{f_s \to x_i} \tag{3.18}$$

If no messages have been sent to the variable or the variable is connected to only 1 factor, it just provides its current belief whenever it sends a message. This is important at the start to kickstart the process as the prior belief is the first belief sent.

2. Factor to Variable Messages This step is the most complex, as it depends on the measurement that defines the factor, which could be a nonlinear function of the state, and the marginalization of the factor's equation. In short, the factor takes the product of all the surrounding variables, except the variable it is sending the message to, and marignalizes out to find an updated best belief for that variable.

$$m_{f_j \to x_i} = \sum_{X_j \setminus x_i} f_j(X_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{x_k \to f_j}. \tag{3.19}$$

To start, we must define the measurement equation $h(X)$ that the factor is based around. This is ideally a linear function of the factor's state, as it makes the computation of the factor' belief based on the measurement covariance Σ_n simple.

$$h(X) = JX + c \tag{3.20}$$

$$\eta = J^\top \Sigma_n^{-1}(z - c), \quad \Lambda = J^\top \Sigma_n^{-1} J \tag{3.21}$$

$$m_{f_j \to x_i} = \sum_{X_j \setminus x_i} f_j(X_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{x_k \to f_j}. \tag{3.22}$$

When the measurement is nonlinear, the current estimate of the factor's variable is used to linearize the equation. This linearization results in $c = h(X_0) - JX_0$.

$$J = \frac{\partial h}{\partial X}\Big|_{X_0} \Rightarrow h(X) = h(X_0) + J(X - X_0) \tag{3.23}$$

The factor is thus described as a Gaussian in the canonical form (as indicated by the exponent of -1).

$$\eta_f = \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix}, \quad \Lambda_f = \begin{bmatrix} \Lambda_{1,1} & \Lambda_{1,2} \\ \Lambda_{2,1} & \Lambda_{2,2} \end{bmatrix} \tag{3.24}$$

To pass a message to a variable, the factor first takes the product of the messages from all the variables it is connected to, except the variable receiving the message. Like before, this enables an improvement in that variable's belief by using the best estimate of the factor's variables to improve one variable's belief. When messages are passed in, their information vector and precision matrices are added to the portion of the factor's information vector and the diagonal of the factor's precision matrix for that variable. For this work, we only looked at factors of 2 variables, but it is possible to extend to any number.

$$\eta_f' = \begin{bmatrix} \eta_1 \\ \eta_2 + \eta_{M_2} \end{bmatrix}, \quad \Lambda_f' = \begin{bmatrix} \Lambda_{1,1} & \Lambda_{1,2} \\ \Lambda_{2,1} & \Lambda_{2,2} + \Lambda_{M_2} \end{bmatrix} \tag{3.25}$$

Once the messages have been added in, it is now time to marginalize. You can transform back into the moment form as it is easier to marginalize there, but you can also use the following equations.

$$\eta_{M_1} = \eta_1 - \Lambda_{1,2}\Lambda_{2,2}^{-1}\eta_2 \quad \text{and} \quad \Lambda_{M_1} = \Lambda_{1,1} - \Lambda_{1,2}\Lambda_{2,2}^{-1}\Lambda_{2,1} \tag{3.26}$$

3. Belief Update The final step is to update the belief of the variable receiving the message. This is essentially just the same process as passing a message to a factor, but all messages from factors are included.

$$b_i(x_i) = \prod_{s \in \mathcal{N}(i)} m_{f_s \to x_i} \tag{3.27}$$

$$\eta_{b_i} = \sum_{s \in \mathcal{N}(i)} \eta_{f_s \to x_i}, \quad \Lambda_{b_i} = \sum_{s \in \mathcal{N}(i)} \Lambda_{f_s \to x_i}. \tag{3.28}$$

By repeatedly following these steps, a factor graph will converge to a best estimate of the factor graph's variables.

Implementation Details and Tricks

To further our understanding, we implemented the factor graph architecture from scratch. Due to the complexity in representing angles' repeating nature, we only focused on position. We also constrained our factors to be between only 2 variables at a time, reflecting the one-to-one communication we envision within a moving robotic swarm. For this work, we used 2 types of measurements. First, a purely linear relative position measurement that describe the position between the 2 robots. We assumed a very accurate measurement model, with uncorrelated noise and standard deviations of 0.1 m, due to the closer operation of the robots.

$$z = \begin{bmatrix} z_x \\ z_y \end{bmatrix} = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix} \tag{3.29}$$

$$\Sigma_n = \begin{bmatrix} 0.1^2 & 0 \\ 0 & 0.1^2 \end{bmatrix} \tag{3.30}$$

And second, a nonlinear range measurement. This one requires the linearization strategies discussed above. Again, we assumed a very accurate measurement with a standard deviation of 0.1 m.

$$z = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.31}$$

$$\Sigma_n = .1^2 \tag{3.32}$$

3.0.2 Problem Studied

Simulation Environment and Parameters

The robot operates within a 10×10 meter map containing 20 fixed landmarks, used for localization and sensor updates. The robot's motion and sensing are modeled using the following general parameters:

- **V (Odometry Noise):** Models uncertainty in the robot's motion estimate, including two components: the distance traveled and the change in orientation. Computed as:
$$V[1] = \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}, \quad V[2] = \theta_{k+1} - \theta_k$$
- **W (Sensor Noise):** Represents the noise in landmark measurements, with two components: range to the landmark and the relative bearing.
- **Q (Process Noise):** Captures uncertainty in the robot's motion model across three dimensions: x, y, and θ, representing random disturbances in robot movement.
- **L (Likelihood Variance):** Defines the variance of the Gaussian used to evaluate the likelihood of particles given a sensor measurement. A higher L results in a broader distribution, increasing the diversity of particles sampled during the resampling phase.

Timing Parameters:

- **Measurement Period:** The interval (in time steps) between successive landmark measurements; configurable (e.g., every 1, 10, or ∞ time steps).
- **Time Step:** The discrete simulation time interval, set to 0.1 seconds.

These parameters collectively define the robot's motion, sensing, and belief update processes in the particle filter framework.

Implementation

In the first test, we evaluated single-agent localization under a linear, deterministic motion model with additive Gaussian noise. The robot moves in a 2D environment with 2 known landmarks to represent walls in the x and y directions. Its state evolves according to the linear motion model:

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{u}_t * \Delta t + \mathbf{w}_t \tag{3.33}$$

where $\mathbf{x}_t$ is the state vector at time t, $\mathbf{u}_t$ is the control input of a velocity in the x and y direction, $\mathbf{w}_t \sim \mathcal{N}(0, Q)$ is the process noise. The bearing is assumed to be fixed, reducing the state vector to 2 dimensions of actual unknown parameters. The observation model is also linear:

$$\mathbf{z}_t = \mathbf{H}\mathbf{x}_t + \mathbf{v}_t, \quad \mathbf{v}_t \sim \mathcal{N}(0, R) \tag{3.34}$$

Here, H alternates between returning the distance to the edge of the map in the positive x and positive y direction, representing a rangefinder that alternates between measuring a wall in the x and y direction.

For the second set of tests, the robot was subjected to a nonlinear bicycle kinematic model, which better reflects real-world robotic movement:

$$x_{t+1} = x_t + v_t \cos(\theta_t)\Delta t + w_x \tag{3.35}$$
$$y_{t+1} = y_t + v_t \sin(\theta_t)\Delta t + w_y \tag{3.36}$$
$$\theta_{t+1} = \theta_t + \omega_t \Delta t + w_\theta \tag{3.37}$$

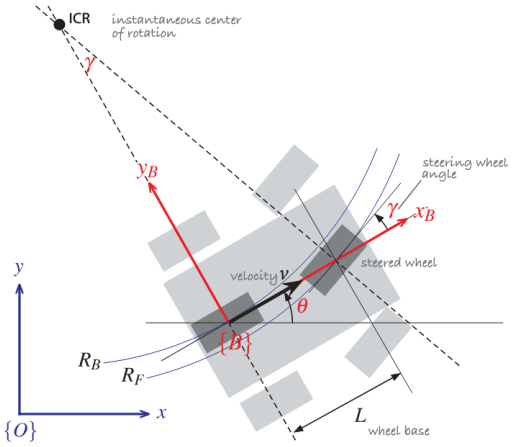

Figure 3.3: The bicycle model Corke (2017)

The observation model is also nonlinear, based on Euclidean distances and angles to known landmarks within the environment:

$$r_i = \sqrt{(x - x_i)^2 + (y - y_i)^2} + v_r \tag{3.38}$$
$$\phi_i = \arctan 2(y_i - y, x_i - x) - \theta + v_\phi \tag{3.39}$$

To implement and simulate the mobile robot localization problem, we utilized the Peter Corke Robotics Toolbox for MATLAB, a powerful and widely adopted library designed for robotics research and education. Our implementation follows the Monte Carlo Localization (MCL) framework using a particle filter approach. The toolbox offers an intuitive abstraction for robotic vehicles, sensors, and probabilistic estimation techniques. The process began with the creation of a vehicle model using the LinearDynamics class, which allows for flexible modeling of a car-like robot. A covariance matrix was defined to simulate realistic odometric noise. The robot was then programmed to follow a random waypoint path using the RandomPath driver within a bounded region.

A sensor model was built using RangeBearingSensor, which emulates a range-bearing sensor mounted on the robot, observing fixed landmarks on the map. A 2D square map with 20 randomly positioned landmarks was created using the XYMap class. The sensor's measurement noise was modeled using a separate covariance matrix to reflect range and bearing inaccuracies.

Next, the particle filter itself was implemented via a modification of the existing ParticleFilter class. The existing class's functionality was preserved, but the inner logic was rewritten to ensure understanding. We initialized the filter with a specified number of particles (1000 in this case), defining both the likelihood function (based on a 2D Gaussian error model) and the particle motion noise. The filter was executed over 100 time steps, during which the robot's actual pose, sensor observations, and odometry were used to iteratively update and converge the particle cloud. Visualization tools provided by the library allowed us to analyze particle convergence, pose estimation accuracy, and standard deviations over time. This real-time visualization demonstrated the efficiency and stability of the filter under noisy conditions. Once initialized, the robot's estimated path closely followed the true path, with discrepancies only evident in the early stages before convergence. The spread and mean of the particles over time revealed how quickly the localization system honed in on the true pose of the robot.

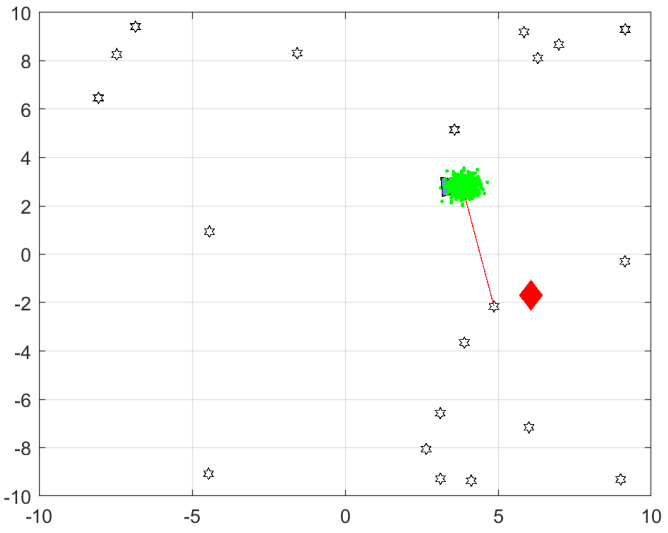

Figure 3.4: The robot localizing itself with the particle filter Corke (2017)

Chapter 4

Test Case

To evaluate the robustness and accuracy of our proposed localization strategies, we implemented a series of test cases in both linear and nonlinear dynamic environments. These scenarios examine how well the particle filter and GBP-based localization techniques perform under varying levels of motion complexity, noise, and agent coordination. Performance was evaluated on how well the resulting estimate matched the true robot position and orientation and if any error stayed within 2σ bounds. All solutions were tested over a 100 time step trajectory with a 0.1 s timestep.

The parameters of both cases were based around working in human environments, with the 20 m by 20 m square being a very generic approximation for experimentation. The noise parameters were based around that scale for odometry (m^2 and degrees) and sensor noise (m^2 and degrees).

4.0.1 Single-Agent Localization Tests

2 robots were tested, with each having different parameters. Roomba was meant to be the more realistic case, with good but not spectacular sensors. Meanwhile Superbot serves as an ideal case, with high precision measurements. Only Roomba was tested in the single agent case, and will be discussed more later. Superbot was just verified to provide a highly accurate,

Robot	**Odometry Noise (V)**	**Sensor Noise (W)**
Roomba	$\text{diag}([0.1^2,\ 1^2])$	$\text{diag}([0.1^2,\ 1^2])$
Superbot	$\text{diag}([0.025^2,\ .25^2])$	$\text{diag}([0.025^2,\ .25^2])$

Table 4.1: True odometry and sensor noise parameters (standard deviations squared).

4.0.2 Multi-Agent Localization

To further understand the limitations of localization under constrained conditions, we conducted experiments removing Roomba's ability to measure external landmarks—forcing it to rely solely on odometry. This modification simulates real-world scenarios such as sensor failure, degraded environments, or stealth operations where external measurements are unavailable. As expected, the localization performance deteriorated rapidly: the particle filter, deprived of corrective measurements, accumulated error from odometry drift, resulting in significant pose uncertainty. The particles spread widely, and the robot's estimated trajectory diverged from its true path within a short time horizon.

In the multi-agent context, we evaluated scenarios where agents could share information sporadically. Although this setup holds promise—especially when one agent (e.g., Superbot) has access to reliable measurements—it requires careful design of communication protocols and estimation fusion. This work can be worth it however, as it promises the ability to use centralized bespoke robots paired with cheaper, less powerful robots to accomplish more complex tasks. For this work, we set the communication rate to occur every 5 timesteps, which equates to 2 Hz. This feels accurate as communication may not be possible as quickly as measurements due to the complexities associated with maintaining connection and ensuring consistent communication.

Chapter 5

Results and Discussion

Due to technical issues and time constraints, only the nonlinear model was completed and only linear relative measurements were used.

Single Agent Performance

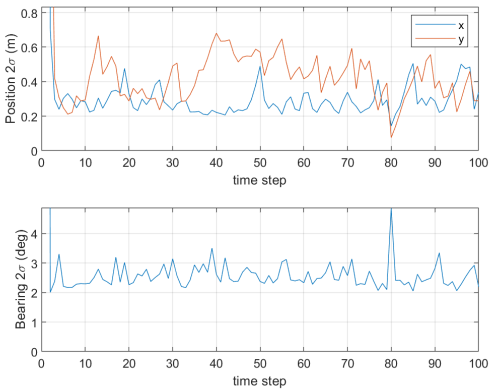

Figure 5.1: Roomba localization uncertainty

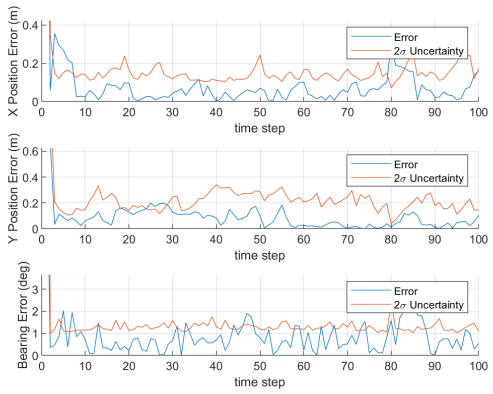

Figure 5.2: Roomba localization error

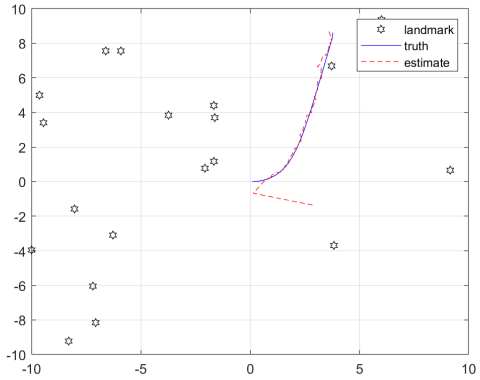

Figure 5.3: Roomba localization mapped

The settings below were used to generate these results. These were also used in the multi-agent tests.

Robot	Process Noise (Q)	Likelihood (L)
Roomba	$\text{diag}([0.25^2,\ 0.25^2,\ (\frac{10\pi}{180})^2])$	$\text{diag}([0.5^2,\ (\frac{10\pi}{180})^2])$
Superbot	$\text{diag}([0.25^2,\ 0.25^2,\ (\frac{1\pi}{180})^2])$	$\text{diag}([0.25^2,\ (\frac{1\pi}{180})^2])$

Table 5.1: Filter process and likelihood noise parameters

The single-agent particle filter posed several challenges due to the inherent nonlinearity of the system. Unlike linear systems, one cannot simply input known values and expect accurate tracking; the filter must begin with a wide distribution and be gradually tuned. Excessive confidence in the initial estimate often led to inaccurate tracking, with errors exceeding expected bounds.

A key difficulty lay in estimating the bearing θ, which was notably the most nonlinear component of the observation model. This parameter required the most tuning effort. When appropriately configured, the system achieved a typical bearing error of approximately 1–2°. However, if the tuning was too aggressive, the system tended to become overconfident and underestimate its own uncertainty, resulting in the error exceeding the 2σ error bounds.

To address this, both the process noise Q and likelihood variance L had to be increased by nearly an order of magnitude compared to their true values. This helped stabilize the filter and prevent overfitting to noisy measurements.

The measurement rate was also found to have a significant effect on performance. Frequent measurements could actually degrade performance by reinforcing overconfidence, especially in bearing estimates that naturally vary. Intentionally reducing the measurement rate introduced intermittent gaps in observations, which encouraged the system to adopt a more cautious stance and improved robustness.

Rotational motion, in particular, introduced substantial uncertainty in bearing. This was expected, as turning is the most nonlinear maneuver the robot can execute and thus presents the greatest challenge for accurate estimation.

5.0.1 Multi-Agent Results

No Measurements

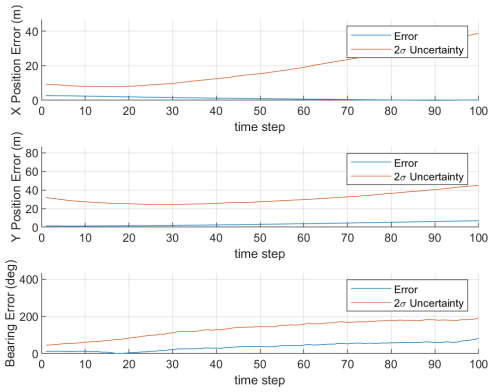

Figure 5.4: Roomba localization error without measurements

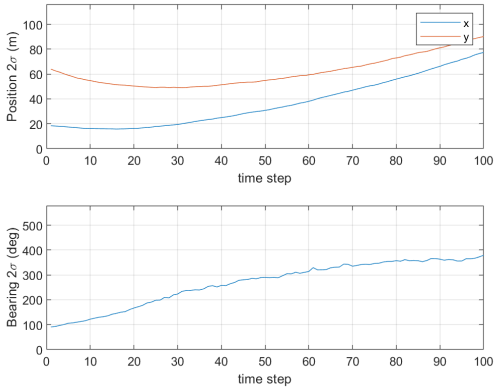

Figure 5.5: Roomba localization uncertainty without measurements

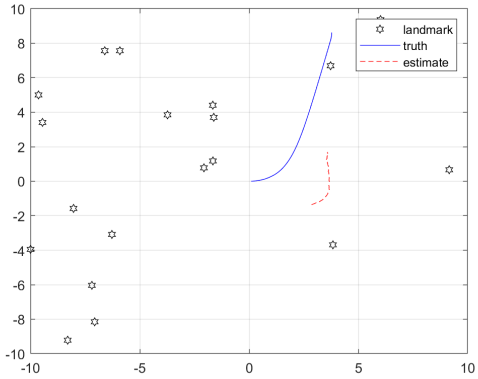

Figure 5.6: Roomba localization mapped without measurements

Without any measurements, the Roomba robot floundered, which is unsurprising. Odometry did not prove to be an adequate solution to the complex problem of robot localization as the robot has nothing to ground itself with. Surprisingly, it was able to somewhat track the true trajectory, albeit from a poor starting point. The orinetation error proved to be a problem however, as the uncertainty grew beyond possible bounds. The unique nature of measuring orientation makes it hard to measure, so it's not suprrising that this was the case.

Multi-Agent Localization

When the robots coordinated using GBP, the following was observed.

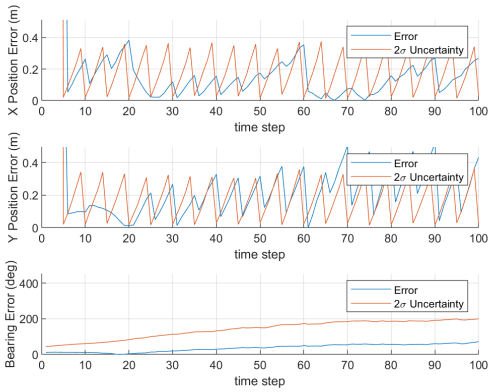

Figure 5.7: Roomba localization error using GBP

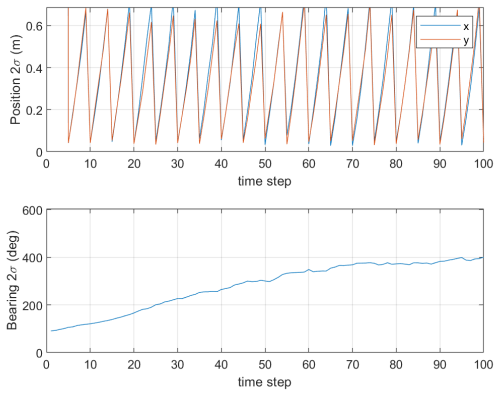

Figure 5.8: Roomba localization uncertainty using GBP

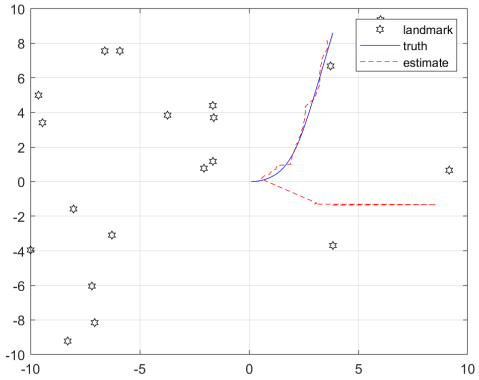

Figure 5.9: Roomba localization mapped using GBP

With GBP, the Roomba did much better. Superbot's relative measurements provided a reliable source of information that helped group its dead reckoning. However, the slower rate meant that it had time for its uncertainty to grow. This resulted in a cyclical pattern that limited its ability to truly localize itself accurately. The error did exceed the 2σ bounds, indicating that this was unsuccessful at maintaing a truly accurate localization.

Additionally, since only position information was provided, bearing still proved to be nearly unobservable. Exploring measurements that can provide at least some observability of the bearing would be vital for future usage. This approach shows promise, but requires more work to be truly reliable.

5.1 Future Work

There were multiple portions of this project that were never quite reached due to technical issues and time constraints. The most basic focus would likely be on finishing the linear model to provide a simpler system to work with, especially for the factor graph to localize. Additionally, it would be useful to explore linearization within the factor graph through nonlinear measurements.

For the more ambitious Level 3 goals, it would likely require integrating additional algorithms to estimate the map. Factor graphs can do this however, so it would be interesting to add the landmarks as another variable in the factor graph. This would likely necessitate moving from tracking the state via odometry to connecting everything with a factor graph, maintaining a history of the robot's state. This is how real robot localization is worked with. This could be done with GTSAM, but a connection between MATLAB and C++ would be required to do so.

Chapter 6

Conclusions

Our experiments demonstrate that a single-agent system can successfully localize in a nonlinear model using the particle filter framework with appropriate parameter tuning and careful initialization. However, when extended to a multi-agent setup, the system fails to converge reliably due to compounded uncertainties and slow measurement updates. Future work will focus on improving inter-agent communication, adaptive resampling strategies, and integrating graph-based SLAM frameworks like GTSAM to enable robust collaborative localization in nonlinear environments. This problem is a fascinating application of graphical models and holds much promise to other researchers. But they would likely be better-suited using existing software for this after doing a basic example using a script, as writing a full factor graph solver is challening and time-intensive.

References

Ahmed, N. (2025), 'More on forward-backward algorithm and dbns: Hmm numerical considerations and applications; linear-gaussian generative state space models'.

Besag (1974), 'Spatial interaction and the statistical analysis of lattice systems'.
URL: *https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1974.tb00999.x*

Corke, P. (2017), *Mobile Robot Vehicles*, Springer International Publishing, Cham, p. 99–124.
URL: *https://doi.org/10.1007/978-3-319-54413-7*$_4$

Dagum, P., Galper, A. and Horvitz, E. (1992), *Dynamic Network Models for Forecasting*, Morgan Kaufmann, p. 41–48.
URL: *https://www.sciencedirect.com/science/article/pii/B9781483282879500104*

Davison, A. J. and Ortiz, J. (2022), 'Futuremapping 2: Gaussian belief propagation for spatial ai', (arXiv:1910.14139). arXiv:1910.14139 [cs].
URL: *http://arxiv.org/abs/1910.14139*

Dellaert, F. (n.d.), 'Factor graphs and gtsam: A hands-on introduction'.

Koller, D. and Friedman, N. (2009), *Probabilistic Graphical Models: Principles and Techniques*, MIT Press. Google-Books-ID: 7dzpHCHzNQ4C.

Pearl, J. (1988), *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Elsevier. Google-Books-ID: mn2jBQAAQBAJ.

Weiss, Y. and Freeman, W. (1999), Correctness of belief propagation in gaussian graphical models of arbitrary topology, *in* 'Advances in Neural Information Processing Systems', Vol. 12, MIT Press.
URL: *https://proceedings.neurips.cc/paper/1999/hash/10c272d06794d3e5785d5e7c5356e9ff-Abstract.html*

```
%ParticleFilter Particle filter class
%
% Monte-carlo based localisation for estimating vehicle pose based on
% odometry and observations of known landmarks.
%
% Methods::
% run         run the particle filter
% plot_xy     display estimated vehicle path
% plot_pdf    display particle distribution
%
% Properties::
%  robot        reference to the robot object
%  sensor       reference to the sensor object
%  history      vector of structs that hold the detailed information from
%               each time step
%  nparticles   number of particles used
%  x            particle states; nparticles x 3
%  weight       particle weights; nparticles x 1
%  x_est        mean of the particle population
%  cov          covariance matrix of the particles
%  Q            covariance of noise added to state at each step
%  L            covariance of likelihood model
%  w0           offset in likelihood model
%  dim          maximum xy dimension
%
% Example::
%
% Create a landmark map
%    map = PointMap(20);
% and a vehicle with odometry covariance and a driver
%    W = diag([0.1, 1*pi/180].^2);
%    veh = Vehicle(W);
%    veh.add_driver( RandomPath(10) );
% and create a range bearing sensor
%    R = diag([0.005, 0.5*pi/180].^2);
%    sensor = RangeBearingSensor(veh, map, R);
%
% For the particle filter we need to define two covariance matrices.  The
% first is is the covariance of the random noise added to the particle
% states at each iteration to represent uncertainty in configuration.
%    Q = diag([0.1, 0.1, 1*pi/180]).^2;
% and the covariance of the likelihood function applied to innovation
%    L = diag([0.1 0.1]);
% Now construct the particle filter
%    pf = ParticleFilter(veh, sensor, Q, L, 1000);
% which is configured with 1000 particles.  The particles are initially
% uniformly distributed over the 3-dimensional configuration space.
%
% We run the simulation for 1000 time steps
%    pf.run(1000);
```

```
% then plot the map and the true vehicle path
%    map.plot();
%    veh.plot_xy('b');
% and overlay the mean of the particle cloud
%    pf.plot_xy('r');
% We can plot the standard deviation against time
%    plot(pf.std(1:100,:))
% The particles are a sampled approximation to the PDF and we can display
% this as
%    pf.plot_pdf()
%
% Acknowledgement::
%
% Based on code by Paul Newman, Oxford University,
% http://www.robots.ox.ac.uk/~pnewman
%
% Reference::
%
%   Robotics, Vision & Control,
%   Peter Corke,
%   Springer 2011
%
% See also Vehicle, RandomPath, RangeBearingSensor, PointMap, EKF.
%
% Modified by Joey Caley for ASEN 5519 Autonomous Bayesian Reasoning



% Copyright (C) 1993-2017, by Peter I. Corke
%
% This file is part of The Robotics Toolbox for MATLAB (RTB).
%
% RTB is free software: you can redistribute it and/or modify
% it under the terms of the GNU Lesser General Public License as published by
% the Free Software Foundation, either version 3 of the License, or
% (at your option) any later version.
%
% RTB is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU Lesser General Public License for more details.
%
% You should have received a copy of the GNU Leser General Public License
% along with RTB.  If not, see <http://www.gnu.org/licenses/>.
%
% http://www.petercorke.com


%note this is not coded efficiently but rather to make the ideas clear
%all loops should be vectorized but that gets a little matlab-speak intensive
```

```
%and may obliterate the elegance of a particle filter....

classdef ParticleFilter_Proj < handle
%TODO
% x_est should be a weighted mean
% std should be a weighted std (x-mean)' W (x-mean)  ???
    properties
        robot
        sensor
        nparticles
        msmt_period % how many steps between measurements
        x           % particle states; nparticles x 3
        weight      % particle weights; nparticles x 1
        x_est        % mean of the particle population
        cov         % covariance matrix of the particle population
        Q           % covariance of noise added to state at each step
        L           % covariance of likelihood model
        history
        keephistory
        dim         % maximum xy dimension

        h           % graphics handle for particles
        randstream
        seed0
        w0
        x0          % initial particle distribution
        anim
    end % properties

    methods
        function pf = ParticleFilter_Proj(robot, sensor, Q, L, np, msmt_period, ↙
varargin)
            %ParticleFilter.ParticleFilter Particle filter constructor
            %
            % PF = ParticleFilter(VEHICLE, SENSOR, Q, L, NP, OPTIONS) is a particle
            % filter that estimates the state of the VEHICLE with a landmark sensor
            % SENSOR.  Q is the covariance of the noise added to the particles
            % at each step (diffusion), L is the covariance used in the
            % sensor likelihood model, and NP is the number of particles.
            %
            % Options::
            % 'verbose'     Be verbose.
            % 'private'     Use private random number stream.
            % 'reset'       Reset random number stream.
            % 'seed',S      Set the initial state of the random number stream.  S must
            %               be a proper random number generator state such as saved in
            %               the seed0 property of an earlier run.
            % 'nohistory'   Don't save history.
            % 'x0'          Initial particle states (Nx3)
            %
```

```
% Notes::
% - ParticleFilter subclasses Handle, so it is a reference object.
% - If initial particle states not given they are set to a uniform
%   distribution over the map, essentially the kidnapped robot problem
%   which is quite unrealistic.
% - Initial particle weights are always set to unity.
% - The 'private' option creates a private random number stream for the
%   methods rand, randn and randi.  If not given the global stream is used.
%
%
% See also Vehicle, Sensor, RangeBearingSensor, PointMap.

if ~exist("msmt_period","var")
    msmt_period = 1;
end

pf.robot = robot;
pf.sensor = sensor;
pf.Q = Q;
pf.L = L;
pf.nparticles = np;
pf.msmt_period = msmt_period;

pf.dim = sensor.map.dim;
pf.history = [];
pf.x = [];
pf.weight = [];
pf.w0 = 0.05;
pf.x0 = [];

opt.private = false;
opt.reset = false;
opt.seed = [];
opt.history = true;
opt.x0 = [];



opt = tb_optparse(opt, varargin);

pf.keephistory = opt.history;
% create a private random number stream if required
if opt.private
    pf.randstream = RandStream.create( 'mt19937ar');
else
    pf.randstream = RandStream.getGlobalStream();
end

% reset the random number stream if required
if opt.reset
```

```
            pf.randstream.reset();
        end

        % return the random number stream to known state if required
        if ~isempty(opt.seed)
            set(pf.randstream.set(opt.seed));
        end

        % save the current state in case it later turns out to give interesting ↙
results
        pf.seed0 = pf.randstream.State;

        if opt.x0
            pf.x0 = opt.x0;
        end


    end


    function init(pf)
        %ParticleFilter.init Initialize the particle filter
        %
        % PF.init() initializes the particle distribution and clears the
        % history.
        %
        % Notes::
        % - If initial particle states were given to the constructor the states are
        %   set to this value, else a random distribution over the map is used.
        % - Invoked by the run() method.
        pf.robot.init();
        pf.history = [];

        % if particles have been initialized
        if pf.x0
            pf.x = pf.x0;    % assign initial particle states
        % TODO: add options for initialization gaussian uncertainty
        else % otherwise, use even distribution across input space
            pf.x = (2*rand(pf.nparticles,3) - 1) .* [pf.dim, pf.dim, pi];
        end
        pf.weight = ones(pf.nparticles, 1);

        pf.x_est = [];
        pf.cov = [];
    end

    function run(pf, niter, initFlag, endFlag, plotFlag, varargin)
        %ParticleFilter.run Run the particle filter
        %
        % PF.run(N, OPTIONS) runs the filter for N time steps.
```

```
        %
        % Options::
        % 'noplot'    Do not show animation.
        % 'movie',M   Create an animation movie file M
        %
        % Notes::
        % - All previously estimated states and estimation history is
        %   cleared.

        if ~exist("plotFlag","var")
            opt.plot = true;
        else
            opt.plot = plotFlag;
        end
        opt.movie = [];
        opt = tb_optparse(opt, varargin);

        anim = Animate(opt.movie);

        if initFlag
            pf.init();
            if opt.plot
                pf.sensor.map.plot();

                a = axis;
                a(5:6) = [-pi pi];
                axis(a)
                zlabel('heading (rad)');

                % display the initial particles
                pf.h = plot3(pf.x(:,1), pf.x(:,2), pf.x(:,3),  'g.');

                set(pf.h, 'Tag', 'particles');

                pf.robot.plot();
            end
        end

        % iterate along the chain
        for i = 1:niter
            pf.step(opt)
            anim.add();
        end

        if endFlag
            anim.close();
        end
    end

    function step(pf, opt)
```

```
            % take a step
            odom_readings = pf.robot.step();

            % move particles forward based on odometry
            pf.propagate(odom_readings)

            % move
            num_states = size(pf.x_est,1);
            if mod(num_states,pf.msmt_period) == 0
                % take a measurement
                [z, jf] = pf.sensor.reading();

                % if something was observed, evaluate particles and resample
                if ~isnan(jf)
                    pf.evaluate(z,jf)
                    pf.resample()
                end % otherwise, continue propagation without resampling
            end

            % update estimate and corresponding statistics
            x_est_curr = mean(pf.x);
            pf.x_est = [pf.x_est; x_est_curr];

            cov_mat = cov(pf.x(:,1:2));
            var_ang = sum(angdiff(x_est_curr(3)*ones(pf.nparticles,1), pf.x(:,3)).^2) ↙
/pf.nparticles;
            cov_mat = blkdiag(cov_mat, var_ang);
            pf.cov = cat(3, pf.cov, cov_mat);

            if opt.plot
                % display the updated particles
                set(pf.h, 'Xdata', pf.x(:,1), 'Ydata', pf.x(:,2), 'Zdata', pf.x(:,3));

                % display the robot
                pf.robot.plot();
                drawnow
            end

            if ~isempty(pf.anim)
                pf.anim.add();
            end

            if pf.keephistory
                hist = [];
                hist.x_est = pf.x;
                hist.w = pf.weight;
                pf.history = [pf.history hist];
            end
        end
```

```
        function plot_pdf(pf)
            %ParticleFilter.plot_pdf Plot particles as a PDF
            %
            % PF.plot_pdf() plots a sparse PDF as a series of vertical line
            % segments of height equal to particle weight.
            clf
            hold on
            for p = 1:pf.nparticles
                x = pf.x(p,:);
                plot3([x(1) x(1)], [x(2) x(2)], [0 pf.weight(p)],  'r');
                plot3([x(1) x(1)], [x(2) x(2)], [0 pf.weight(p)],  'k.', 'MarkerSize',↙
12);

            end
            grid on
            xyzlabel
            zlabel('particle weight')
            view(30,60);
            rotate3d on
        end

        function plot_xy(pf, varargin)
            %ParticleFilter.plot_xy Plot vehicle position
            %
            % PF.plot_xy() plots the estimated vehicle path in the xy-plane.
            %
            % PF.plot_xy(LS) as above but the optional line style arguments
            % LS are passed to plot.
            plot(pf.x_est(:,1), pf.x_est(:,2), varargin{:});
        end

        function display(pf)
            %ParticleFilter.display Display status of particle filter object
            %
            % PF.display() displays the state of the ParticleFilter object in
            % human-readable form.
            %
            % Notes::
            % - This method is invoked implicitly at the command line when the result
            %   of an expression is a ParticleFilter object and the command has no ↙
trailing
            %   semicolon.
            %
            % See also ParticleFilter.char.

            loose = strcmp( get(0, 'FormatSpacing'), 'loose');
            if loose
                disp(' ');
            end
```

```
            disp([inputname(1), ' = '])
            disp( char(pf) );
        end % display()

        function s = char(pf)
            %ParticleFilter.char Convert to string
            %
            % PF.char() is a string representing the state of the ParticleFilter
            % object in human-readable form.
            %
            % See also ParticleFilter.display.
            s = sprintf('ParticleFilter object: %d particles', pf.nparticles);
            if ~isempty(pf.robot)
                s = char(s, char(pf.robot) );
            end
            if ~isempty(pf.sensor)
                s = char(s, char(pf.sensor));
            end
            s = char(s, ['Q:  ' mat2str(pf.Q, 3)] );
            s = char(s, ['L:  ' mat2str(pf.L, 3)] );
            s = char(s, sprintf('w0: %g', pf.w0) );
        end


    end % methods

    methods(Access=protected)

        % propagate particles forward based on motion from odometry
        % readings
        function propagate(pf, odo)
            % sample noise
            noise_gaussian = pf.randstream.randn(pf.nparticles,3);
            Q_lower = chol(pf.Q, 'lower'); % TODO check with KF notes

            % propagate with process noise, wrap angles to +/- pi
            pf.x = pf.robot.f(pf.x, odo) + noise_gaussian*Q_lower;
            pf.x(:,3) = wrapToPi(pf.x(:,3));
        end

        % evaluate particle weights based upon the measurement received
        function evaluate(pf, z, jf)
            % Evaluate the probability of getting the observed
            % measurement for each particle's state
            z_pred = pf.sensor.h(pf.x, jf);

            del_z(:,1) = z_pred(:,1) - z(1);
            del_z(:,2) = angdiff(z_pred(:,2), z(2)*ones(pf.nparticles,1));

            probability = mvnpdf(del_z,[0 0],pf.L);
```

```
            % reset weights based on their probabilities
            pf.weight = probability / sum(probability);
        end

        % resample particles using updated weights
        function resample(pf)
            pf.x = datasample(pf.x,pf.nparticles, 'Weights',pf.weight);
        end

        function r = rand(pf, varargin)
            r = pf.randstream.rand(varargin{:});
        end

        function r = randn(pf, varargin)
            r = pf.randstream.randn(varargin{:});
        end
    end % private methods
end
```

```
classdef Factor
    %FACTOR Generic factor of a factor graph
    %   NOTE this is only set up to be a factor of 2 variables. With the
    %   variables being ordered in [x_a, x_b] with i_a < i_b

    properties
        eta % information vector
        lambda % precision matrix
        msmt % true measurement that we compare against
    end

    methods
        function obj = Factor(x_a, x_b, msmt_uncertainty)
            %FACTOR Initializes factor
            %   x_a: true value for variable a
            %   x_b: true value for variable b
            %   msmt_uncertainty: measurement noise covariance
            %   variables: indexes of the variables x_a and x_b

            d = obj.measurement(x_a, x_b, msmt_uncertainty);
            [J, c] = obj.msmtLinearization(x_a, x_b);

            % set based on dimensions
            obj.eta = J'*inv(msmt_uncertainty)*(d - c);  % TODO update with initialization equation
            obj.lambda = J'*inv(msmt_uncertainty)*J;  % TODO update with initialization equation
            obj.msmt = d;
        end

        function msgs_factor2var = msg2var(obj, i_recepients, msgs_var2factor, msgs_factor2var)
            %METHOD1 TODO

            i_recepients = sort(i_recepients);

            % initialize message values
            eta_factor = obj.eta;
            lambda_factor = obj.lambda;

            % find what messages have been sent to this factor
            i_msgs = find(~cellfun(@isempty,msgs_var2factor));

            if isempty(i_msgs)
                error("NO MESSAGES PASSED IN")
            end

            % Update message values
            if length(i_recepients) > 1  % all being update
                eta_update = [msgs_var2factor{i_msgs(1)}.eta;  ...
```

```
            msgs_var2factor{i_msgs(2)}.eta];
        lambda_update = blkdiag(msgs_var2factor{i_msgs(1)}.lambda, ...
            msgs_var2factor{i_msgs(2)}.lambda);

    elseif i_msgs(1) == i_recepients % just the first variable being updated
        eta_update = [zeros(2,1); msgs_var2factor{i_msgs(2)}.eta];
        lambda_update = blkdiag(zeros(2), msgs_var2factor{i_msgs(2)}.lambda);

    else % otherwise assume the second variable is being updated
        eta_update = [msgs_var2factor{i_msgs(1)}.eta; zeros(2,1)];
        lambda_update = blkdiag(msgs_var2factor{i_msgs(1)}.lambda, ...
            zeros(2));

    end

    eta_factor = eta_factor + eta_update;
    lambda_factor = lambda_factor + lambda_update;

    % decompose for ease of viewing
    eta_a = eta_factor(1:2);
    eta_b = eta_factor(3:4);

    lambda_aa = lambda_factor(1:2,1:2);
    lambda_ab = lambda_factor(1:2,3:4);
    lambda_ba = lambda_factor(3:4,1:2);
    lambda_bb = lambda_factor(3:4,3:4);

    % compute message to the recepient
    for i_recepient = i_recepients
        i_order = find(i_msgs == i_recepient);

        if i_order == 1
            eta_msg = eta_a - lambda_ab* ...
                inv(lambda_bb)*eta_b;
            lambda_msg = lambda_aa - lambda_ab* ...
                inv(lambda_bb)*lambda_ba;
        else
            eta_msg = eta_b - lambda_ba* ...
                inv(lambda_aa)*eta_a;
            lambda_msg = lambda_bb - lambda_ba* ...
                inv(lambda_aa)*lambda_ab;
        end

        % eta & lambda updates
        msgs_factor2var{i_recepient} = GBP.Message(eta_msg, lambda_msg);
    end
end


function msmt = measurement(obj, x_a, x_b, msmt_uncertainty)
```

```
            % MEASUREMENT Factor measurement function
            %    Noisy measurement of current state for now

            n = length(x_a) + length(x_b);

            msmt = [x_a;x_b] + mvnrnd(zeros(n,1),msmt_uncertainty)';
        end

        function [J, c] = msmtLinearization(obj, x_a, x_b)
            % MSMTLINEARIZATION Linearizes the msmt function
            %    Linearizes the msmt function about the current estimate
            %    for the factor variables x_a and x_b

            n = length(x_a) + length(x_b);

            J = eye(n);
            c = zeros(n,1);
        end

    end
end
```

```
classdef Graph
    %GRAPH Factor Graph
    %   A bipartite graph used for computing a maximum a priori estimate of
    %   a set of variables using the relationships between described by
    %   factors.
    %
    %   Currently only built to handle factor graphs with factors only
    %   connecting 2 variables.
    %
    %   NOTE: connections, var2factorMsgs, and factor2varMsgs all are the
    %   same size, with dimensions set as [# variables (v) x # factors (f)]

    properties
        variables % variables of graph, vx1 vector
        factors % factors in graph, fx1 vector
        connections % the variables each factor is connected to, fx1 cell array of ↙
vectors of connections
        msgs_var2factor % most recent messages from a variable to a factor, vxf cell ↙
array of message class
        msgs_factor2var % most recent messages from a factor to a variable, vxf cell ↙
array of message class

        % NOTE: connections, var2factorMsgs, and factor2varMsgs all are the
        % same size, with dimensions set as [# variables x # factors]
    end

    methods
        function obj = Graph()
            %GRAPH Construct an instance of this class
            obj.variables = [];
            obj.factors = [];
            obj.connections = [];
            obj.msgs_var2factor = {};
            obj.msgs_factor2var = {};
        end

        function obj = addVars(obj,variables)
            %addVars Adds variables to factor graph
            %   Should only be done BEFORE passing messages
            obj.variables = [obj.variables, variables];

            % add a row to the connections and msg matrixes
            num_newVars = length(variables);
            num_factors = length(obj.factors);

            obj.connections = [obj.connections; false(num_newVars,num_factors)];
            obj.msgs_factor2var = [obj.msgs_factor2var; cell(num_newVars,num_factors)];
            obj.msgs_var2factor = [obj.msgs_var2factor; cell(num_newVars,num_factors)];
        end
```

```
function obj = addFactors(obj, factors, factor_connection_idxs)
    %addFactors Adds factors to the factor graph
    %   Should only be done BEFORE passing messages
    %
    %   factors: nx1 vector of factor objects
    %   factor_conenction_idxs: nx1 cell array of 2x1 vectors
    %      listing the indexes of the variables that factor
    %      connects to.

    if ~iscell(factor_connection_idxs)
        factor_connection_idxs = {factor_connection_idxs};
    end

    if length(factors) ~= length(factor_connection_idxs)
        error("Must add factors with connections")
    end

    obj.factors = [obj.factors, factors];

    % add a column to the connections and msg matrixes
    num_vars = length(obj.variables);
    num_newFactors = length(factors);

    obj.connections = [obj.connections, false(num_vars,num_newFactors)];
    obj.msgs_factor2var = [obj.msgs_factor2var, cell(num_vars,num_newFactors)];
    obj.msgs_var2factor = [obj.msgs_var2factor, cell(num_vars,num_newFactors)];

    % update the connections
    for i = 1:num_newFactors
        factor_i_connection_idxs = factor_connection_idxs{i};
        obj.connections(factor_i_connection_idxs, i) = true;
    end
end

function obj = passMessage(obj, i_var1, i_var2, uniFlag)
    %passMessage  Message pass between variables
    %   Passes a message between variables 1 and 2 using their
    %   shared factor. Can be either bidirecitonal or
    %   unidirectional, as set by the uniFlag. If unidirectional,
    %   assume message is passed from variable 1 to variable 2.

    if ~exist("uniFlag","var")
        uniFlag = false;
    end

    i_vars = [i_var1, i_var2];

    % search to find matching factor
    j_factor = find(obj.connections(i_var1,:) & obj.connections(i_var2,:));
```

```
            % variable to factor
            obj.msgs_var2factor{i_var1,j_factor} = obj.variables(i_var1).msg2factor( ...
                obj.msgs_factor2var(i_var1,:), j_factor);

            if ~uniFlag
                obj.msgs_var2factor{i_var2,j_factor} = obj.variables(i_var2).msg2factor ↙
(...
                    obj.msgs_factor2var(i_var2,:), j_factor);
            end

            % factor to variable
            factor_j_msgsFromvar = obj.msgs_var2factor(:,j_factor);
            factor_j_msgs2var = obj.msgs_factor2var(:,j_factor);

            if ~uniFlag
                i_recepients = i_vars;
            else
                i_recepients = i_var2;
            end

            obj.msgs_factor2var(:,j_factor) = obj.factors(j_factor).msg2var( ...
                i_recepients, factor_j_msgsFromvar, factor_j_msgs2var);

            % belief update
            for i_recepient = i_recepients
               obj.variables(i_recepient) =  obj.variables(i_recepient).beliefUpdate ↙
(obj.msgs_factor2var(i_recepient,:));
            end
        end


    end
end
```

```
classdef Message
    %MESSAGE

    properties
        eta % Information vector
        lambda % Precision matrix
    end

    methods
        function msg = Message(eta,lambda)
            %MESSAGE Message betweeen factors and graphs
            msg.eta = eta;
            msg.lambda = lambda;
        end
    end
end
```

```
classdef RelPosFactor < GBP.Factor
    %FACTOR Relative position measurement factor
    %   NOTE this is only set up to be a factor of 2 variables. With the
    %   variables being ordered in [x_a, x_b] with i_a < i_b

    methods

        function msmt = measurement(obj, x_a, x_b, msmt_uncertainty)
            % MEASUREMENT Relative position measurement of the 2 robots
            %    Noisy relative position measurement

            msmt = x_b(1:2) - x_a(1:2) + mvnrnd(zeros(2,1),msmt_uncertainty)';
        end

        function [J, c] = msmtLinearization(obj, x_a, x_b)
            % MSMTLINEARIZATION Linearizes the rel pos msmt function
            %    Linearizes the msmt function about the current estimate
            %    for the factor variables x_a and x_b

            J = [-1 0 1 0; 0 -1 0 1];
            c = zeros(2,1);
        end

    end
end
```

```
classdef Variable
    %VARIABLE Variable node in a factor graph
    %   Describes the variable node in a factor graph, including its
    %   estimate and uncertainty, message passing, and belief updates.

    properties
        mu % n x 1 estimated value
        sigma % n x n estimate uncertainty

        eta % n x 1 information vector
        lambda % n x n precision matrix
    end

    methods
        function obj = Variable(mu,sigma)
            %VARIABLE Initializes the variable
            obj.mu = mu;
            obj.sigma = sigma;

            [obj.eta, obj.lambda] = obj.moment2canon(mu, sigma);
        end

        function msg_var2factor = msg2factor(obj, msgs_factor2var, j_factor)
            %VAR2FACTOR Fuses incoming messages and passes it to a factor
            %   Takes the product of all incoming messages to the variable,
            %   with the exception of messages from the factor the variable
            %   is sending to, and sends to the factor indicated by
            %   j_factor.
            %
            %   msgs_factor2var: row of messages to the variable from
            %   factors it is connected to
            %   j_factor: the index of the factor it is sending the message
            %   to

            % initialize eta and lambda
            n = length(obj.eta);
            eta_msg = zeros(n,1);
            lambda_msg = zeros(n,n);

            % check for messages it has received
            j_msgs = find(~cellfun(@isempty,msgs_factor2var));
            num_msgs = length(j_msgs);

            % if more than 1
            if num_msgs > 1
                for j_msg = j_msgs
                    if j_msg ~= j_factor
                        eta_msg = eta_msg + msgs_factor2var{j_msg}.eta;
                        lambda_msg = lambda_msg + msgs_factor2var{j_msg}.lambda;
                    end
```

```
            end

        % if just 1
        elseif num_msgs == 1
            eta_msg = msgs_factor2var{j_msgs}.eta;
            lambda_msg = msgs_factor2var{j_msgs}.lambda;
        % if it has received no messages
        else
            eta_msg = obj.eta;
            lambda_msg = obj.lambda;
        end

        % create message to send
        msg_var2factor = GBP.Message(eta_msg, lambda_msg);
    end

    function obj = beliefUpdate(obj, msgs_factor2var)
        %BELIEFUPDATE Fuses incoming messages and updates belief
        %   Takes the product of all incoming messages to the variable
        %   to update its belief.

        % initialize eta and lambda
        n = length(obj.eta);
        eta_belief = zeros(n,1);
        lambda_belief = zeros(n,n);

        % check for messages it has received and fuse them
        j_msgs = find(~cellfun(@isempty,msgs_factor2var));
        num_msgs = length(j_msgs);

        if num_msgs > 0
            for j_msg = j_msgs
                eta_belief = eta_belief + msgs_factor2var{j_msg}.eta;
                lambda_belief = lambda_belief + msgs_factor2var{j_msg}.lambda;
            end
        else
            error("NO INCOMING MESSAGES, BELIEF UPDATE NOT POSSIBLE")
        end

        % update belief
        obj.eta = eta_belief;
        obj.lambda = lambda_belief;

        [obj.mu, obj.sigma] = obj.canon2moment(eta_belief, lambda_belief);
    end
end

methods (Static)
    function [mu, sigma] = canon2moment(eta, lambda)
        % canon2moment converts between canon and moment forms
```

```
            % args:
            %   eta: information vector
            %   lambda: precision matrix
            %
            % returns:
            %   mu: mean vector
            %   sigma: covar matrix
            sigma = inv(lambda);
            mu = sigma*eta;
        end

        function [eta, lambda] = moment2canon(mu, sigma)
            % moment2canon Converts between moment and canon forms
            % args:
            %   mu: mean vector
            %   sigma: covar matrix
            %
            % returns:
            %   eta: information vector
            %   lambda: precision matrix
            lambda = inv(sigma);
            eta = lambda*mu;
        end
    end
end
```

```
clear all; close all;

rng(1)


%% setup
msmt_rate = 1000;
info_rate = 5;

map_all = LandmarkMap(20, 10);

%% set up truth robot
% true values
V = diag([0.025, .1*pi/180].^2);  % odometry noise
W = diag([0.025, .1*pi/180].^2);  % sensor noise

% filter values
Q = diag([0.25, 0.25, 6*pi/180]).^2;  % odometry noise
L = diag([0.25 6*pi/180].^2);  % sensor noise

% set up robot
map = map_all;

veh_gps = Bicycle(V);
veh_gps.add_driver( modCorke.RandomPath(10, false) );

sensor_gps = RangeBearingSensor(veh_gps,map, 'covar',W);

pf_gps = ParticleFilter_Proj(veh_gps, sensor_gps, Q, L, 1000, 1);

%% set up second robot
% true values
V = diag([.1, 1*pi/180].^2);  % odometry noise
W = diag([.1, 1*pi/180].^2);  % sensor noise

% filter values
Q = diag([.25, .25, 10*pi/180]).^2;  % process noise
L = diag([.5 10*pi/180].^2);  % sensor noise

% set up robot
map = map_all;

veh_cheap = Bicycle(V);
veh_cheap.add_driver( modCorke.RandomPath(10, false) );

sensor_cheap = RangeBearingSensor(veh_cheap,map, 'covar',W);

pf_cheap = ParticleFilter_Proj(veh_cheap, sensor_cheap, Q, L, 1000, msmt_rate);

%% combine into one variable
```

```
filters = [pf_gps, pf_cheap];
vehicles = [veh_gps, veh_cheap];

%% run loop

% relative position measurement
msmt_uncertainty = .1^2*eye(2);

% run filters for 1 step
for i_pf = 1:length(filters)

    filters(i_pf).run(1,true,false,false);

end

% run continuously with updates
for i = 2:100
    for i_pf = 1:length(filters)
        filters(i_pf).run(1,false,false,false);
    end

    if mod(i,info_rate) == 0 && i > 0
        filters = GBP_update(filters, msmt_uncertainty);
    end
end



%% plotting
close all
for i_pf = 1:length(filters)
    pf = filters(i_pf);
    veh = vehicles(i_pf);

    % robot motion
    figure; map.plot()
    veh.plot_xy('b');
    pf.plot_xy('r--');
    legend("landmark","truth", "estimate")
    % figure; plot(pf.std); xlabel('time step'); ylabel('standard deviation'); legend
('x', 'y', '\theta'); grid

    % covariance
    for i = 1:size(pf.x_est,1)
        std(i,:) = diag(pf.cov(:,:,i));
        std(i,3) = rad2deg(std(i,3));
    end
    figure
    subplot(2,1,1); plot(2*std(:,1:2)); xlabel( 'time step'); ylabel('Position 2\sigma
(m)'); legend('x', 'y'); grid; %set(gca, 'YScale', 'log')
```

```
    ylim([0 4*max(median(std(:,1:2)))])
    subplot(2,1,2); plot(2*std(:,3)); plot(2*std(:,3)); xlabel( 'time step'); ylabel↙
('Bearing 2\sigma (deg)'); grid; %set(gca, 'YScale', 'log')
    ylim([0 4*median(std(:,3))])

    % error
    figure;
    subplot(3,1,1); hold on; plot(abs(veh.x_hist(:,1) - pf.x_est(:,1))); plot(std(:, ↙
1)); xlabel("time step");
    ylabel("X Position Error (m)"); legend("Error","2\sigma Uncertainty"); grid; %set↙
(gca, 'YScale', 'log')
    ylim([0 3*median(std(:,1))])

    subplot(3,1,2); hold on; plot(abs(veh.x_hist(:,2) - pf.x_est(:,2))); plot(std(:, ↙
2)); xlabel("time step");
    ylabel("Y Position Error (m)"); legend("Error","2\sigma Uncertainty"); grid; %set↙
(gca, 'YScale', 'log')
    ylim([0 3*median(std(:,2))])

    subplot(3,1,3); hold on; plot(rad2deg(abs(wrapToPi(abs(veh.x_hist(:,3) - pf.x_est ↙
(:,3)))))); plot(std(:,3)); xlabel("time step");
    ylabel("Bearing Error (deg)"); legend("Error","2\sigma Uncertainty"); grid; %set↙
(gca, 'YScale', 'log')
    ylim([0 3*median(std(:,3))])
end

%% functions
function filters = GBP_update(filters, msmt_uncertainty)

    variables = [];
    factors = [];
    connections = {};

    for i_pf = 1:length(filters)

        i_pf

        pf = filters(i_pf);
        % extract position data
        mu = pf.x_est(end,1:2)'
        sigma = diag(diag(pf.cov(1:2,1:2,end)))
        var = GBP.Variable(mu, sigma);

        % initialize variables
        variables = [variables, var];

        % connect in a long chain
        if i_pf > 1
            x_i = pf.robot.x(1:2);
            x_i_prev = pf_i_prev.robot.x(1:2);
```

```
            factor = GBP.RelPosFactor(x_i_prev, x_i, msmt_uncertainty);
            factors = [factors, factor];
            connections = [connections, [i_pf-1, i_pf]];

        end

        % set to previous
        pf_i_prev = pf;
    end

    % set up graph
    factor_graph = GBP.Graph();
    factor_graph = factor_graph.addVars(variables);
    factor_graph = factor_graph.addFactors(factors, connections);

    % message passing
    for k = 1:3
        for i = 1:length(connections)
            factor_graph = factor_graph.passMessage(i, i+1);
        end
    end

    % resample
    for i = 1:length(filters)

        mu = factor_graph.variables(i).mu';
        sigma = factor_graph.variables(i).sigma;

        particle_pos = mvnrnd(mu,sigma,filters(i).nparticles);

        filters(i).x_est(1:2) = mu(1:2);
        filters(i).x(:,1:2) = particle_pos;
        filters(i).cov(1:2,1:2,end) = sigma;
    end

end
```

```
%Bicycle Car-like vehicle class
%
% This concrete class models the kinematics of a car-like vehicle (bicycle
% or Ackerman model) on a plane.  For given steering and velocity inputs it
% updates the true vehicle state and returns noise-corrupted odometry
% readings.
%
% Methods::
%   Bicycle      constructor
%   add_driver   attach a driver object to this vehicle
%   control      generate the control inputs for the vehicle
%   deriv        derivative of state given inputs
%   init         initialize vehicle state
%   f            predict next state based on odometry
%   Fx           Jacobian of f wrt x
%   Fv           Jacobian of f wrt odometry noise
%   update       update the vehicle state
%   run          run for multiple time steps
%   step         move one time step and return noisy odometry
%
% Plotting/display methods::
%   char             convert to string
%   display          display state/parameters in human readable form
%   plot             plot/animate vehicle on current figure
%   plot_xy          plot the true path of the vehicle
%   Vehicle.plotv    plot/animate a pose on current figure
%
% Properties (read/write)::
%   x               true vehicle state: x, y, theta (3x1)
%   V               odometry covariance (2x2)
%   odometry        distance moved in the last interval (2x1)
%   rdim             dimension of the robot (for drawing)
%   L               length of the vehicle (wheelbase)
%   alphalim        steering wheel limit
%   maxspeed        maximum vehicle speed
%   T               sample interval
%   verbose         verbosity
%   x_hist          history of true vehicle state (Nx3)
%   driver          reference to the driver object
%   x0              initial state, restored on init()
%
% Examples::
%
% Odometry covariance (per timstep) is
%       V = diag([0.02, 0.5*pi/180].^2);
% Create a vehicle with this noisy odometry
%       v = Bicycle( 'covar', diag([0.1 0.01].^2 );
% and display its initial state
%       v
% now apply a speed (0.2m/s) and steer angle (0.1rad) for 1 time step
```

```
%       odo = v.step(0.2, 0.1)
% where odo is the noisy odometry estimate, and the new true vehicle state
%       v
%
% We can add a driver object
%      v.add_driver( RandomPath(10) )
% which will move the vehicle within the region -10<x<10, -10<y<10 which we
% can see by
%      v.run(1000)
% which shows an animation of the vehicle moving for 1000 time steps
% between randomly selected wayoints.
%
% Notes::
% - Subclasses the MATLAB handle class which means that pass by reference semantics
%   apply.
%
% Reference::
%
%   Robotics, Vision & Control, Chap 6
%   Peter Corke,
%   Springer 2011
%
% See also RandomPath, EKF.



% Copyright (C) 1993-2017, by Peter I. Corke
%
% This file is part of The Robotics Toolbox for MATLAB (RTB).
%
% RTB is free software: you can redistribute it and/or modify
% it under the terms of the GNU Lesser General Public License as published by
% the Free Software Foundation, either version 3 of the License, or
% (at your option) any later version.
%
% RTB is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU Lesser General Public License for more details.
%
% You should have received a copy of the GNU Leser General Public License
% along with RTB.  If not, see <http://www.gnu.org/licenses/>.
%
% http://www.petercorke.com

classdef Bicycle < Vehicle

    properties
        % state
        L           % length of vehicle
```

```
        steermax
        accelmax
        vprev
        steerprev
    end

    methods

        function veh = Bicycle(varargin)
        %Bicycle.Bicycle Vehicle object constructor
        %
        % V = Bicycle(OPTIONS)  creates a Bicycle object with the kinematics of a
        % bicycle (or Ackerman) vehicle.
        %
        % Options::
        % 'steermax',M    Maximu steer angle [rad] (default 0.5)
        % 'accelmax',M    Maximum acceleration [m/s2] (default Inf)
        %--
        % 'covar',C       specify odometry covariance (2x2) (default 0)
        % 'speedmax',S    Maximum speed (default 1m/s)
        % 'L',L           Wheel base (default 1m)
        % 'x0',x0         Initial state (default (0,0,0) )
        % 'dt',T          Time interval (default 0.1)
        % 'rdim',R        Robot size as fraction of plot window (default 0.2)
        % 'verbose'       Be verbose
        %
        % Notes::
        % - The covariance is used by a "hidden" random number generator within the ↙
class.
        % - Subclasses the MATLAB handle class which means that pass by reference ↙
semantics
        %   apply.
        %
        % Notes::
        % - Subclasses the MATLAB handle class which means that pass by reference ↙
semantics
        %   apply.

            veh = veh@Vehicle(varargin{:});

            veh.x = zeros(3,1);

            opt.L = 1;
            opt.steermax = 0.5;
            opt.accelmax = Inf;

            veh = tb_optparse(opt, veh.options, veh);
            veh.vprev = 0;
            veh.x = veh.x0;
```

```
        end

        function xnext = f(veh, x, odo, w)
            %Bicycle.f Predict next state based on odometry
            %
            % XN = V.f(X, ODO) is the predicted next state XN (1x3) based on current
            % state X (1x3) and odometry ODO (1x2) = [distance, heading_change].
            %
            % XN = V.f(X, ODO, W) as above but with odometry noise W.
            %
            % Notes::
            % - Supports vectorized operation where X and XN (Nx3).
            if nargin < 4
                w = [0 0];
            end

            dd = odo(1) + w(1); dth = odo(2) + w(2);

            % straightforward code:
            % thp = x(3) + dth;
            % xnext = zeros(1,3);
            % xnext(1) = x(1) + (dd + w(1))*cos(thp);
            % xnext(2) = x(2) + (dd + w(1))*sin(thp);
            % xnext(3) = x(3) + dth + w(2);
            %
            % vectorized code:

            thp = x(:,3) + dth;
            %xnext = x + [(dd+w(1))*cos(thp)  (dd+w(1))*sin(thp) ones(size(x,1),1) ↙
*dth+w(2)];
            xnext = x + [dd*cos(thp)  dd*sin(thp) ones(size(x,1),1)*dth];
        end

        function [dx,u] = deriv(veh, t, x, u)
            %Bicycle.deriv  Time derivative of state
            %
            % DX = V.deriv(T, X, U) is the time derivative of state (3x1) at the state
            % X (3x1) with input U (2x1).
            %
            % Notes::
            % - The parameter T is ignored but  called from a continuous time ↙
integrator such as ode45 or
            %   Simulink.

            % implement acceleration limit if required
            if ~isinf(veh.accelmax)
                if (u(1) - veh.vprev)/veh.dt > veh.accelmax
                    u(1) = veh.vprev + veh.accelmax * veh.dt;
                elseif (u(1) - veh.vprev)/veh.dt < -veh.accelmax
                    u(1) = veh.vprev - veh.accelmax * veh.dt;
```

```
            end
            veh.vprev = u(1);
        end

        % implement speed and steer angle limits
        u(1) = min(veh.speedmax, max(u(1), -veh.speedmax));
        u(2) = min(veh.steermax, max(u(2), -veh.steermax));

        % compute the derivative
        dx = zeros(3,1);
        dx(1) = u(1)*cos(x(3));
        dx(2) = u(1)*sin(x(3));
        dx(3) = u(1)/veh.L * tan(u(2));
    end

    function odo = update(veh, u)
        %Bicycle.update Update the vehicle state
        %
        % ODO = V.update(U) is the true odometry value for
        % motion with U=[speed,steer].
        %
        % Notes::
        % - Appends new state to state history property x_hist.
        % - Odometry is also saved as property odometry.

        % update the state
        dx = veh.dt * veh.deriv([], veh.x, u);
        veh.x = veh.x + dx;

        % compute and save the odometry
        odo = [ norm(dx(1:2)) dx(3) ];
        veh.odometry = odo;

        veh.x_hist = [veh.x_hist; veh.x'];    % maintain history
    end


    function J = Fx(veh, x, odo)
    %Bicycle.Fx  Jacobian df/dx
    %
    % J = V.Fx(X, ODO) is the Jacobian df/dx (3x3) at the state X, for
    % odometry input ODO (1x2) = [distance, heading_change].
    %
    % See also Bicycle.f, Vehicle.Fv.
        dd = odo(1); dth = odo(2);
        thp = x(3) + dth;

        J = [
            1   0   -dd*sin(thp)
            0   1   dd*cos(thp)
```

```
                0   0   1
                ];
        end

        function J = Fv(veh, x, odo)
            %Bicycle.Fv  Jacobian df/dv
            %
            % J = V.Fv(X, ODO) is the Jacobian df/dv (3x2) at the state X, for
            % odometry input ODO (1x2) = [distance, heading_change].
            %
            % See also Bicycle.F, Vehicle.Fx.
            dd = odo(1); dth = odo(2);
            thp = x(3);

            J = [
                cos(thp)    0
                sin(thp)    0
                0           1
                ];
        end

        function s = char(veh)
        %Bicycle.char Convert to a string
        %
        % s = V.char() is a string showing vehicle parameters and state in
        % a compact human readable format.
        %
        % See also Bicycle.display.

            ss = char@Vehicle(veh);

            s = 'Bicycle object';
            s = char(s, sprintf('  L=%g, steer.max=%g, accel.max=%g', veh.L, veh.↙
steermax, veh.accelmax));
            s = char(s, ss);
        end
    end % method

end % classdef
```

```
%RangeBearingSensor Range and bearing sensor class
%
% A concrete subclass of the Sensor class that implements a range and bearing
% angle sensor that provides robot-centric measurements of landmark points in
% the world. To enable this it holds a references to a map of the world (LandmarkMap ↙
object)
% and a robot (Vehicle subclass object) that moves in SE(2).
%
% The sensor observes landmarks within its angular field of view between
% the minimum and maximum range.
%
% Methods::
%
% reading   range/bearing observation of random landmark
% h         range/bearing observation of specific landmark
% Hx        Jacobian matrix with respect to vehicle pose dh/dx
% Hp        Jacobian matrix with respect to landmark position dh/dp
% Hw        Jacobian matrix with respect to noise dh/dw
%-
% g         feature position given vehicle pose and observation
% Gx        Jacobian matrix with respect to vehicle pose dg/dx
% Gz        Jacobian matrix with respect to observation dg/dz
%
% Properties (read/write)::
% W            measurement covariance matrix (2x2)
% interval     valid measurements returned every interval'th call to reading()
% landmarklog  time history of observed landmarks
%
% Reference::
%
%   Robotics, Vision & Control, Chap 6,
%   Peter Corke,
%   Springer 2011
%
% See also Sensor, Vehicle, LandmarkMap, EKF.



% Copyright (C) 1993-2017, by Peter I. Corke
%
% This file is part of The Robotics Toolbox for MATLAB (RTB).
%
% RTB is free software: you can redistribute it and/or modify
% it under the terms of the GNU Lesser General Public License as published by
% the Free Software Foundation, either version 3 of the License, or
% (at your option) any later version.
%
% RTB is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
% GNU Lesser General Public License for more details.
%
% You should have received a copy of the GNU Leser General Public License
% along with RTB.  If not, see <http://www.gnu.org/licenses/>.
%
% http://www.petercorke.com

classdef RangeBearingSensor < Sensor

    properties
        W           % measurment covariance
        r_range     % range limits
        theta_range % angle limits

        randstream  % random stream just for Sensors

        landmarklog  % time history of observed landmarks
    end

    properties (SetAccess = private)
        count       % number of reading()s
    end

    methods

        function s = RangeBearingSensor(robot, map, varargin)
            %RangeBearingSensor.RangeBearingSensor Range and bearing sensor constructor
            %
            % S = RangeBearingSensor(VEHICLE, MAP, OPTIONS) is an object
            % representing a range and bearing angle sensor mounted on the Vehicle
            % subclass object VEHICLE and observing an environment of known landmarks
            % represented by the LandmarkMap object MAP.  The sensor covariance is W
            % (2x2) representing range and bearing covariance.
            %
            % The sensor has specified angular field of view and minimum and maximum
            % range.
            %
            % Options::
            % 'covar',W               covariance matrix (2x2)
            % 'range',xmax            maximum range of sensor
            % 'range',[xmin xmax]     minimum and maximum range of sensor
            % 'angle',TH              angular field of view, from -TH to +TH
            % 'angle',[THMIN THMAX]   detection for angles betwen THMIN
            %                         and THMAX
            % 'skip',K                return a valid reading on every K'th call
            % 'fail',[TMIN TMAX]      sensor simulates failure between
            %                         timesteps TMIN and TMAX
            % 'animate'               animate sensor readings
            %
            % See also options for Sensor constructor.
```

```
            %
            % See also RangeBearingSensor.reading, Sensor.Sensor, Vehicle, LandmarkMap, EKF.


            % call the superclass constructor
            s = s@Sensor(robot, map, varargin{:});

            s.randstream = RandStream.create( 'mt19937ar');

            opt.range = [];
            opt.angle = [];
            opt.covar = zeros(2,2);

            [opt,args] = tb_optparse(opt, varargin);

            s.W = opt.covar;
            if ~isempty(opt.range)
                if length(opt.range) == 1
                    s.r_range = [0 opt.range];
                elseif length(opt.range) == 2
                    s.r_range = opt.range;
                end
            end
            if ~isempty(opt.angle)
                if length(opt.angle) == 1
                    s.theta_range = [-opt.angle opt.angle];
                elseif length(opt.angle) == 2
                    s.theta_range = opt.angle;
                end
            end

            s.count = 0;
            s.verbose = opt.verbose;
        end

        function init(s)
            s.landmarklog = [];
        end

        function k = selectFeature(s)
            k = s.randstream.randi(s.map.nlandmarks);
        end

        function [z,jf] = reading(s)
            %RangeBearingSensor.reading Choose landmark and return observation
            %
            % [Z,K] = S.reading() is an observation of a random visible landmark where
            % Z=[R,THETA] is the range and bearing with additive Gaussian noise of
            % covariance W (property W). K is the index of the map feature that was
```

```
            % observed.
            %
            % The landmark is chosen randomly from the set of all visible landmarks,
            % those within the angular field of view and range limits.  If no valid
            % measurement, ie. no features within range, interval subsampling enabled
            % or simulated failure the return is Z=[] and K=0.
            %
            % Notes::
            % - Noise with covariance W (property W) is added to each row of Z.
            % - If 'animate' option set then show a line from the vehicle to the
            %   landmark
            % - If 'animate' option set and the angular and distance limits are set
            %   then display that region as a shaded polygon.
            % - Implements sensor failure and subsampling if specified to constructor.
            %
            % See also RangeBearingSensor.h.

            % TODO probably should return K=0 to indicated invalid

            % model a sensor that emits readings every interval samples
            s.count = s.count + 1;

            % check conditions for NOT returning a value
            z = [];
            jf = 0;
            % sample interval
            if mod(s.count, s.interval) ~= 0
                return;
            end
            % simulated failure
            if ~isempty(s.fail) && (s.count >= s.fail(1)) && (s.count <= s.fail(2))
                return;
            end

            % create a polygon to indicate the active sensing area based on range+angle ↙
limits
            if s.animate && ~isempty(s.theta_range) && ~isempty(s.r_range)
                h = findobj(gca, 'tag', 'sensor-area');
                if isempty(h)

                    th=linspace(s.theta_range(1), s.theta_range(2), 20);
                    x = s.r_range(2) * cos(th);
                    y = s.r_range(2) * sin(th);
                    if s.r_range(1) > 0
                        th = flip(th);
                        x = [x s.r_range(1) * cos(th)];
                        y = [y s.r_range(1) * sin(th)];
                    else
                        x = [x 0];
                        y = [y 0];
```

```
                end
                % no sensor zone, create one
                plot_poly([x; y], 'fillcolor', 'r', 'alpha', 0.1, 'edgecolor',↙
'none', 'animate', 'tag', 'sensor-area');
            else
                %hg = get(h, 'Parent');
                plot_poly(h, s.robot.x);

            end
        end

        if ~isempty(s.r_range)  || ~isempty(s.theta_range)
            % if range and bearing angle limits are in place look for
            % any landmarks that match criteria

            % get range/bearing to all landmarks, one per row
            z = s.h(s.robot.x');
            jf = 1:numcols(s.map.map);

            if ~isempty(s.r_range)
                % find all within range
                k = find( z(:,1) >= s.r_range(1) & z(:,1) <= s.r_range(2) );
                z = z(k,:);
                jf = jf(k);
            end
            if ~isempty(s.theta_range)
                % find all within angular range as well
                k = find( z(:,2) >= s.theta_range(1) & z(:,2) <= s.theta_range(2) ↙
);
                z = z(k,:);
                jf = jf(k);
            end

            % deal with cases for 0 or > 1 features found
            if isempty(z)
                % no landmarks found
                jf = 0;
            elseif length(k) >= 1
                % more than 1 in range, pick a random one
                i = s.randstream.randi(length(k));
                z = z(i,:);
                jf = jf(i);
            end

        else
            % randomly choose the feature
            jf = s.selectFeature();

            % compute the range and bearing from robot to feature
            z = s.h(s.robot.x', jf);
```

```
            end

            if s.verbose
                if isempty(z)
                    fprintf('Sensor:: no features\n');
                else
                    fprintf('Sensor:: feature %d: %.1f %.1f\n', jf, z);
                end
            end
            if s.animate
                s.plot(jf);
            end

            z = z';

            % add the reading to the landmark log
            s.landmarklog = [s.landmarklog jf];
        end


        function z = h(s, xv, jf)
            %RangeBearingSensor.h Landmark range and bearing
            %
            % Z = S.h(X, K) is a sensor observation (1x2), range and bearing, from ↙
vehicle at
            % pose X (1x3) to the K'th landmark.
            %
            % Z = S.h(X, P) as above but compute range and bearing to a landmark at ↙
coordinate P.
            %
            % Z = s.h(X) as above but computes range and bearing to all
            % map features.  Z has one row per landmark.
            %
            % Notes::
            % - Noise with covariance W (propertyW) is added to each row of Z.
            % - Supports vectorized operation where XV (Nx3) and Z (Nx2).
            % - The landmark is assumed visible, field of view and range liits are not
            %   applied.
            %
            % See also RangeBearingSensor.reading, RangeBearingSensor.Hx, ↙
RangeBearingSensor.Hw, RangeBearingSensor.Hp.

            % get the landmarks, one per row
            if nargin < 3
                % s.h(XV)
                xlm = s.map.map';
            elseif length(jf) == 1
                % s.h(XV, JF)
                xlm = s.map.map(:,jf)';
            else
```

```
            % s.h(XV, XF)
            xlm = jf(:)';
        end

        % Straightforward code:
        %
        % dx = xf(1) - xv(1); dy = xf(2) - xv(2);
        %
        % z = zeros(2,1);
        % z(1) = sqrt(dx^2 + dy^2);       % range measurement
        % z(2) = atan2(dy, dx) - xv(3);   % bearing measurement
        %
        % Vectorized code:

        % compute range and bearing
        dx = xlm(:,1) - xv(:,1); dy = xlm(:,2) - xv(:,2);
        z = [sqrt(dx.^2 + dy.^2) angdiff(atan2(dy, dx), xv(:,3)) ];    % range &↙
bearing measurement

        % add noise with covariance W
        z = z + s.randstream.randn(size(z)) * sqrtm(s.W) ;
    end

    function J = Hx(s, xv, jf)
        %RangeBearingSensor.Hx Jacobian dh/dx
        %
        % J = S.Hx(X, K) returns the Jacobian dh/dx (2x3) at the vehicle
        % state X (3x1) for map landmark K.
        %
        % J = S.Hx(X, P) as above but for a landmark at coordinate P.
        %
        % See also RangeBearingSensor.h.
        if length(jf) == 1
            xf = s.map.map(:,jf);
        else
            xf = jf;
        end
        if isempty(xv)
            xv = s.robot.x;
        end
        Delta = xf - xv(1:2)';
        r = norm(Delta);
        J = [
            -Delta(1)/r,    -Delta(2)/r,        0
            Delta(2)/(r^2),  -Delta(1)/(r^2),   -1
        ];
    end

    function J = Hp(s, xv, jf)
        %RangeBearingSensor.Hp Jacobian dh/dp
```

```
            %
            % J = S.Hp(X, K) is the Jacobian dh/dp (2x2) at the vehicle
            % state X (3x1) for map landmark K.
            %
            % J = S.Hp(X, P) as above but for a landmark at coordinate P (1x2).
            %
            % See also RangeBearingSensor.h.
            if length(jf) == 1
                xf = s.map.map(:,jf);
            else
                xf = jf;
            end
            Delta = xf - xv(1:2)';
            r = norm(Delta);
            J = [
                Delta(1)/r,          Delta(2)/r
                -Delta(2)/(r^2),     Delta(1)/(r^2)
            ];
        end

        function J = Hw(s, xv, jf)
            %RangeBearingSensor.Hx Jacobian dh/dw
            %
            % J = S.Hw(X, K) is the Jacobian dh/dw (2x2) at the vehicle
            % state X (3x1) for map landmark K.
            %
            % See also RangeBearingSensor.h.
            J = eye(2,2);
        end

        function xf = g(s, xv, z)
            %RangeBearingSensor.g Compute landmark location
            %
            % P = S.g(X, Z) is the world coordinate (2x1) of a feature given
            % the observation Z (1x2) from a vehicle state with X (3x1).
            %
            % See also RangeBearingSensor.Gx, RangeBearingSensor.Gz.

            range = z(1);
            bearing = z(2) + xv(3);  % bearing angle in vehicle frame

            xf = [xv(1)+range*cos(bearing); xv(2)+range*sin(bearing)];
        end

        function J = Gx(s, xv, z)
            %RangeBearingSensor.Gxv Jacobian dg/dx
            %
            % J = S.Gx(X, Z) is the Jacobian dg/dx (2x3) at the vehicle state X (3x1) ↙
for
            % sensor observation Z (2x1).
```

```
            %
            % See also RangeBearingSensor.g.
            theta = xv(3);
            r = z(1);
            bearing = z(2);
            J = [
                1,   0,   -r*sin(theta + bearing);
                0,   1,    r*cos(theta + bearing)
                ];
        end


        function J = Gz(s, xv, z)
            %RangeBearingSensor.Gz Jacobian dg/dz
            %
            % J = S.Gz(X, Z) is the Jacobian dg/dz (2x2) at the vehicle state X (3x1) ↙
for
            % sensor observation Z (2x1).
            %
            % See also RangeBearingSensor.g.
            theta = xv(3);
            r = z(1);
            bearing = z(2);
            J = [
                cos(theta + bearing),   -r*sin(theta + bearing);
                sin(theta + bearing),    r*cos(theta + bearing)
                ];
        end

        function str = char(s)
            str = char@Sensor(s);
            str = char(str, ['W = ', mat2str(s.W, 3)]);

            str = char(str, sprintf('interval %d samples', s.interval) );
            if ~isempty(s.r_range)
                str = char(str, sprintf('range: %g to %g', s.r_range) );
            end
            if ~isempty(s.theta_range)
                str = char(str, sprintf('angle: %g to %g', s.theta_range) );
            end
        end

    end % method
end % classdef
```

```
%Sensor Sensor superclass
%
% An abstract superclass to represent robot navigation sensors.
%
% Methods::
%   plot        plot a line from robot to map feature
%   display     print the parameters in human readable form
%   char        convert to string
%
% Properties::
% robot   The Vehicle object on which the sensor is mounted
% map     The PointMap object representing the landmarks around the robot
%
% Reference::
%
%   Robotics, Vision & Control,
%   Peter Corke,
%   Springer 2011
%
% See also RangeBearingSensor, EKF, Vehicle, LandmarkMap.



% Copyright (C) 1993-2017, by Peter I. Corke
%
% This file is part of The Robotics Toolbox for MATLAB (RTB).
%
% RTB is free software: you can redistribute it and/or modify
% it under the terms of the GNU Lesser General Public License as published by
% the Free Software Foundation, either version 3 of the License, or
% (at your option) any later version.
%
% RTB is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU Lesser General Public License for more details.
%
% You should have received a copy of the GNU Leser General Public License
% along with RTB.  If not, see <http://www.gnu.org/licenses/>.
%
% http://www.petercorke.com

classdef (Abstract) Sensor < handle
    % TODO, pose option, wrt vehicle

    properties
        robot
        map

        verbose
```

```
        ls
        animate      % animate sensor measurements
        interval     % measurement return subsample factor
        fail
        delay


    end

    methods

        function s = Sensor(robot, map, varargin)
        %Sensor.Sensor Sensor object constructor
        %
        % S = Sensor(VEHICLE, MAP, OPTIONS) is a sensor mounted on a vehicle
        % described by the Vehicle subclass object VEHICLE and observing landmarks
        % in a map described by the LandmarkMap class object MAP.
        %
        % Options::
        % 'animate'    animate the action of the laser scanner
        % 'ls',LS      laser scan lines drawn with style ls (default 'r-')
        % 'skip', I    return a valid reading on every I'th call
        % 'fail',T     sensor simulates failure between timesteps T=[TMIN,TMAX]
        %
        % Notes::
        % - Animation shows a ray from the vehicle position to the selected
        %   landmark.
        %

            opt.skip = 1;
            opt.animate = false;
            opt.fail =  [];
            opt.ls = 'r-';
            opt.delay = 0.1;

            [opt,args] = tb_optparse(opt, varargin);

            s.interval = opt.skip;
            s.animate = opt.animate;

            s.robot = robot;
            s.map = map;
            s.verbose = false;
            s.fail = opt.fail;
            s.ls = opt.ls;
        end

        function plot(s, jf)
        %Sensor.plot Plot sensor reading
```

```
        %
        % S.plot(J) draws a line from the robot to the J'th map feature.
        %
        % Notes::
        % - The line is drawn using the linestyle given by the property ls
        % - There is a delay given by the property delay

            if isempty(s.ls)
                return;
            end

            h = findobj(gca, 'tag', 'sensor');
            if isempty(h)
                % no sensor line, create one
                h = plot(0, 0, s.ls, 'tag', 'sensor');
            end

            % there is a sensor line animate it

            if jf == 0
                set(h, 'Visible', 'off');
            else
                xi = s.map.map(:,jf);
                set(h, 'Visible', 'on', 'XData', [s.robot.x(1), xi(1)], 'YData', [s.↙
robot.x(2), xi(2)]);
            end
            pause(s.delay);

            drawnow
        end

        function display(s)
            %Sensor.display Display status of sensor object
            %
            % S.display() displays the state of the sensor object in
            % human-readable form.
            %
            % Notes::
            % - This method is invoked implicitly at the command line when the result
            %   of an expression is a Sensor object and the command has no trailing
            %   semicolon.
            %
            % See also Sensor.char.
            loose = strcmp( get(0, 'FormatSpacing'), 'loose');
            if loose
                disp(' ');
            end
            disp([inputname(1), ' = '])
            disp( char(s) );
        end % display()
```

```
        function str = char(s)
            %Sensor.char Convert sensor parameters to a string
            %
            % s = S.char() is a string showing sensor parameters in
            % a compact human readable format.
            str = [class(s) ' sensor class:'];
            str = char(str, char(s.map));
        end

    end % method
end % classdef
```

```
%Vehicle Abstract vehicle class
%
% This abstract class models the kinematics of a mobile robot moving on
% a plane and with a pose in SE(2).  For given steering and velocity inputs it
% updates the true vehicle state and returns noise-corrupted odometry
% readings.
%
% Methods::
%   Vehicle          constructor
%   add_driver       attach a driver object to this vehicle
%   control          generate the control inputs for the vehicle
%   f                predict next state based on odometry
%   init             initialize vehicle state
%   run              run for multiple time steps
%   run2             run with control inputs
%   step             move one time step and return noisy odometry
%   update           update the vehicle state
%
% Plotting/display methods::
%   char             convert to string
%   display          display state/parameters in human readable form
%   plot             plot/animate vehicle on current figure
%   plot_xy          plot the true path of the vehicle
%   Vehicle.plotv    plot/animate a pose on current figure
%
% Properties (read/write)::
%   x               true vehicle state: x, y, theta (3x1)
%   V               odometry covariance (2x2)
%   odometry        distance moved in the last interval (2x1)
%   rdim             dimension of the robot (for drawing)
%   L               length of the vehicle (wheelbase)
%   alphalim        steering wheel limit
%   speedmax        maximum vehicle speed
%   T               sample interval
%   verbose         verbosity
%   x_hist          history of true vehicle state (Nx3)
%   driver          reference to the driver object
%   x0              initial state, restored on init()
%
% Examples::
%
% If veh is an instance of a Vehicle class then we can add a driver object
%      veh.add_driver( RandomPath(10) )
% which will move the vehicle within the region -10<x<10, -10<y<10 which we
% can see by
%      veh.run(1000)
% which shows an animation of the vehicle moving for 1000 time steps
% between randomly selected wayoints.
%
% Notes::
```

```
% - Subclass of the MATLAB handle class which means that pass by reference semantics
%   apply.
%
% Reference::
%
%   Robotics, Vision & Control, Chap 6
%   Peter Corke,
%   Springer 2011
%
% See also Bicycle, Unicycle, RandomPath, EKF.



% Copyright (C) 1993-2017, by Peter I. Corke
%
% This file is part of The Robotics Toolbox for MATLAB (RTB).
%
% RTB is free software: you can redistribute it and/or modify
% it under the terms of the GNU Lesser General Public License as published by
% the Free Software Foundation, either version 3 of the License, or
% (at your option) any later version.
%
% RTB is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU Lesser General Public License for more details.
%
% You should have received a copy of the GNU Leser General Public License
% along with RTB.  If not, see <http://www.gnu.org/licenses/>.
%
% http://www.petercorke.com

classdef Vehicle < handle

    properties
        % state
        x           % true state (x,y,theta)
        x_hist      % x history

        % parameters

        speedmax    % maximum speed
        dim         % dimension of the world -dim -> +dim in x and y
        rdim    % dimension of the robot
        dt           % sample interval
        V           % odometry covariance
        odometry    % distance moved in last interval
        verbose
        driver      % driver object
        x0          % initial state
```

```
        options

        vhandle      % handle to vehicle graphics object
        vtrail       % vehicle trail
    end

    methods(Abstract)
        f
    end

    methods

        function veh = Vehicle(varargin)
        %Vehicle Vehicle object constructor
        %
        % V = Vehicle(OPTIONS) creates a Vehicle object that implements the
        % kinematic model of a wheeled vehicle.
        %
        % Options::
        % 'covar',C       specify odometry covariance (2x2) (default 0)
        % 'speedmax',S    Maximum speed (default 1m/s)
        % 'L',L           Wheel base (default 1m)
        % 'x0',x0         Initial state (default (0,0,0) )
        % 'dt',T          Time interval (default 0.1)
        % 'rdim',R        Robot size as fraction of plot window (default 0.2)
        % 'verbose'       Be verbose
        %
        % Notes::
        % - The covariance is used by a "hidden" random number generator within the ↙
class.
        % - Subclasses the MATLAB handle class which means that pass by reference ↙
semantics
        %   apply.


            % vehicle common
            opt.covar = [];
            opt.rdim = 0.2;
            opt.dt = 0.1;
            opt.x0 = zeros(3,1);
            opt.speedmax = 1;
            opt.vhandle = [];

            [opt,args] = tb_optparse(opt, varargin);

            veh.V = opt.covar;
            veh.rdim = opt.rdim;
            veh.dt = opt.dt;
            veh.x0 = opt.x0(:);
            assert(isvec(veh.x0, 3), 'Initial configuration must be a 3-vector');
```

```
            veh.speedmax = opt.speedmax;
            veh.options = args;   % unused options go back to the subclass
            veh.vhandle = opt.vhandle;
            veh.x_hist = [];
        end

        function init(veh, x0)
            %Vehicle.init Reset state
            %
            % V.init() sets the state V.x := V.x0, initializes the driver
            % object (if attached) and clears the history.
            %
            % V.init(X0) as above but the state is initialized to X0.

            % TODO: should this be called from run?

            if nargin > 1
                veh.x = x0(:);
            else
                veh.x = veh.x0;
            end
            veh.x_hist = [];

            if ~isempty(veh.driver)
                veh.driver.init();
            end

            veh.vhandle = [];
        end

        function yy = path(veh, t, u, y0)
            %Vehicle.path Compute path for constant inputs
            %
            % XF = V.path(TF, U) is the final state of the vehicle (3x1) from the ↙
initial
            % state (0,0,0) with the control inputs U (vehicle specific).  TF is  a ↙
scalar to
            % specify the total integration time.
            %
            % XP = V.path(TV, U) is the trajectory of the vehicle (Nx3) from the ↙
initial
            % state (0,0,0) with the control inputs U (vehicle specific).  T is a ↙
vector (N) of
            % times for which elements of the trajectory will be computed.
            %
            % XP = V.path(T, U, X0) as above but specify the initial state.
            %
            % Notes::
            % - Integration is performed using ODE45.
            % - The ODE being integrated is given by the deriv method of the vehicle ↙
```

```
object.
            %
            % See also ODE45.

                if length(t) == 1
                    tt = [0 t];
                else
                    tt = t;
                end

                if nargin < 4
                    y0 = [0 0 0];
                end
                out = ode45( @(t,y) veh.deriv(t, y, u), tt, y0);

                y = out.y';
                if nargout == 0
                    plot(y(:,1), y(:,2));
                    grid on
                    xlabel('X'); ylabel('Y')
                else
                    yy = y;
                    if length(t) == 1
                        % if scalar time given, just return final state
                        yy = yy(end,:);
                    end
                end
        end

        function add_driver(veh, driver)
            %Vehicle.add_driver Add a driver for the vehicle
            %
            % V.add_driver(D) connects a driver object D to the vehicle.  The driver
            % object has one public method:
            %        [speed, steer] = D.demand();
            % that returns a speed and steer angle.
            %
            % Notes::
            % - The Vehicle.step() method invokes the driver if one is attached.
            %
            % See also Vehicle.step, RandomPath.
            veh.driver = driver;
            driver.veh = veh;
        end

        function odo = update(veh, u)
            %Vehicle.update Update the vehicle state
            %
            % ODO = V.update(U) is the true odometry value for
            % motion with U=[speed,steer].
```

```
            %
            % Notes::
            % - Appends new state to state history property x_hist.
            % - Odometry is also saved as property odometry.

            xp = veh.x; % previous state
            veh.x(1) = veh.x(1) + u(1)*veh.dt*cos(veh.x(3));
            veh.x(2) = veh.x(2) + u(1)*veh.dt*sin(veh.x(3));
            veh.x(3) = veh.x(3) + u(1)*veh.dt/veh.L * u(2);
            odo = [colnorm(veh.x(1:2)-xp(1:2)) veh.x(3)-xp(3)];
            veh.odometry = odo;

            veh.x_hist = [veh.x_hist; veh.x'];    % maintain history
        end

        function odo = step(veh, varargin)
            %Vehicle.step Advance one timestep
            %
            % ODO = V.step(SPEED, STEER) updates the vehicle state for one timestep
            % of motion at specified SPEED and STEER angle, and returns noisy odometry.
            %
            % ODO = V.step() updates the vehicle state for one timestep of motion and
            % returns noisy odometry.  If a "driver" is attached then its DEMAND() ↙
method
            % is invoked to compute speed and steer angle.  If no driver is attached
            % then speed and steer angle are assumed to be zero.
            %
            % Notes::
            % - Noise covariance is the property V.
            %
            % See also Vehicle.control, Vehicle.update, Vehicle.add_driver.

            % get the control input to the vehicle from either passed demand or driver
            u = veh.control(varargin{:});

            % compute the true odometry and update the state
            odo = veh.update(u);

            % add noise to the odometry
            if ~isempty(veh.V)
                odo = veh.odometry + randn(1,2)*sqrtm(veh.V);
            end
        end

        function u = control(veh, speed, steer)
            %Vehicle.control Compute the control input to vehicle
            %
            % U = V.control(SPEED, STEER) is a control input (1x2) = [speed,steer]
            % based on provided controls SPEED,STEER to which speed and steering angle
            % limits have been applied.
```

```
        %
        % U = V.control() as above but demand originates with a "driver" object if
        % one is attached, the driver's DEMAND() method is invoked. If no driver is
        % attached then speed and steer angle are assumed to be zero.
        %
        % See also Vehicle.step, RandomPath.
        if nargin < 2
            % if no explicit demand, and a driver is attached, use
            % it to provide demand
            if ~isempty(veh.driver)
                [speed, steer] = veh.driver.demand();
            else
                % no demand, do something safe
                speed = 0;
                steer = 0;
            end
        end

        % clip the speed
        if isempty(veh.speedmax)
            u(1) = speed;
        else
            u(1) = min(veh.speedmax, max(-veh.speedmax, speed));
        end

        % clip the steering angle
        if isprop(veh, 'steermax') && ~isempty(veh.steermax)
            u(2) = max(-veh.steermax, min(veh.steermax, steer));
        else
            u(2) = steer;
        end
    end

    function p = run(veh, nsteps)
        %Vehicle.run Run the vehicle simulation
        %
        % V.run(N) runs the vehicle model for N timesteps and plots
        % the vehicle pose at each step.
        %
        % P = V.run(N) runs the vehicle simulation for N timesteps and
        % return the state history (Nx3) without plotting.  Each row
        % is (x,y,theta).
        %
        % See also Vehicle.step, Vehicle.run2.

        if nargin < 2
            nsteps = 1000;
        end
        if ~isempty(veh.driver)
            veh.driver.init()
```

```
        end
        %veh.clear();
        if ~isempty(veh.driver)
            veh.driver.plot();
        end

        veh.plot();
        for i=1:nsteps
            veh.step();
            if nargout == 0
                % if no output arguments then plot each step
                veh.plot();
                drawnow
            end
        end
        p = veh.x_hist;
    end

    % TODO run and run2 should become superclass methods...

    function p = run2(veh, T, x0, speed, steer)
        %Vehicle.run2 Run the vehicle simulation with control inputs
        %
        % P = V.run2(T, X0, SPEED, STEER) runs the vehicle model for a time T with
        % speed SPEED and steering angle STEER.  P (Nx3) is the path followed and
        % each row is (x,y,theta).
        %
        % Notes::
        % - Faster and more specific version of run() method.
        % - Used by the RRT planner.
        %
        % See also Vehicle.run, Vehicle.step, RRT.
        veh.init(x0);

        for i=1:(T/veh.dt)
            veh.update([speed steer]);
        end
        p = veh.x_hist;
    end

    function h = plot(veh, varargin)
    %Vehicle.plot Plot vehicle
    %
    % The vehicle is depicted graphically as a narrow triangle that travels
    % "point first" and has a length V.rdim.
    %
    % V.plot(OPTIONS) plots the vehicle on the current axes at a pose given by
    % the current robot state.  If the vehicle has been previously plotted its
    % pose is updated.
    %
```

```
        % V.plot(X, OPTIONS) as above but the robot pose is given by X (1x3).
        %
        % H = V.plotv(X, OPTIONS) draws a representation of a ground robot as an
        % oriented triangle with pose X (1x3) [x,y,theta].  H is a graphics handle.
        %
        % V.plotv(H, X) as above but updates the pose of the graphic represented
        % by the handle H to pose X.
        %
        % Options::
        % 'scale',S    Draw vehicle with length S x maximum axis dimension
        % 'size',S     Draw vehicle with length S
        % 'color',C    Color of vehicle.
        % 'fill'       Filled
        % 'trail',S    Trail with line style S, use line() name-value pairs
        %
        % Example::
        %          veh.plot('trail', {'Color', 'r', 'Marker', 'o', 'MarkerFaceColor', ↙
'r', 'MarkerEdgeColor', 'r', 'MarkerSize', 3})

        % Notes::
        % - The last two calls are useful if animating multiple robots in the same
        %   figure.
        %
        % See also Vehicle.plotv, plot_vehicle.


            if isempty(veh.vhandle)
                veh.vhandle = Vehicle.plotv(veh.x, varargin{:});
            end

            if ~isempty(varargin) && isnumeric(varargin{1})
                % V.plot(X)
                pos = varargin{1};  % use passed value
            else
                % V.plot()
                pos = veh.x;     % use current state
            end

            % animate it
            Vehicle.plotv(veh.vhandle, pos);

            end


        function out = plot_xy(veh, varargin)
            %Vehicle.plot_xy Plots true path followed by vehicle
            %
            % V.plot_xy() plots the true xy-plane path followed by the vehicle.
            %
            % V.plot_xy(LS) as above but the line style arguments LS are passed
```

```
        % to plot.
        %
        % Notes::
        % - The path is extracted from the x_hist property.

        xyt = veh.x_hist;
        if nargout == 0
            plot(xyt(:,1), xyt(:,2), varargin{:});
        else
            out = xyt;
        end
    end

    function verbosity(veh, v)
    %Vehicle.verbosity Set verbosity
    %
    % V.verbosity(A) set verbosity to A.  A=0 means silent.
        veh.verbose = v;
    end

    function display(nav)
    %Vehicle.display Display vehicle parameters and state
    %
    % V.display() displays vehicle parameters and state in compact
    % human readable form.
    %
    % Notes::
    % - This method is invoked implicitly at the command line when the result
    %   of an expression is a Vehicle object and the command has no trailing
    %   semicolon.
    %
    % See also Vehicle.char.

        loose = strcmp( get(0, 'FormatSpacing'), 'loose');
        if loose
            disp(' ');
        end
        disp([inputname(1), ' = '])
        disp( char(nav) );
    end % display()

    function s = char(veh)
    %Vehicle.char Convert to string
    %
    % s = V.char() is a string showing vehicle parameters and state in
    % a compact human readable format.
    %
    % See also Vehicle.display.

        s = '  Superclass: Vehicle';
```

```
            s = char(s, sprintf( ...
            '    max speed=%g, dT=%g, nhist=%d', ...
                veh.speedmax, veh.dt, ...
                numrows(veh.x_hist)));
            if ~isempty(veh.V)
                s = char(s, sprintf( ...
                '    V=(%g, %g)', ...
                    veh.V(1,1), veh.V(2,2)));
            end
            s = char(s, sprintf('    configuration: x=%g, y=%g, theta=%g', veh.x));
            if ~isempty(veh.driver)
                s = char(s, '    driven by::');
                s = char(s, [['      '; '      '] char(veh.driver)]);
            end
        end

    end % method

    methods(Static)

        function h = plotv(varargin)
        %Vehicle.plotv Plot ground vehicle pose
        %
        % H = Vehicle.plotv(X, OPTIONS) draws a representation of a ground robot as an
        % oriented triangle with pose X (1x3) [x,y,theta].  H is a graphics handle.
        % If X (Nx3) is a matrix it is considered to represent a trajectory in which ↙
case
        % the vehicle graphic is animated.
        %
        % Vehicle.plotv(H, X) as above but updates the pose of the graphic represented
        % by the handle H to pose X.
        %
        % Options::
        % 'scale',S       Draw vehicle with length S x maximum axis dimension
        % 'size',S        Draw vehicle with length S
        % 'fillcolor',C   Color of vehicle.
        % 'fps',F         Frames per second in animation mode (default 10)
        %
        % Example::
        %
        % Generate some path 3xN
        %         p = PRM.plan(start, goal);
        % Set the axis dimensions to stop them rescaling for every point on the path
        %         axis([-5 5 -5 5]);
        %
        % Now invoke the static method
        %         Vehicle.plotv(p);
        %
        % Notes::
        % - This is a class method.
```

```
        %
        % See also Vehicle.plot.

        if isstruct(varargin{1})
            plot_vehicle(varargin{2}, 'handle', varargin{1});
        else
            h = plot_vehicle(varargin{1}, 'fillcolor', 'b', 'alpha', 0.5);
        end

        end
    end % static methods

end % classdef
```