DR. MICHAEL P. HAYES

ENEL442

MOBILE ROBOTICS LECTURE NOTES

V50

ELECTRICAL AND COMPUTER ENGINEERING

Contents

Glossary and definitions

1 Notation

$\mathbf{x}$ vector

$\check{\mathbf{x}}$ predicted vector (from model)

$\acute{\mathbf{x}}$ inferred vector (from measurement)

$\hat{\mathbf{x}}$ estimated vector

$\mathbf{M}$ matrix

$\check{\mathbf{M}}$ predicted matrix

$\hat{\mathbf{M}}$ estimated matrix

X random variable

$\mu_X = \mathrm{E}\{X\}$ expected value (mean) of random variable X

$\sigma_X^2 = \mathrm{Var}\{X\} = \mathrm{E}\left\{(X - \mathrm{E}\{X\})^2\right\}$ variance of random variable X

σ_X standard deviation of random variable X

$F_X(x)$ cumulative probability density distribution for random variable X

$f_X(x)$ probability density distribution for random variable X

$f_{X,Y}(x,y)$ joint probability density distribution for random variables X and Y

$f_{X|Y}(x|y)$ conditional probability density distribution for random variable X given random variable Y

$\rho = \rho_{XY}$ correlation coefficient between random variables X and Y

$\mu_{\mathbf{X}} = \mathrm{E}\{\mathbf{X}\}$ mean of random vector $\mathbf{X}$

$\Sigma_{\mathbf{X}} = \mathrm{E}\left\{(\mathbf{X} - \mathrm{E}\{\mathbf{X}\})(\mathbf{X} - \mathrm{E}\{\mathbf{X}\})^{\mathrm{T}}\right\}$ covariance matrix of random vector $\mathbf{X}$

$\text{bel}(\mathbf{x}_t)$ belief of $\mathbf{x}_t$ (a posteriori)

$\breve{\text{bel}}(\mathbf{x}_t)$ predicted belief of $\mathbf{x}_t$ (a priori)

w_1, w_2 weights

R_1, R_2 range measurements

$\hat{R}$ estimated range

$\mathbf{K}_t$ Kalman gain vector at time-step t

A state transition matrix

B input matrix

C output matrix

D feed-through matrix

x state vector

u control action vector

z measurement vector

$\mathbf{v}_t$ measurement noise vector

$\mathbf{w}_t$ process noise vector

$\mathbf{\Sigma}_v$ measurement noise covariance matrix

$\mathbf{\Sigma}_w$ process noise covariance matrix

$\hat{\mathbf{P}}_t$ estimated state covariance matrix

Δt sampling period

I identity matrix

J Jacobian matrix

2 *Glossary*

x State vector

x̆ Predicted state vector

x̂ Estimated state vector

z Measurement vector

v Measurement noise vector

w Process noise vector

m Map entry

A State transition matrix

B Input matrix

C Output matrix

D Feed-through matrix

K Kalman gain vector

P State covariance matrix

$\mathbf{\Sigma}_v$ Measurement noise covariance matrix

$\mathbf{\Sigma}_w$ Process noise covariance matrix

$\check{\mathbf{P}}$ Predicted state covariance matrix

$\hat{\mathbf{P}}$ Estimated state covariance matrix

3 *Definitions*

Kinematics is the study of classical mechanics which describes the motion of points, bodies, and systems of bodies without consideration of the causes of motion[1].

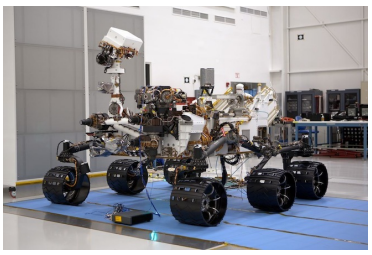
Dynamics concerns with the study of forces and torques and their effect on motion (as opposed to kinematics, which studies the motion of objects without reference to its causes).

Pose describes the location and orientation. For a robot it is a subset of its configuration (which also describes the angles of joints. etc. of manipulators).

Holonomic drives have three degrees of freedom and provide motion in any direction with independent rotation. In comparison, differential drives have only two degrees of freedom.

[1] From the Greek kinema for movement or motion.

1
Introduction



Mobile robotics is difficult since it requires mastery of many disciplines, including:

- Kinematics
- Dynamics
- Locomotion
- Sensors
- Actuators
- Electronics
- Power electronics
- Power management
- Control theory
- Machine vision
- Perception
- Computer hardware
- Computer software
- Computer networking
- Statistics
- ...

1 Probabilistic robotics

Surprisingly, one of the key technologies for autonomous mobile robotics is statistics, in particular the application of Bayes' theorem. The main reason is that the environments of robots are inherently unpredictable[1], especially for mobile robots. Moreover, sensors are subject to physical limitations and their measurements are noisy and sometimes ambiguous. Similarly, actuators are also unpredictable; wheels slip, gears have backlash, motors can fail. Furthermore, the complexity of robot environments requires approximate models[2] that can be analysed in real-time. These algorithmic approximations introduce further uncertainty.

Probabilistic algorithms have an advantage in that they can handle multiple hypotheses. Instead of relying on a single best guess, probabilistic algorithms represent information by probability distributions.

[1] Plants grow, snow covers things, people move.

[2] Consider a robot environment with 20 objects each having 10 states. The number of permutations (with replacement) is 10^{20}.

2 The Turtlebot2

Figure 1.1: The Turtlebot2, with a Kobuki base and Kinect 3D sensor.

The TurtleBot2 is a low-cost personal robot kit[3]. It is comprised of three components:

- Kobuki mobile base
- Kinect 3-D sensor[4]
- Computer

[3] Developed by Willow Garage.

[4] This fakes a simple laser scanner.

The original Turtlebot robot used the iRobot Create vacuum cleaner base. This was superseded by the Kobuki (see Figure 1.1); a low-cost mobile research base designed for education and robotics research. It has independent drive for each wheel but is not holonomic[5] unlike the URANUS robot shown in Figure 1.2.

The Turtlebot2 has the following sensors:

- Wheel drop sensor for each wheel
- Three front bumper sensors: left, centre, right)
- Three front cliff sensors: left, centre, right (to prevent falling off stairs, ledges)
- Odometry sensor for each wheel
- Gyroscope (1 axis) for orientation

The Kobuki control outputs are:

- Wheel motors (left and right)
- Bicolour LEDs (two)
- Audible alarm

[5] It only two of the three degrees of motional freedom.

Figure 1.2: "URANUS" omnidirectional mobile robot using Mecanum wheels. Designed in 1985 by Gregg Podnar at the Robotics Institute of Carnegie-Mellon University.

2.1 *Turtlebot SLAM demo*

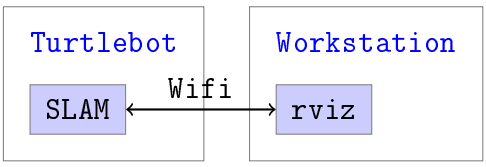


Figure 1.3: Simultaneous Localisation and Mapping (SLAM) algorithm running on Turtlebot controlled by rviz running on remote workstation. rviz sends goals and displays the map.

3 Summary

1. Robot environments are unpredictable
2. Robot environments are complicated and approximate models are required
3. Sensor information is ambiguous and noisy
4. Actuators are unreliable and noisy
5. Probabilistic models can support multiple hypotheses
6. Heuristics are required to handle combinatorial explosion
7. Why is the Turtlebot not holonomic?
8. How do the Turtlebot cliff sensors work?
9. How does the Turtlebot determine how far away things are?

4 Further reading

- http://www.probabilistic-robotics.org/
- "Probabilistic Robotics", S. Thrun, W. Burgard, and D. Fox. MIT Press, 2005. *This is the book!*
- "Autonomous Mobile Robots", R. Siegwart and I. Nourbaksh, MIT Press, 2004. *This is simpler but with more arm waving.*

2
ROS introduction

ROS is an open-source, meta-operating system[1] for a robot[2]. It operates on a heterogeneous network of computers using TCP/IP over ethernet, WiFi, etc. It provides hardware abstraction, low-level device control, message-passing, etc. It also provides tools and libraries for running and developing code across multiple computers, including package management.

[1] It currently runs on top of a Unix operating system.
[2] ROS is similar in some respects to 'robot frameworks,' such as Player, YARP, Orocos, CARMEN, Orca, MOOS, and Microsoft Robotics Studio.

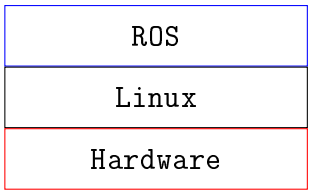


Figure 2.1: ROS is meta operating system, usually in conjunction with Linux.

1 *ROS computation graph*

The ROS Computation Graph[3] is a peer-to-peer network of processes (*nodes*) that communicate using *messages*.

[3] A picture of the graph is generated by the program `rxgraph`. This displays the nodes currently running and the topics that connect them.

1.1 *Nodes*

A node performs an aspect of computation, for example,

- laser range-finder control
- motor control
- localisation
- path planning

There are two special nodes:

Master provides name registration and lookup[4].

Parameter Server allows data to be stored by key in a central repository[5].

[4] Similar to a Domain Name Server (DNS). It tracks publishers and subscribers of topics as well as services.
[5] Currently the Parameter Server is part of the Master.

1.2 *Messages*

Nodes communicate using message passing. Messages are data structures with typed fields. These can be primitive types include integer, floating point, Boolean, etc., arrays of primitive types, and arbitrarily nested structures and arrays (like C structs). There are two ways of passing messages:

Topics provide an asynchronous publish/subscribe model. A node transmits a message by publishing it with a topic identifying the message content (for example, laser-scan data). Other nodes subscribed to the topic will receive the message.

Services provide a synchronous request/reply model similar to a remote procedure call. A node sends a request message and then waits for a reply message (for example, requesting a map).

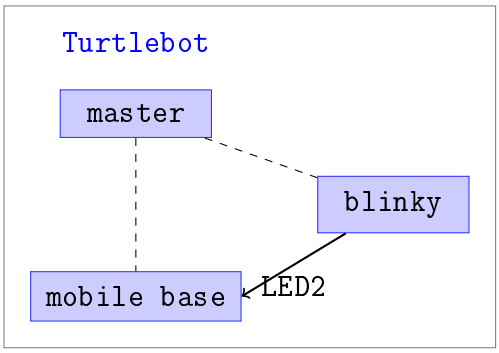


Figure 2.2: Node communication example using the LED2 topic. The blinky node publishes messages on the LED2 topic subscribed to by the base node.

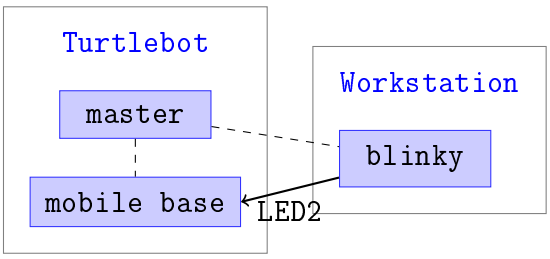


Figure 2.3: Node communication example using the LED2 topic with the nodes running on different computers.

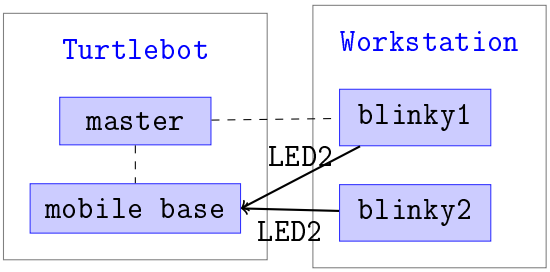


Figure 2.4: Node communication example using the LED2 topic with two publishers.

1.3 Message bags

Messages can be saved in *bag*[6] files using the rosbag node. Message bags are useful for developing and testing algorithms since the data can be recorded once and replayed[7].

[6] For example, to record a bag called `wander.bag` with messages from the topics `imu` and `scan`, use `rosbag record -O wander.bag /imu /scan`.
[7] `rosbag play wander.bag`.

2 ROS distributions

ROS Jade Turtle is the latest ROS release.

ROS Indigo Igloo is the recommended current distribution [8] targetted for Ubuntu 14.04 LTS (Trusty) .

ROS Hydro Medusa is the previous recommended distribution. Much documentation still refers to this.

[8] `http://wiki.ros.org/indigo`

3 ROS commands

There are many ROS commands, the common ones being:

rospack displays information about packages.

rosnode displays information about nodes.

rostopic displays information about topics.

rossrv displays information about services.

rosmsg displays information about message types.

rosrun runs an executable in a package.

roslaunch launches multiple nodes.

rosbag records messages to a bag or replays messages from a bag.

The nodes running on a system can be determined with the `rosnode` command, for example:

```
$ rosnode list
/app_manager
/bumper2pointcloud
/cmd_vel_mux
/diagnostic_aggregator
/gateway
/mobile_base
/mobile_base_nodelet_manager
/robot_state_publisher
```

```
/rosout
/zeroconf/zeroconf
```

All the message topics can be determined with the `rostopic` command, for example:

```
$ rostopic list
/cmd_vel_mux/active
/cmd_vel_mux/input/navi
/cmd_vel_mux/input/safety_controller
/cmd_vel_mux/input/teleop
/cmd_vel_mux/parameter_descriptions
/cmd_vel_mux/parameter_updates
/diagnostics
/diagnostics_agg
/diagnostics_toplevel_state
/gateway/force_update
/gateway/gateway_info
/joint_states
/mobile_base/commands/controller_info
/mobile_base/commands/digital_output
/mobile_base/commands/external_power
/mobile_base/commands/led1
/mobile_base/commands/led2
/mobile_base/commands/motor_power
/mobile_base/commands/reset_odometry
/mobile_base/commands/sound
/mobile_base/commands/velocity
/mobile_base/controller_info
/mobile_base/debug/raw_control_command
/mobile_base/debug/raw_data_command
/mobile_base/debug/raw_data_stream
/mobile_base/events/bumper
/mobile_base/events/button
/mobile_base/events/cliff
/mobile_base/events/digital_input
/mobile_base/events/power_system
/mobile_base/events/robot_state
/mobile_base/events/wheel_drop
/mobile_base/sensors/bumper_pointcloud
/mobile_base/sensors/core
/mobile_base/sensors/dock_ir
/mobile_base/sensors/imu_data
/mobile_base/sensors/imu_data_raw
/mobile_base/version_info
/mobile_base_nodelet_manager/bond
/odom
/rosout
/rosout_agg
```

```
/tf
/turtlebot/app_list
/zeroconf/lost_connections
/zeroconf/new_connections
```

The format of a message can be printed using `rosmsg`, for example, a Pose message is a structure comprised of a position and an orientation[9]

[9] Quaternions are used to avoid gimbal lock that can occur when representing orientations with Euler angles. They simply specify a single rotation around an arbitrary vector.

```
$ rosmsg show Pose
[geometry_msgs/Pose]:
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

4 ROS packages

ROS comprises a large number of packages[10]; each package might contain ROS nodes, a dataset, configuration files, etc. Example packages[11] include:

[10] Early versions of ROS (using rosbuild) used stacks that are similar to metapackages

[11] Use `rospack list-names` to find installed ROS packages.

amcl a localisation algorithm given a map.

gmapping a SLAM algorithm for producing a map.

map-server a server for map data.

tf an infrastructure for coordinate transforms.

base_local_planner a local navigation algorithm.

global_planner a global path planning algorithm.

4.1 The tf package

A robot typically has many 3D coordinate frames such as a world frame, base frame, IMU frame, gripper frame, etc. tf keeps track of all the frame relationships[12] over time, and can answer questions like:

[12] Using a transform tree.

- Where was the base frame relative to the world frame, 5 seconds ago?
- What is the pose of the object in the gripper relative to the base?

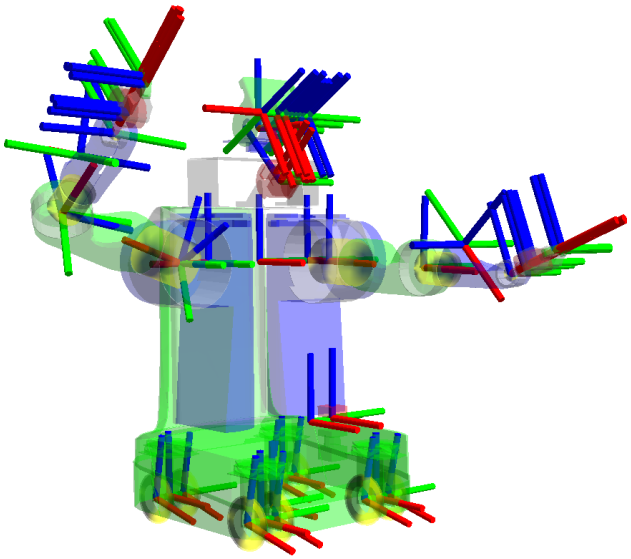


Figure 2.5: Visualising the frames of a robot. Red (x), green (y), and blue (z).

- What is the current pose of the base frame in the map frame?

5 *Launchers*

A launcher is a file[13] that is used for starting multiple ROS nodes and setting parameters on the Parameter Server. It is used by the `roslaunch` program.

Names of services and topics can be remapped. For example, say we wanted to use a laser scanner instead of the Kinect. Rather than changing our applications, we can remap the message names.

[13] An XML file with a `.launch` suffix.

```
<launch>
  <!-- turtlebot_teleop_key already has its own built in
     velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key"
     name="turtlebot_teleop_keyboard"  output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="
       cmd_vel_mux/input/teleop"/>
  </node>
</launch>
```

Notes:

1. The launcher defines two parameters with default values, `scale_linear` and `scale_angular`.
2. The launcher remaps the topic name `cmd_vel` to `cmd_vel_mux`.

6 Message passing implementation

Messages are passed peer to peer and not via the master node[14]. When a node publishes or subscribes to a topic, it communicates with the master. The master thus keeps track of all the topics and whoever is subscribing or publishing.

When a topic has both a publisher and a subscriber, the master informs publishers and subscribers of each others existence. Each publisher can then send messages to the subscribers. Note, no messages are sent on a topic unless there is at least one publisher and one subscriber.

[14] http://wiki.ros.org/Master

7 Exercises

1. How does a node determine which node publishes a topic?
2. Is the master node required for message transfers?
3. How are parameters specified?
4. What does a heterogenous operating system mean?
5. What does a meta operating system mean?
6. How does a topic differ from a service?
7. What is a ROS node?
8. What is a launcher?
9. What is the point of message name remapping?
10. Give examples of four different ROS commands and what they do.
11. What is a bag file and state an advantage of using one.
12. What does the tf package do?
13. How many floating point words are required to store a Pose message?
14. Why are quaternions used for Pose messages?
15. In broad terms how does rosbag work?

3
ROS and the Turtlebot2

ROS nodes can be written in a number of languages with the appropriate client library[1]. The client library handles all the data transport between nodes. This lecture shows some examples for controlling the Turtlebot.

[1] Python client support is provided by the `rospy` module.

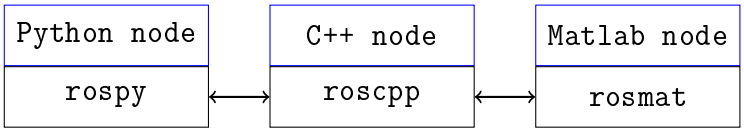


Figure 3.1: ROS nodes can be written in different languages using the appropriate ROS client library.

1 *Message publishing*

Here's an Python snippet to show how messages can be published to control a LED on the Kobuki base:

```
from rospy import Publisher
from kobuki_msgs.msg import Led

# Create publisher object to control LED2 on Kobuki base
pub = Publisher('/mobile_base/commands/led2', Led)

# Create LED message
msg = Led()
msg.value = msg.Green

# Publish LED message
pub.publish(msg)
```

Notes:

1. The format of each message is predefined, e.g., `Led` in the module `kobuki_msgs.msg`.
2. `Publisher` notifies the master node that this node is publishing messages on the topic `/mobile_base/events/wheel_drop`.
3. The message topics use a hierarchical name-space[2].

[2] Similar to a filesystem.

Here's another example:

```
from rospy import Publisher
from kobuki_msgs.msg import Led

# Create publisher object to control LED2 on Kobuki base
pub = Publisher('/mobile_base/commands/led2', Led)

# Create LED message object
msg = Led()

# Publish message to turn LED2 green
msg.value = msg.Green
pub.publish(msg)

# Publish message to turn LED2 red
msg.value = msg.Red
pub.publish(msg)
```

2 *Message subscribing*

Sensor reading is slightly more tricky since sensor events are asynchronous[3]. This is handled by registering a callback function whenever a desired message topic is subscribed to. For example,

[3] They can happen at any time.

```
from rospy import Subscriber, signal_shutdown
from kobuki_msgs.msg import WheelDropEvent

def wheel_drop_callback(data):

    # Stop Robot's motion if picked up
    if data.state != WheelDropEvent.RAISED:
        signal_shutdown("Wheel %d drop" % data.wheel)

Subscriber('/mobile_base/events/wheel_drop', WheelDropEvent,
   wheel_drop_callback)
```

Notes:

- `Subscriber` registers the topic `/mobile_base/events/wheel_drop` with the master node.
- The callback function `wheel_drop_callback` gets called when a message topic `/mobile_base/events/wheel_drop` is received. The parameter is the message specifying which wheel and which transition.
- In this example the program is aborted when either of the wheel's drop.

3 *Movement control*

Movement is controlled by publishing speed messages. The messages specify the linear speed and the angular speed. Usually a Twist message is used that can specify all six linear and angular speeds. For the Turtlebot, only the angular speed around the z-axis and the linear speed along the x-axis is interpreted.

These messages are not usually sent to the base directly but to a velocity multiplexer node, for example:

```
from rospy import Publisher
from geometry_msgs.msg import Twist

# Create publisher object
twistpub = Publisher('/cmd_vel_mux/input/navi', Twist)

# Create twist message
twist = Twist()
twist.linear.x = linear_speed
twist.angular.z = angular_speed

# Publish twist message
twistpub.publish(twist)
```

Here the Twist messages are published as the topic `/cmd_vel_mux/input/navi'`. This is subscribed to by the command velocity multiplexer. It has three inputs of different priorities and and passes the highest priority messages to the `kobuki_node` that controls the base.

/cmd_vel_mux/input/navi lowest priority from a navigation node

/cmd_vel_mux/input/safety_controller middle priority from controller checking safety sensors

/cmd_vel_mux/input/teleop highest priority from remote operator

Moving the Turtlebot a specified distance is not straightforward since the control inputs are speeds and thus they need to be applied for a certain time[4]. Usually, simple kinematics are used but of course a large payload will affect the dynamics.

If a twist command has not been sent within 0.6 s, as a safety measure, the base controller sets the speeds to zero.

[4] Assuming there is no wheel slippage.

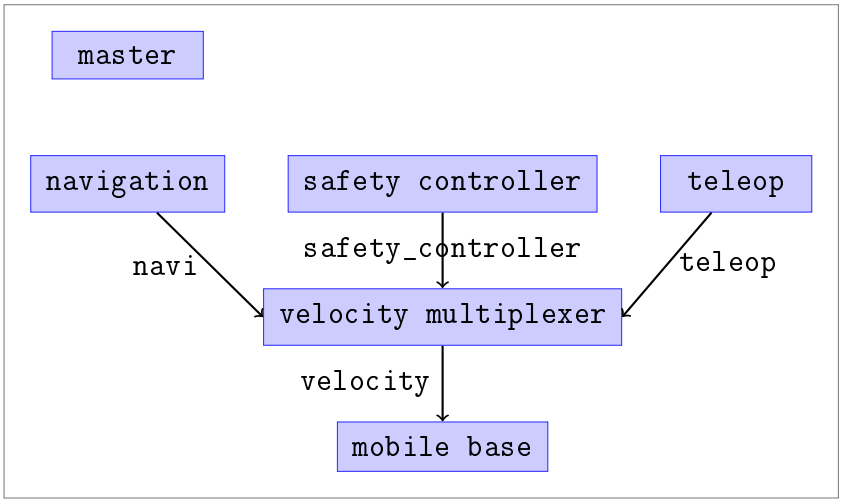


Figure 3.2: ROS computation graph for velocity multiplexer example.

4 Rospy threading

For each subscribed topic in a node, there is a thread to handle the callback. Publishers are synchronous/blocking, so the publishing occurs in the same thread.

5 ROS example applications

Here are some example applications I wrote[5]:

blinky1 This flashes the Turtlebot's LED. The code is shown in Listing 3.1. This example gets a parameter from the parameter server.

move1 This moves the Turtlebot's around in a large polygon. The code is shown in Listing 3.2.

move2 This is similar to move1 but it causes the Turtlebot to stop if it is lifted off the ground causing a wheel drop event. The code is shown in Listing 3.3.

wander1 This makes the Turtlebot wander around in a straight line until a bumper event causes it to back off and try a different direction. The code is shown in Listing 3.4.

whereami1 This uses the tf package to monitor transform frame messages. The pose of the Turtlebot in the map frame is displayed.

goto1 This uses the navigation action server to move the Turtlebot to a specific position in the map frame using path planning and local navigation.

[5] They can be downloaded using git from `https://eng-git.canterbury.ac.nz/mph/turtlebot-demo.git`.

```
#!/usr/bin/env python
from rospy import init_node, Publisher, get_param
from rospy import on_shutdown, is_shutdown, sleep, ROSInterruptException
from kobuki_msgs.msg import Led

def shutdown_callback():
    print "Shutting down..."


def blinker():
    init_node('blinky1')

    update_rate = get_param('~update_rate', 1.0)

    on_shutdown(shutdown_callback)

    print("Blinking...")

    pub = Publisher('/mobile_base/commands/led2', Led)
    msg = Led()

    colours = (msg.GREEN, msg.ORANGE, msg.RED)

    colour_index = 0
    while not is_shutdown():
        colour_index = (colour_index + 1) % len (colours)
        msg.value = colours[colour_index]

        pub.publish(msg)
        sleep(1.0 / update_rate)


if __name__ == '__main__':
    try:
        blinker()
    except ROSInterruptException:
        pass
```

Listing 3.1: Python ROS example: `blinky1.py`.

```
#!/usr/bin/env python
from rospy import Publisher, signal_shutdown, is_shutdown
from rospy import init_node, get_time, sleep, ROSInterruptException
from geometry_msgs.msg import Twist

action_duration = 0.25
move_speed = 0.15
turn_speed = 0.5


class Mover(object):

    def __init__(self):
        self.twistpub = Publisher('/cmd_vel_mux/input/navi', Twist,
            queue_size=1)

    def action(self, duration, angular_speed, linear_speed):
        twist = Twist()
        twist.linear.x = linear_speed
        twist.angular.z = angular_speed

        # Repeatedly send twist message for specified duration
        stopTime = get_time() + duration;
        while (get_time() < stopTime):
            self.twistpub.publish(twist)
            sleep(action_duration)

    def turn(self, duration, speed):
        self.action(duration, speed, 0.0)

    def move(self, duration, speed):
        self.action(duration, 0.0, speed)


def move1():
     init_node('move1')

     mover = Mover()

     while not is_shutdown():
         mover.turn(1.0, turn_speed)
         mover.move(2.0, move_speed)


if __name__ == '__main__':
     try:
         move1()
     except ROSInterruptException:
         pass
```

Listing 3.2: Python ROS example: `move1.py`.

```
#!/usr/bin/env python
from rospy import Publisher, signal_shutdown, is_shutdown, loginfo,
    Subscriber
from rospy import init_node, get_time, sleep, ROSInterruptException
from geometry_msgs.msg import Twist
from kobuki_msgs.msg import WheelDropEvent

action_duration = 0.25
move_speed = 0.15
turn_speed = 0.5

class Mover(object):

    def __init__(self):
        self.twistpub = Publisher('/cmd_vel_mux/input/navi', Twist,
            queue_size=1)

    def action(self, duration, angular_speed, linear_speed):

        twist = Twist()
        twist.linear.x = linear_speed
        twist.angular.z = angular_speed

        stopTime = get_time() + duration;
        while (get_time() < stopTime):
            self.twistpub.publish(twist)
            sleep(action_duration)

    def turn(self, duration, speed):
        self.action(duration, speed, 0.0)

    def move(self, duration, speed):
        self.action(duration, 0.0, speed)


def wheel_drop_callback(data):
    # Stop Robot's motion if picked up
    if data.state != WheelDropEvent.RAISED:
        loginfo("Shutting down due to wheel %d drop" % data.wheel)
        signal_shutdown("Wheel drop on wheel %d" % data.wheel)


def move2():
     init_node('move2')
     Subscriber("/mobile_base/events/wheel_drop", WheelDropEvent,
         wheel_drop_callback)

     mover = Mover()
     while not is_shutdown():
         mover.turn(1.0, turn_speed)
         mover.move(2.0, move_speed)

if __name__ == '__main__':
     try: move2()
     except ROSInterruptException: pass
```

Listing 3.3: Python ROS example: `move2.py`.

```
#!/usr/bin/env python
from rospy import Publisher, signal_shutdown, is_shutdown, loginfo, Subscriber
from rospy import init_node, get_time, sleep, ROSInterruptException
from geometry_msgs.msg import Twist
from kobuki_msgs.msg import WheelDropEvent, BumperEvent, CliffEvent, Sound

action_duration = 0.25
move_speed = 0.15
turn_speed = 0.5


class Robot(object):

    def __init__(self):
        self.twist_pub = Publisher('/cmd_vel_mux/input/navi', Twist, queue_size=1)
        self.sound_pub = Publisher('/mobile_base/commands/sound', Sound, queue_size=1)

        Subscriber('/mobile_base/events/wheel_drop', WheelDropEvent, self.wheel_drop_callback)
        Subscriber('/mobile_base/events/bumper', BumperEvent, self.bumper_callback)
        Subscriber('/mobile_base/events/cliff', CliffEvent, self.cliff_callback)

        self.bumper = 'off'
        self.cliff = 'off'

    def action(self, duration, angular_speed, linear_speed):
        twist = Twist()
        twist.linear.x = linear_speed
        twist.angular.z = angular_speed

        stopTime = get_time() + duration;
        while (get_time() < stopTime):
            self.twist_pub.publish(twist)
            sleep(action_duration)

    def turn(self, duration, speed):
        self.action(duration, speed, 0.0)

    def move(self, duration, speed):
        self.action(duration, 0.0, speed)

    def backup(self):
        self.move(action_duration, -move_speed)

    def cry(self):
         sound = Sound()
         sound.value = 6
         self.sound_pub.publish(sound)

    def bumper_callback(self, data):
         if data.state == BumperEvent.RELEASED:
              self.bumper = 'off'
         else:
              if data.bumper == BumperEvent.LEFT:
                   self.bumper = 'left'
              elif data.bumper == BumperEvent.CENTER:
                   self.bumper = 'centre'
              else:
                   self.bumper = 'right'

         loginfo('bumper state is %s' % self.bumper)

    def cliff_callback(self, data):
         if data.state == CliffEvent.FLOOR:
              self.cliff = 'off'
         else:
              if data.sensor == CliffEvent.LEFT:
                   self.cliff = 'left'
              elif data.sensor == CliffEvent.CENTER:
                   self.cliff = 'centre'
              else:
                   self.cliff = 'right'

         loginfo('cliff state is %s' % self.cliff)

    def wheel_drop_callback(self, data):
         # Stop Robot's motion if picked up
         if data.state != WheelDropEvent.RAISED:
              self.cry()
              loginfo('Shutting down due to wheel drop on wheel %d' % data.wheel)
              signal_shutdown('Wheel drop on wheel %d' % data.wheel)
```

```
from rospy import init_node, is_shutdown, loginfo
from geometry_msgs.msg import Point, Quaternion
from tf.transformations import euler_from_quaternion
from tf import TransformListener, LookupException, ConnectivityException, ExtrapolationException
from math import degrees

class TurtlebotPose(object):

    def __init__(self, position=(0, 0, 0), orientation=(0, 0, 1, 0)):
        self.position = Point(*position)
        self.orientation = Quaternion(*orientation)

    @property
    def x(self):
        return self.position.x

    @property
    def y(self):
        return self.position.y

    @property
    def heading(self):
        roll, pitch, yaw = euler_from_quaternion([self.orientation.x,
                                                  self.orientation.y,
                                                  self.orientation.z,
                                                  self.orientation.w])
        return yaw

class TurtlebotTracker(object):
    def __init__(self, *args):
        init_node('whereami')
        self.tf = TransformListener()
        self.pose_time = None
        self.pose = TurtlebotPose()

    def get_pose(self):
        """Determine pose of turtlebot (base_link frame) with respect to map frame."""
        if not self.tf.frameExists("/base_link") or not self.tf.frameExists("/map"):
            return
        try:
            t = self.tf.getLatestCommonTime("/base_link", "/map")
            position, orientation = self.tf.lookupTransform("/base_link", "/map", t)
            self.pose_time = t
            self.pose = TurtlebotPose(position, orientation)
        # Exceptions are raised until enough transform messages arrive
        except (Exception, LookupException, ConnectivityException, ExtrapolationException):
            return

tracker = TurtlebotTracker()

last_pose_time = 0
while not is_shutdown():

    tracker.get_pose()
    if tracker.pose_time is None:
        continue

    pose_time = tracker.pose_time.secs
    if last_pose_time != pose_time:
        pose = tracker.pose
        loginfo("Turtlebot pose: position=(%.3f, %.3f) heading=%1.f" %
                (pose.x, pose.y, degrees(pose.heading)))
        last_pose_time = pose_time
```

Listing 3.5: Python ROS example: `whereami1.py`.

```python
#!/usr/bin/env python
from rospy import init_node, loginfo
from actionlib import SimpleActionClient
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from geometry_msgs.msg import Quaternion
from tf.transformations import quaternion_from_euler
from math import radians

# Based on code written by Sam Pfeiffer
# https://github.com/reem-utils/rockin_snippets/blob/master/scripts/navigation.py

class Goto(object):
    """This is a wrapper class for the navigation action server (NAS).
    It creates a goal and asks the NAS to move the Turtlebot to it."""

    def __init__(self):
        # Connect to the navigation action server
        self.nav_as = SimpleActionClient('/move_base', MoveBaseAction)
        self.nav_as.wait_for_server()
        loginfo("Waiting to connect to navigation action server...")

    def _create_goal(self, x, y, heading):
        """Create a MoveBaseGoal with x, y position and heading rotation (in degrees)."""
        mb_goal = MoveBaseGoal()
        mb_goal.target_pose.header.frame_id = '/map'
        mb_goal.target_pose.pose.position.x = x
        mb_goal.target_pose.pose.position.y = y
        mb_goal.target_pose.pose.position.z = 0.0

        angle = radians(heading)
        quat = quaternion_from_euler(0.0, 0.0, angle)
        mb_goal.target_pose.pose.orientation = Quaternion(*quat.tolist())

        return mb_goal

    def move_goal(self, x, y, heading):
        """Try to move base to x, y position with heading angle."""

        nav_goal = self._create_goal(x, y, heading)

        loginfo("Sending goal to navigation action server...")
        self.nav_as.send_goal(nav_goal)

    def wait(self):
        """Wait for navigation server to finish."""

        loginfo("Waiting for navigation action server to finish...")
        self.nav_as.wait_for_result()

        result = self.nav_as.get_state()

        # Possible states PENDING, ACTIVE, RECALLED, REJECTED,
        # PREEMPTED, ABORTED, SUCCEEDED, LOST.
        loginfo("Navigation action server finished with status %s" %
                result)
        return result


def move_goal(x, y, heading):

    goto = Goto()
    goto.move_goal(x, y, heading)

    return goto.wait()


if __name__ == '__main__':
    init_node("goto1")

    from sys import argv
    argc = len(argv)

    x = float(argv[1]) if argc > 1 else 0.0
    y = float(argv[2]) if argc > 2 else 0.0
    heading = float(argv[3]) if argc > 3 else 0.0
    move_goal(x, y, heading)
```

Listing 3.6: Python ROS example: `goto1.py`.

6 *Exercises*

1. How is the Turtlebot heading determined?
2. What sensor fusion is performed in the Turtlebot base?
3. What is the point of a velocity multiplexer?
4. What does a Twist message specify?
5. How does a ROS program handle sensor events?
6. Why do the speed messages need to be repeated?
7. How do you move the Turtlebot a specified distance?
8. How do you orient the Turtlebot to a specified orientation?

4
Sensor fusion

Consider a robot with two range sensors: one an accurate laser range-finder; the other a less accurate ultrasonic range-finder. So is can a more accurate estimate be achieved combining the two measurements? Surprisingly, the answer is yes. To understand how this improves the variance of the estimate, it is necessary to consider some statistics and the method of weighted least squares.

Sensor fusion is commonly used in mobile robotics to combine (fuse) measurements from different sensors. The resulting estimate has a lower variance than the individual measurements. An example, is the fusing of measurements from an inertial measurement unit[1] (IMU) with a global positioning system (GPS).

[1] These contain accelerometers, gyroscopes, and magnetometers.

1 *Weighted averaging*

Let's assume that the range measured by a laser range finder is described by a random variable R_1 and the range measured by a ultrasonic range finder is described by a random variable R_2. Let's also assume that the errors are uncorrelated and Gaussian distributed[2] with variances σ_1^2 and σ_2^2.

[2] This is a poor assumption with some sensors.

The weighted least squares estimate of the range, $\hat{R}$ is found from

$$\hat{R} = \frac{w_1 R_1 + w_2 R_2}{w_1 + w_2}, \tag{4.1}$$

where the weights are chosen inversely proportional to the variances[3]

[3] Provided the measurements are uncorrelated.

$$w_i = \frac{1}{\sigma_i^2}. \tag{4.2}$$

Thus the weighted least means square estimate is

$$\hat{R} = \frac{\frac{1}{\sigma_1^2}R_1 + \frac{1}{\sigma_2^2}R_2}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}, \tag{4.3}$$

with a variance[4]

$$\sigma_{\hat{R}}^2 = \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}, \tag{4.4}$$

$$= \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}. \tag{4.5}$$

[4] This result is analogous to two resistors in parallel or two springs in series.

Note that this is smaller than σ_1^2 and σ_2^2.

1.1 Examples

For example, let $\sigma_1^2 = 0.25$ and $\sigma_2^2 = 1$, so $w_1 = 4$ and $w_2 = 1$ and thus

$$\hat{R} = \frac{4R_1 + R_2}{5}, \tag{4.6}$$

$$= 0.8R_1 + 0.2R_2, \tag{4.7}$$

with a variance

$$\sigma_{\hat{R}}^2 = \frac{0.25 \times 1}{1 + 0.25}, \tag{4.8}$$

$$= 0.2. \tag{4.9}$$

Again note, this is smaller than both σ_1^2 and σ_2^2.

σ_1	σ_2	w_1	w_2	$\sigma_{\hat{R}}$
1.00	1.00	0.50	0.50	0.71
1.00	2.00	0.80	0.20	0.89
1.00	3.00	0.90	0.10	0.95
1.00	4.00	0.94	0.06	0.97
1.00	5.00	0.96	0.04	0.98

Table 4.1: Optimal weights for the weighted sum of two random variables. Note, the resulting standard deviation, $\sigma_{\hat{R}}$, is smaller than the standard deviations of the two random variables but there is diminishing returns.

1.2 General result

In general, if there are N uncorrelated range sensor measurements, the weighted least squares estimate of the range, $\hat{R}$ is found from

$$\hat{R} = \frac{\sum_{i=1}^{N} w_i R_i}{\sum_{i=1}^{N} w_i}, \tag{4.10}$$

$$= \frac{\sum_{i=1}^{N} \frac{1}{\sigma_i^2} R_i}{\sum_{i=1}^{N} \frac{1}{\sigma_i^2}}, \tag{4.11}$$

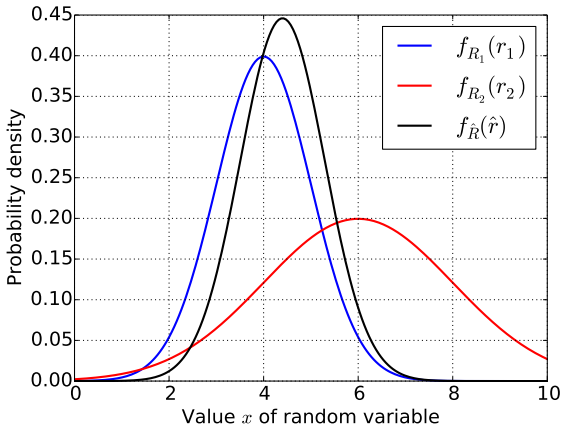


Figure 4.1: Fusing probability densities of two measurements. R_1 has a mean $\mu_1 = 4$ and variance $\sigma_1^2 = 1.0$; R_2 has a mean $\mu_2 = 6$ and variance $\sigma_2^2 = 2.0$. Note, how the resultant pdf, $p(\hat{R})$, is higher and narrower. The mean of the result is closer to the mean of the measurement with the smaller variance.

with a variance

$$\sigma_{\hat{R}}^2 = \frac{1}{\sum_{i=1}^{N} \frac{1}{\sigma_i^2}}. \tag{4.12}$$

2 *Gaussian pdf*

The noise distribution for most sensors is Gaussian[5]. This distribution for a random variable X can be parameterised by two parameters, the mean, μ_X, and variance, σ_X^2, (see Figure 4.2)

$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma_X} \exp\left(-\frac{1}{2}\left(\frac{x-\mu_X}{\sigma_X}\right)^2\right). \tag{4.13}$$

[5] If it is not, averaging often produces a Gaussian distribution due to the central limit theorem.

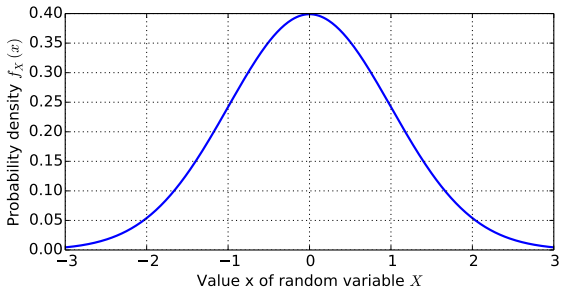


Figure 4.2: Gaussian probability distribution, zero mean and unit variance: $\mu_X = 0$, $\sigma_X^2 = 1$.

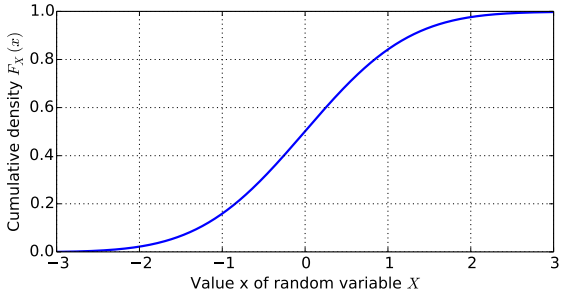


Figure 4.3: Gaussian cumulative probability distribution, zero mean and unit variance: $\mu_X = 0$, $\sigma_X^2 = 1$.

3 *Multivariate Gaussian pdf*

The N-dimensional Gaussian probability distribution for a real random vector $\mathbf{X} = (X_1, X_2, \cdots X_N)^{\mathrm{T}}$, is[6]

$$f_{\mathbf{X}}(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^N \det(\mathbf{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}} \mathbf{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})\right), \tag{4.14}$$

where $\boldsymbol{\mu} = \mathrm{E}\{\mathbf{X}\}$ is the mean of $\mathbf{X}$ and

$$\mathbf{\Sigma} = \mathrm{E}\left\{(\mathbf{X}-\boldsymbol{\mu})(\mathbf{X}-\boldsymbol{\mu})^{\mathrm{T}}\right\} \tag{4.15}$$

is the *covariance matrix* of $\mathbf{X}$. Note, the pdf is parameterised by a vector of mean values and a covariance matrix. No higher order statistics are required.

For the bivariate Gaussian distribution, $\mathbf{X} = (X, Y)^{\mathrm{T}}$, $\boldsymbol{\mu} = (\mu_X, \mu_Y)^{\mathrm{T}}$, and the covariance matrix is

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{bmatrix}, \tag{4.16}$$

where ρ is the correlation coefficient between X and Y. When $\rho = 0$ then X and Y are uncorrelated[7]. When $\rho = 1$ then X and Y are perfectly correlated[8].

[6] Note, in the 1-D case, this collapses to $f_X(x) = \frac{1}{\sqrt{2\pi\sigma_X^2}} \exp\left(-0.5(x-\mu_X)^2/\sigma_X^2\right)$.

[7] $f_{X,Y}(x,y) = f_X(x) f_Y(y)$.

[8] $f_{X,Y}(x,y) = f_X(x)\delta(y-x)$.

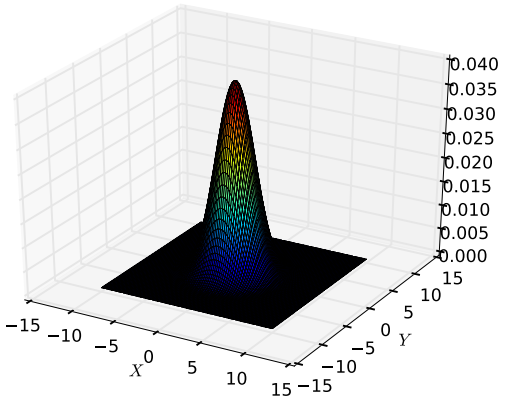


Figure 4.4: 2-D Gaussian pdf where $\mu_X = 0$, $\mu_Y = 0$, $\sigma_X = 2$, $\sigma_2 = 2$, $\rho = 0$ (uncorrelated).

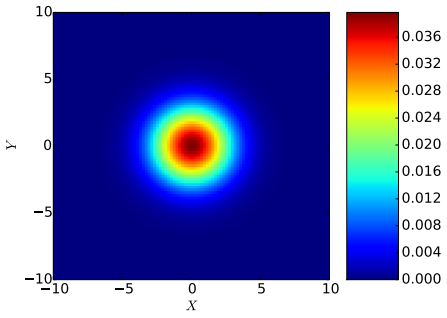


Figure 4.5: 2-D Gaussian pdf where $\mu_X = 0$, $\mu_Y = 0$, $\sigma_X = 2$, $\sigma_2 = 2$, $\rho = 0$ (uncorrelated).

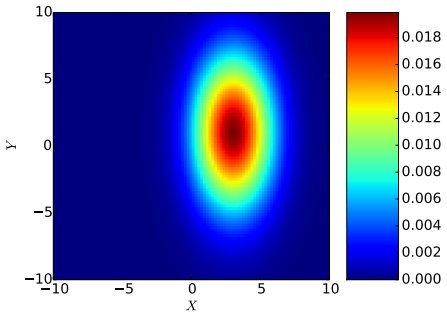


Figure 4.6: 2-D Gaussian pdf where $\mu_X = 3$, $\mu_Y = 1$, $\sigma_X = 2$, $\sigma_Y = 4$, $\rho = 0$ (uncorrelated).

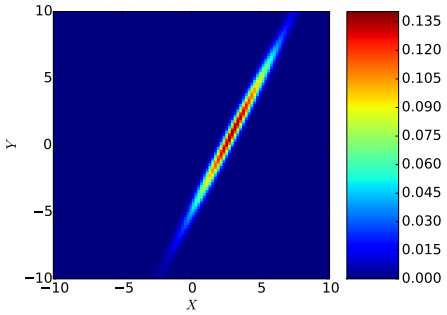


Figure 4.7: 2-D Gaussian function where $\mu_X = 3$, $\mu_Y = 1$, $\sigma_X = 2$, $\sigma_Y = 4$, $\rho = 0.9$ (highly correlated).

For the trivariate Gaussian distribution, $\mathbf{X} = (X_1, X_2, X_3)^{\mathrm{T}}$, $\boldsymbol{\mu} = (\mu_{X_1}, \mu_{X_2}, \mu_{X_3})^{\mathrm{T}}$, and the covariance matrix is

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & \rho_{12}\sigma_{X_1}\sigma_{X_2} & \rho_{13}\sigma_{X_1}\sigma_{X_3} \\ \rho_{12}\sigma_{X_1}\sigma_{X_3} & \sigma_{X_2}^2 & \rho_{23}\sigma_{X_2}\sigma_{X_3} \\ \rho_{13}\sigma_{X_2}\sigma_{X_3} & \rho_{23}\sigma_{X_2}\sigma_{X_3} & \sigma_{X_3}^2 \end{bmatrix}. \tag{4.17}$$

The distribution can be described by 9 parameters; three means, three variances, and three correlations.

4 *Probability revision*

Random variables can be discrete or continuous. In the following continuous random variables are assumed. For discrete random variables it is necessary to replace the integrals with discrete summations.

The probability that a continuous random variable X is less than some value x is

$$\mathrm{P}(X \leq x) = F_X(x), \tag{4.18}$$

where $F_X(x)$ is the cumulative probability distribution of X. In general, the probability that X lies in the semi-closed interval $(a, b]$ is

$$\begin{aligned} \mathrm{P}(a < X \leq b) &= \int_a^b f_X(x)\mathrm{d}x, & (4.19) \\ &= F_X(b) - F_X(a), & (4.20) \end{aligned}$$

where $f_X(x)$ is the probability density function[9] of X, related to the cumulative distribution by

$$F_X(x) = \int_{-\infty}^{x} f_X(t)\,\mathrm{d}t. \tag{4.21}$$

The mean or expected value of X can be determined using

$$\mu_X = \mathrm{E}[X] = \int_{-\infty}^{\infty} x f(x)\,\mathrm{d}x. \tag{4.22}$$

The expected value of a function of X, $g(X)$ is given by

$$\mathrm{E}[g(X)] = \int_{-\infty}^{\infty} g(x) f(x)\,\mathrm{d}x. \tag{4.23}$$

The variance or expected value of X can be determined using

$$\sigma_X^2 = \mathrm{Var}\{X\} = \int_{-\infty}^{\infty} (x-\mu_X)^2 f(x)\,\mathrm{d}x, \tag{4.24}$$

$$= \int_{-\infty}^{\infty} x^2 f(x)\,\mathrm{d}x - \mu_X^2. \tag{4.25}$$

Let a random variable Z be formed from the addition of random variables X and Y,

$$Z = X + Y. \tag{4.26}$$

The resultant mean is the sum of the means,

$$\mu_Z = \mu_X + \mu_Y, \tag{4.27}$$

but the resultant variance depends on the correlation $\rho = \rho_{XY}$,

$$\sigma_Z^2 = \sigma_X^2 + \sigma_Y^2 + 2\rho\sigma_X\sigma_Y. \tag{4.28}$$

When X and Y are uncorrelated, $\rho = 0$ and so the variances sum

$$\sigma_Z^2 = \sigma_X^2 + \sigma_Y^2. \tag{4.29}$$

When X and Y are fully correlated, $\rho = 1$ and so the standard deviations sum

$$\sigma_Z = \sigma_X + \sigma_Y. \tag{4.30}$$

Let a random variable Y be formed from the scaling of a random variables X,

$$Y = aX. \tag{4.31}$$

The resultant mean is scaled by the same factor a,

$$\mu_Y = a\mu_X, \tag{4.32}$$

[9] The notation $p_X(x)$ is also common and so is $p(x)$, although the latter is ambiguous. The notation is further blurred when using a lowercase symbol for a random variable.

and the resultant variance is scaled by a^2,

$$\sigma_Y^2 = a^2\sigma_X^2. \tag{4.33}$$

Let a random variable Y be a function of a random variable X,

$$Y = g(X). \tag{4.34}$$

The cumulative density functions are related by

$$F_Y(y) = F_X\left(g^{-1}(y)\right), \tag{4.35}$$

and thus the probability density function for Y is

$$f_Y(y) = \frac{\mathrm{d}F_Y(y)}{\mathrm{d}y} = \frac{\mathrm{d}F_X\left(g^{-1}(y)\right)}{\mathrm{d}y}. \tag{4.36}$$

5 *Weighted least squares derivation*

Let's consider the case of a weighted sum of two random variables so the resulting value is

$$Y = \frac{w_1X_1 + w_2X_2}{w_1 + w_2}, \tag{4.37}$$

or in matrix form

$$\begin{aligned} Y &= \begin{bmatrix} \frac{w_1}{w_1+w_2} & \frac{w_2}{w_1+w_2} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}, & (4.38)\\ &= \mathbf{w}^T\mathbf{X}, & (4.39) \end{aligned}$$

where $\mathbf{X} = (X_1, X_2)^T$ and $\mathbf{w} = \left(\frac{w_1}{w_1+w_2}, \frac{w_2}{w_1+w_2}\right)$. Let's assume that the errors of $\mathbf{X}$ are uncorrelated so that

$$\mathbf{\Sigma}_X = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}, \tag{4.40}$$

and thus using the propagation of uncertainty

$$\begin{aligned} \sigma_Y^2 &= \mathbf{w}^T\mathbf{\Sigma}_X\mathbf{w}, & (4.41)\\ &= \begin{bmatrix} \frac{w_1}{w_1+w_2} & \frac{w_2}{w_1+w_2} \end{bmatrix} \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix} \begin{bmatrix} \frac{w_1}{w_1+w_2} \\ \frac{w_2}{w_1+w_2} \end{bmatrix}, & (4.42)\\ &= \frac{w_1^2\sigma_1^2 + w_2^2\sigma_2^2}{(w_1 + w_2)^2}. & (4.43) \end{aligned}$$

Differentiating[10] σ_Y^2 with respect to w_1 and solving for

[10] checkme

w_1 where the derivative is zero, yields

$$w_1 = \frac{\sigma_2^2}{\sigma_1^2} w_2. \tag{4.44}$$

The optimal choice of weights are $w_i = 1/\sigma_i^2$, i.e., the inverse of the variance,

$$Y = \frac{\frac{1}{\sigma_1^2} X_1 + \frac{1}{\sigma_2^2} X_2}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}, \tag{4.45}$$

$$= \frac{\sigma_2^2 X_1 + \sigma_1^2 X_2}{\sigma_1^2 + \sigma_2^2}. \tag{4.46}$$

Using these optimal weights, the resulting variance is

$$\sigma_Y^2 = \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}, \tag{4.47}$$

$$= \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}. \tag{4.48}$$

6 *Summary*

- Sensor fusion combines measurements from disparate sources to improve the estimate (for example, GPS and IMU).
- Sensor noise is commonly modelled with a Gaussian pdf; this only requires a mean and variance to parameterise.
- Multivariate Gaussian pdfs are parameterised by a vector of means and a covariance matrix.
- A covariance matrix specifies the variance of each variable and the correlations between them.

7 *Exercises*

1. An ultrasonic sensor with a variance of 0.01 measures the range to a wall to be 1.2 m. A lidar with a variance 0.01 measures the range to the wall to be 1.1 m.
   (a) If the measurements are to be fused using weighted least squares, what should the weightings be?
   (b) If the measurements are to be fused using weighted least squares, what is the resulting variance?

2. An ultrasonic sensor with a variance of 0.04 measures the range to a wall to be 1.2 m. A lidar with a variance 0.01 measures the range to the wall to be 1.1 m.

   (a) If the measurements are to be fused using weighted least squares, what should the weightings be?

   (b) If the measurements are to be fused using weighted least squares, what is the resulting variance?

5
Kalman filter introduction

The lecture introduces the basic concept of a Kalman filter[1]. Kalman filters are a class of Gaussian Bayes filters, considered in more detail in Chapter 10. They are used for state estimation and are useful for sensor fusion.

[1] These were invented around 1960 and were used on the Apollo lunar lander.

1 1-D state estimation problem

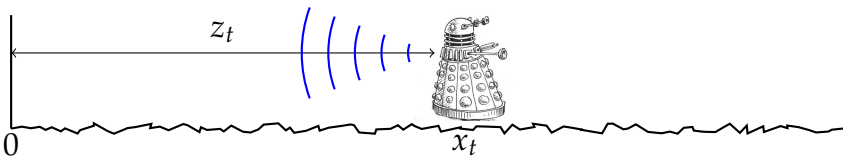


Figure 5.1: Dalek position estimation problem.

Let's consider a Dalek moving in a straight line over a rough surface with a nominally constant speed v (see Figure 5.1). We can model the position of the Dalek at a time-step t by x_t where

$$x_t = x_{t-1} + v\Delta t. \tag{5.1}$$

Here Δt is the interval between time-steps. In practice, there are disturbances due to friction, wind gusts, wheel slippage, etc. We can model the position error due to these disturbances[2] with a zero-mean Gaussian random variable $\tilde{x}$ and thus we can convert the deterministic model (5.1) into a probabilistic model[3],

$$x_t = x_{t-1} + v\Delta t + \tilde{x}. \tag{5.2}$$

To help the Dalek find out where it is it is equipped with a distance sensor. At each time-step it provides a reading z_t, where

$$z_t = x_t + \tilde{z}. \tag{5.3}$$

Here $\tilde{z}$ is a zero mean Gaussian random variable that models the sensor noise.

[2] We call this process noise.

[3] x_t is now a random variable but I'll use lower case to simplify the notation.

Note, we have two equations that we can use to determine the Dalek's position. So by combining the measurements with the model a better estimate of the Dalek's position can be estimated. To see this, let's denote the current estimate of the Dalek's positition by $\hat{x}_t$. Using (5.2), the predicted position, $\check{x}_t$, is

$$\check{x}_t = \hat{x}_{t-1} + v\Delta t. \tag{5.4}$$

Now if we consider the measurement, z_t, we can infer the Dalek's position, $\acute{x}_t$, where

$$\acute{x}_t = z_t. \tag{5.5}$$

Finally, we can combine $\check{x}_t$ and $\acute{x}_t$ using a weighted average to get a new estimate $\hat{x}_t$. This is the basis of a Kalman filter.

1.1 *Weighted average*

A Kalman filter is a state estimator; it uses the measurement, z_t, in conjunction with the update model (5.2) to estimate the state.

The Kalman filter estimates the state using a weighted sum of the predicted and inferred states,

$$\hat{x}_t = \frac{\check{x}_t w_1 + \acute{x}_t w_2}{w_1 + w_2}, \tag{5.6}$$

where w_1 and w_2 are weighting factors. This is shown diagrammatically in Figure 5.2.

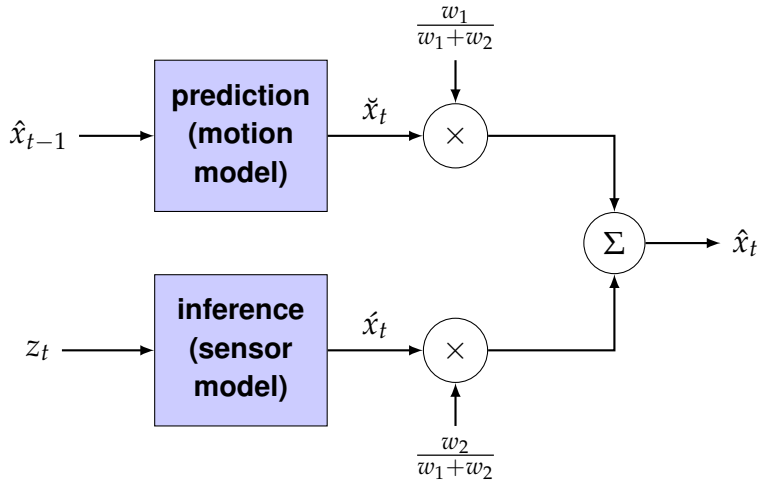


Figure 5.2: Block diagram of a Kalman filter as a weighted average.

Equation (5.6) can be cunningly expressed as (see Figure 5.3)

$$\hat{x}_t = \check{x}_t + \frac{w_2}{w_1 + w_2}\left(\acute{x}_t - \check{x}_t\right), \tag{5.7}$$

or

$$\hat{x}_t = \breve{x}_t + K_t \left(z_t - \breve{x}_t \right), \tag{5.8}$$

where the factor in parentheses is called the *innovation* and K_t is the Kalman gain given by

$$K_t = \frac{w_2}{w_1 + w_2}. \tag{5.9}$$

Note, these weights are time varying.

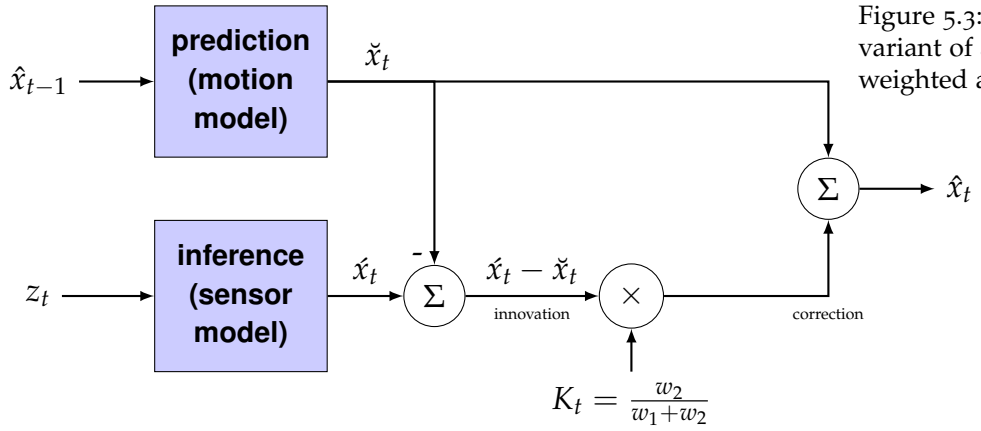


Figure 5.3: Block diagram variant of a Kalman filter as a weighted average.

1.2 *Kalman gain*

When combining noisy measurements, the best approach is to weight them inversely by their variances. Assuming the errors are uncorrelated, the variances of the two estimates (from (5.2) and (5.3)) are

$$\sigma^2_{\breve{x}_t} = \sigma^2_{\hat{x}_{t-1}} + \sigma^2_{\tilde{x}}, \tag{5.10}$$

$$\sigma^2_{\acute{x}_t} = \sigma^2_{\tilde{z}}. \tag{5.11}$$

and so the optimal Kalman gain is

$$K_t = \frac{\frac{1}{\sigma^2_{\breve{x}_t}}}{\frac{1}{\sigma^2_{\breve{x}_t}} + \frac{1}{\sigma^2_{\acute{x}_t}}}, \tag{5.12}$$

$$= \frac{\sigma^2_{\breve{x}_t}}{\sigma^2_{\acute{x}_t} + \sigma^2_{\breve{x}_t}}. \tag{5.13}$$

The Kalman filter dynamically adjusts the weighting depending on the variance of the state estimate, the variance of the process noise, and the variance of the measurement. The updated estimate of the state variance is

$$\sigma^2_{\hat{x}_t} = \left(1 - K_t\right) \sigma^2_{\breve{x}_t}. \tag{5.14}$$

2 Kalman filter implementation

Most Kalman filters are implemented with two steps: a predict step, followed by an update step that uses the measurement(s).

2.1 Predict step

The predicted (*a priori*) state estimate is

$$\check{x}_t = \hat{x}_{t-1} + v\Delta t, \tag{5.15}$$

and the predicted (*a priori*) state variance estimate is

$$\sigma^2_{\check{x}_t} = \sigma^2_{\hat{x}_{t-1}} + \sigma^2_{\tilde{x}}. \tag{5.16}$$

2.2 Update step

The updated (*a posteriori*) state estimate is

$$\hat{x}_t = \check{x}_t + K_t \left(z_t - \check{x}\right), \tag{5.17}$$

and the updated (*a posteriori*) state variance estimate is

$$\sigma^2_{\hat{x}_t} = \left(1 - K_t\right) \sigma^2_{\check{x}_t}, \tag{5.18}$$

where K_t is the optimal Kalman gain

$$K_t = \frac{\sigma^2_{\check{x}_t}}{\sigma^2_{\check{x}_t} + \sigma^2_{\tilde{z}}}. \tag{5.19}$$

2.3 Comments

The Kalman filter is recursive and only needs to maintain the current state estimate, $\hat{x}_t$ and its variance $\sigma^2_{\hat{x}_t}$. All the previous measurements (and control actions) do not need to be stored.

If the process and measurement noise variances are constant then the Kalman gain will also converge to a constant called the *steady state Kalman gain*.

3 Weighted least squares revisited

From Section 5, the optimal weights for weighted least squares is the reciprocal of the variances. Thus for the fusion of two random variables (X_1 and X_2),

$$Y = \frac{\frac{1}{\sigma_1^2}X_1 + \frac{1}{\sigma_2^2}X_2}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}, \tag{5.20}$$

$$= \frac{\sigma_2^2 X_1 + \sigma_1^2 X_2}{\sigma_1^2 + \sigma_2^2}. \tag{5.21}$$

This result can be expressed as

$$Y = X_1 + \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}(X_2 - X_1), \tag{5.22}$$

$$= X_1 + K(X_2 - X_1), \tag{5.23}$$

where

$$K = \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}. \tag{5.24}$$

This formulation is the basis of a *Kalman filter* where K is called the Kalman gain.

4 Averaging filter

Let's say we have a sensor providing measurements x_t with a variance $\sigma_{x_t}^2$, at time step t, and we want to average the measurements on-the-fly to obtain a more accurate estimate, $\hat{x}_t$.

A naïve approach[4] is to average all the previous measurements,

$$\hat{x}_t = \frac{1}{t+1}\sum_{i=0}^{t} x_i. \tag{5.25}$$

[4] This is the optimum approach if the measurement variance is unchanging.

To avoid the storage of all the previous measurements, the averaging can be performed recursively,

$$\hat{x}_t = \frac{x_t}{t+1} + \frac{t}{t+1}\hat{x}_{t-1}, \tag{5.26}$$

$$= \hat{x}_{t-1} + \frac{1}{t+1}(x_t - \hat{x}_{t-1}). \tag{5.27}$$

Note, this just requires storage of the time step index t and the previous estimate of the average $\hat{x}_{t-1}$.

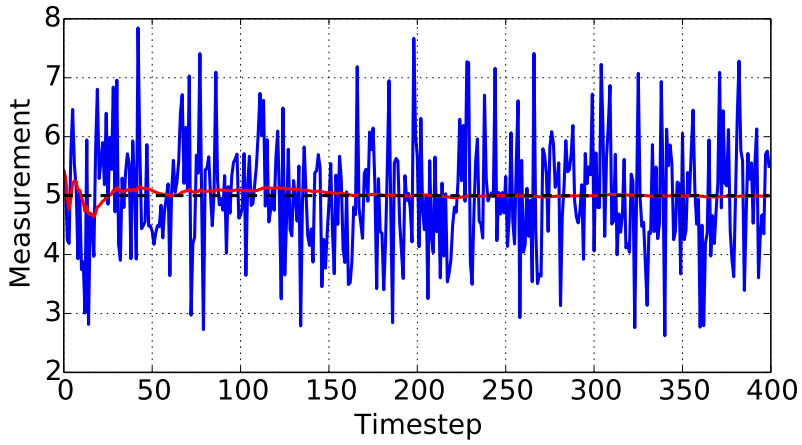


Figure 5.4: Demonstration of averaging filter. The measurements are of a constant signal with Gaussian noise. The dashed curve is the true signal.

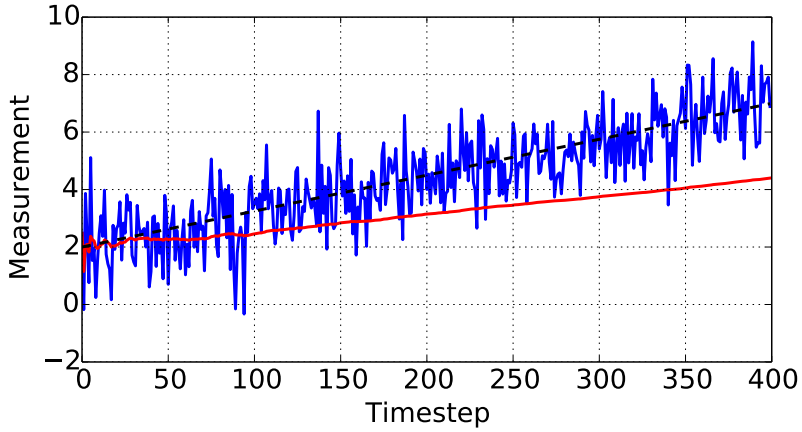


Figure 5.5: Demonstration of averaging filter. The measurements are of a linearly varying signal with Gaussian noise. The dashed curve is the true signal. The problem here is that the signal is nonstationary; its mean is changing with time. We need a better system model to handle the additional parameter.

5 *Averaging with a Kalman filter*

When the variance of each measurement varies, we do not want to weight the measurements equally. We can achieve this by changing the weighting of the second term in (5.27),

$$\hat{x}_t = \hat{x}_{t-1} + K_t \left(x_t - \hat{x}_{t-1} \right). \tag{5.28}$$

The weighting K_t is called the *Kalman gain*. The Kalman gain[5] is

$$K_t = \frac{\sigma^2_{\hat{x}_{t-1}}}{\sigma^2_{\hat{x}_{t-1}} + \sigma^2_{x_t}}, \tag{5.29}$$

where $\sigma^2_{\hat{x}_{t-1}}$ is the previous estimate of the averaged signal variance given by

$$\sigma^2_{\hat{x}_t} = \frac{\sigma^2_{\hat{x}_{t-1}} \sigma^2_{x_t}}{\sigma^2_{\hat{x}_{t-1}} + \sigma^2_{x_t}}. \tag{5.30}$$

The Kalman gain determines how much weighting to give the *innovation* (the difference between the new measurement and the current estimate).

The cunning aspect of the Kalman filter is that it does not need to store the entire measurement history, x_i for $i = 1 \cdots t$, but just the previous estimate, $\hat{x}_{t-1}$, and the previous estimate of the variance, $\sigma^2_{\hat{x}_{t-1}}$. The Kalman filter computes the new fused estimate using (5.28) and the new estimate of the fused variance[6] using (5.30).

Note, when the variance of the input measurements is constant, $\sigma^2_{x_t} = \sigma^2_x$, then the Kalman filter simplifies to an averaging filter with

$$K_t = \frac{1}{t+1}. \tag{5.31}$$

This results in

$$\hat{x}_t = \frac{1}{t} \sum_{i=1}^{t} x_i. \tag{5.32}$$

The Kalman filter just needs an estimate of the measurement variance at each time step. Given this estimate, it minimises the variance of the fused estimate. However, if a measurement has a bias, this Kalman filter cannot compensate for it since the underlying model is incorrect.

[5] Yes the notation is confusing with all these hats and subscripts. With reference to (5.23), $y = \hat{x}_t$, $X_1 = \hat{x}_{t-1}$, $X_2 = x_t$.

[6] Initially, $\sigma^2_{\hat{x}_1} = \infty$.

6 Summary

- The Kalman filter is an on-the-fly (online) state estimation algorithm.
- The Kalman gain dynamically weights the contribution of each new measurement.
- The Kalman filter stores only the estimated mean state and the estimated variance[7].
- The Kalman filter is only applicable for linear systems.
- The Kalman filter requires the noise to be additive and Gaussian.

[7] Or covariance matrix for multidimensional problems.

7 Exercises

1. What should the initial Kalman gain be?
2. If the Kalman gain is zero, does a Kalman filter trust the model or the measurement?
3. What does a Kalman gain of unity imply?
4. Consider a system with a single state variable and a single measurement sensor. If the measurement variance is constant, sketch how the Kalman gain varies with increasing time step.
5. If you require a robot to respond quickly to new measurements, should the Kalman gain be low or high?

```
# Kalman filter demonstration program to estimate
# position of a Dalek moving at constant speed.
# M.P. Hayes UCECE 2015
from numpy.random import randn

# Initial position
x = 0
# Speed
v = 0.2

# Time step
dt = 0.1

# Position standard deviation
sigma_x = 0.02 * dt
# Measurement standard deviation
sigma_z = 0.1

# Position and measurement variances
sigmasq_x = sigma_x ** 2
sigmasq_z = sigma_z ** 2

# Start with a poor estimate of Dalek's position
x_est = 10
sigmasq_x_est = 10 ** 2

while True:

    # Update Dalek position
    x = x + v * dt + randn(1) * sigmasq_x

    # Predict position from model
    x_pred = x_est + v * dt

    # Measure range
    z = x + randn(1) * sigma_z

    # Infer position from measurement
    x_infer = z

    # Calculate Kalman gain
    sigmasq_x_pred = sigmasq_x_est + sigmasq_x
    K = sigmasq_x_pred / (sigmasq_z + sigmasq_x_pred)

    # Update position estimate and its variance
    x_est = x_pred + K * (x_infer - x_pred)
    sigmasq_x_est = (1 - K) * sigmasq_x_pred
```

Listing 5.1: Python implementation of a Kalman filter for the Dalek position estimation problem.

6
State-space Kalman filter

Kalman filters can be applied to estimate multiple state variables simultaneously for a linear system.

1 State-space model

The state-space representation of a discrete time-invariant[1] linear system is

$$\begin{aligned} \mathbf{x}_t &= \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t, \quad \text{(state equation)} \\ \mathbf{z}_t &= \mathbf{C}\mathbf{x}_t + \mathbf{D}\mathbf{u}_t. \quad \text{(output equation)} \end{aligned} \tag{6.1}$$

[1] A time variant system can be modelled by replacing $\mathbf{A}$ with $\mathbf{A}_t$, etc. Some authors use $\mathbf{F}$, $\mathbf{H}$, etc., to distinguish the discrete-time and continuous time cases.

Here $\mathbf{x}_t$ denotes the system state at time step t, $\mathbf{u}$ is the control input, $\mathbf{z}_t$ is the measured output, $\mathbf{A}$ is the state transition matrix, $\mathbf{B}$ is the input matrix, $\mathbf{C}$ is the output matrix, and $\mathbf{D}$ is the feed-through matrix (usually zero).

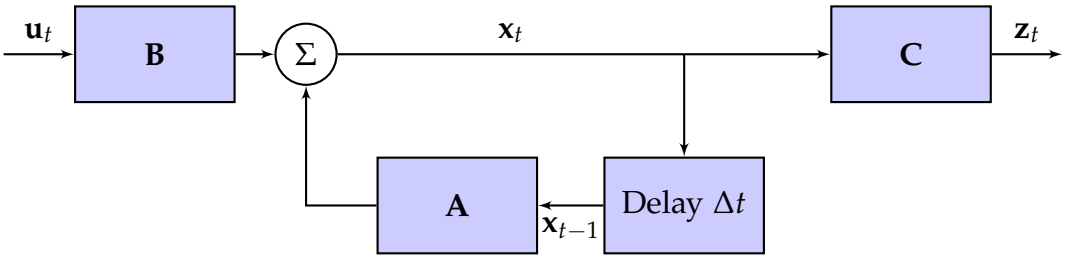


Figure 6.1: Discrete linear state-space system block diagram assuming $\mathbf{D} = 0$.

Let's consider a mobile robot system where:

- The state, $\mathbf{x}$, is the base pose[2] (x, y, θ) and rate of change of pose[3] $(v_x, v_y, \omega) = (\dot{x}, \dot{y}, \dot{\theta})$.
- The control action, $\mathbf{u}$, is the linear and angular speeds commanded to the base motors.
- The measurement input, $\mathbf{z}$, is the odometry from each wheel and the yaw rate measured by a gyroscope.

Unfortunately, such a system turns out to be non-linear and cannot be modelled by (6.1). Nevertheless, the state-space model is useful for many robotic sub-systems.

[2] θ is the heading angle (yaw).

[3] ω is the angular speed.

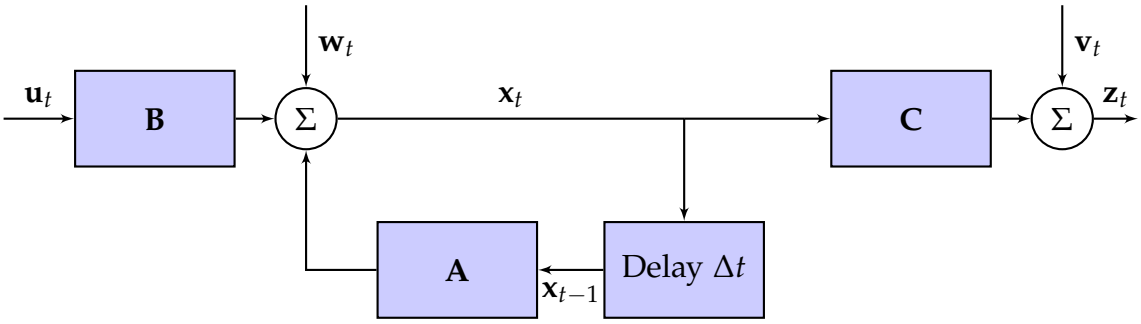


Figure 6.2: Discrete linear state-space system block diagram with measurement noise $\mathbf{v}_t$ and process noise $\mathbf{w}_t$, assuming $\mathbf{D} = 0$.

2 *Noisy state-space model*

In practice, we need to consider both process noise and measurement noise, and the feed-through is usually zero[4], so the state-space equations (6.1) become

$$\begin{aligned} \mathbf{x}_t &= \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t, \\ \mathbf{z}_t &= \mathbf{C}\mathbf{x}_t + \mathbf{v}_t. \end{aligned} \tag{6.2}$$

Here $\mathbf{w}_t$ is a vector of zero-mean random variables that models the process noise and $\mathbf{v}_t$ is a vector of zero-mean random variables that models the measurement noise. A block diagram of the system is shown in Figure 6.2.

[4] $\mathbf{D} = 0$.

3 *Multiple variable Kalman filter*

Kalman filters can be applied to systems with many state variables. Again there are two steps, predict and update (see Figure 6.3 and Figure 6.4).

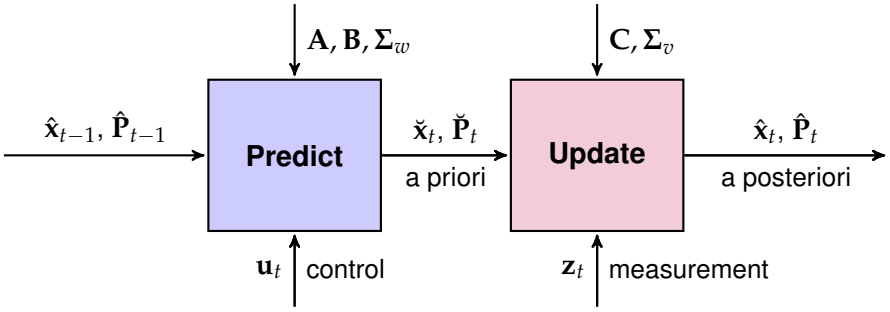


Figure 6.3: Kalman filter predict and update steps.

3.1 *Predict step*

The predicted (*a priori*) mean state estimate is

$$\check{\mathbf{x}}_t = \mathbf{A}\hat{\mathbf{x}}_{t-1} + \mathbf{B}\mathbf{u}_t, \tag{6.3}$$

and the predicted (*a priori*) state covariance[5] estimate is

[5] This is a generalisation of variance for multiple random variables.

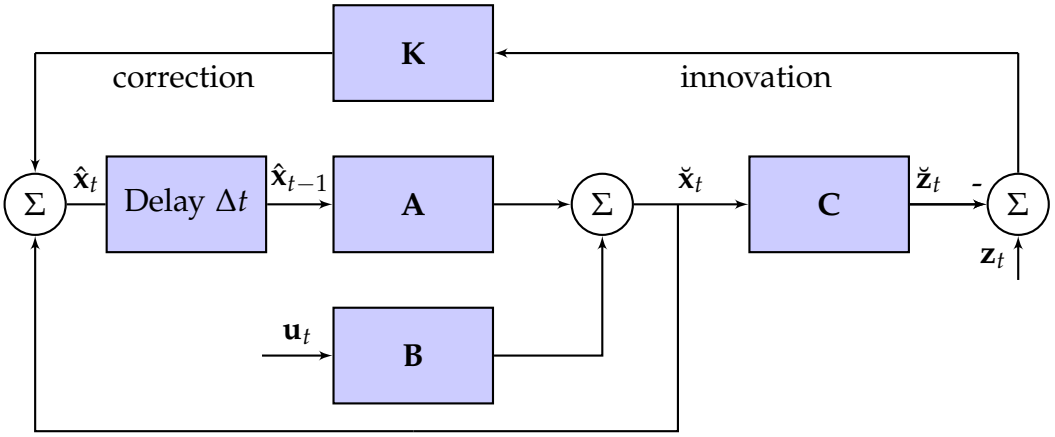


Figure 6.4: Discrete state-space Kalman filter block diagram. This shows how the state is estimated. A similar diagram would show how the covariance of the state is estimated.

$$\check{\mathbf{P}}_t = \mathbf{A}\hat{\mathbf{P}}_{t-1}\mathbf{A}^{\mathrm{T}} + \mathbf{\Sigma}_w, \tag{6.4}$$

where, $\mathbf{\Sigma}_w = \mathrm{E}\left\{\mathbf{w}\mathbf{w}^{\mathrm{T}}\right\}$ is the process noise covariance matrix[6].

3.2 *Update step*

The updated (*a posteriori*) mean state estimate is[7]

$$\hat{\mathbf{x}}_t = \check{\mathbf{x}}_t + \mathbf{K}_t\left(\mathbf{z}_t - \mathbf{C}\check{\mathbf{x}}_t\right), \tag{6.5}$$

and the updated (*a posteriori*) state covariance estimate is

$$\hat{\mathbf{P}}_t = \left(\mathbf{I} - \mathbf{K}_t\mathbf{C}\right)\check{\mathbf{P}}_t, \tag{6.6}$$

where $\mathbf{K}_t$ is the optimal Kalman gain

$$\mathbf{K}_t = \check{\mathbf{P}}_t\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\check{\mathbf{P}}_t\mathbf{C}^{\mathrm{T}} + \mathbf{\Sigma}_v\right)^{-1}. \tag{6.7}$$

Here, $\mathbf{\Sigma}_v = \mathrm{E}\left\{\mathbf{v}\mathbf{v}^{\mathrm{T}}\right\}$ is the measurement noise covariance matrix[8].

3.3 *Comments*

The state of the Kalman filter is represented by:

$\hat{\mathbf{x}}_t$ — the *a posteriori* mean state estimate[9].

$\hat{\mathbf{P}}_t$ — the *a posteriori* error covariance matrix.

Together, these are sufficient to describe a Gaussian belief distribution.

The predict and update steps alternate but if no measurements are available, the update step[10] is skipped.

The Kalman filter requires a matrix inversion and this may require too much computation for a microcontroller. A variant called the Steady State Kalman filter precomputes the Kalman gain to avoid this matrix inversion at each time step, see Section 6.

[6] In general, this can vary with time.

[7] The first term is the predicted state estimate. The second term is called the correction since it corrects the predicted state using the measurement. It is the product of the Kalman gain and the innovation, the difference between the actual and predicted measurements.

[8] In general, this can vary with time.

[9] This is often just called the state estimate.

[10] This step only uses the measurement $\mathbf{z}_t$.

4 Single state variable

The Kalman filter equations for a single state variable, x, single control action, u, and single measurement, z, are:

$$\begin{aligned}
\breve{x}_t &= A\hat{x}_{t-1} + Bu_t && (6.8)\\
\sigma^2_{\breve{x}_t} &= A^2\sigma^2_{\hat{x}_{t-1}} + \sigma^2_w && (6.9)\\
K_t &= \frac{C\sigma^2_{\breve{x}_t}}{C^2\sigma^2_{\breve{x}_t} + \sigma^2_v} && (6.10)\\
\hat{x}_t &= \breve{x}_t + K_t\left(z_t - C\breve{x}_t\right) && (6.11)\\
\sigma^2_{\hat{x}_t} &= \left(1 - K_tC\right)\sigma^2_{\breve{x}_t} = \frac{\sigma^2_v\sigma^2_{\breve{x}_t}}{C^2\sigma^2_{\breve{x}_t} + \sigma^2_v} && (6.12)
\end{aligned}$$

5 Alternative implementations

There are many descriptions of the Kalman filter. Some model the state and output equations as[11]

[11] This has a different block diagram to Figure 6.1.

$$\begin{aligned}
\mathbf{x}_{t+1} &= \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t,\\
\mathbf{z}_t &= \mathbf{C}\mathbf{x}_t + \mathbf{D}\mathbf{u}_t + \mathbf{v}_t. && (6.13)
\end{aligned}$$

This leads to the state being estimated by

$$\hat{\mathbf{x}}_{t+1} = \left(\mathbf{A}\hat{\mathbf{x}}_t + \mathbf{B}\mathbf{u}_t\right) + \mathbf{K}_t\left(\mathbf{z}_t - \mathbf{C}\hat{\mathbf{x}}_t\right), \qquad (6.14)$$

where the Kalman gain is

$$\mathbf{K}_t = \mathbf{A}\mathbf{P}_t\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\mathbf{P}_t\mathbf{C}^{\mathrm{T}} + \mathbf{\Sigma}_v\right)^{-1}, \qquad (6.15)$$

and the estimated covariance matrix is

$$\mathbf{P}_{t+1} = \mathbf{A}\mathbf{P}_t\mathbf{A}^{\mathrm{T}} + \mathbf{\Sigma}_w - \mathbf{A}\mathbf{P}_t\mathbf{C}^{\mathrm{T}}\mathbf{\Sigma}_v^{-1}\mathbf{C}\mathbf{P}_t\mathbf{A}^{T}. \qquad (6.16)$$

6 Steady-state Kalman filter

The Kalman filter requires a matrix inversion. To avoid this overhead, one solution is to use pre-computed Kalman gains ($\mathbf{K}_t = \mathbf{K}$). This assumes that the noise statistics are unchanging, the system is time-invariant[12], and the system is stable. With constant Kalman gains, the steady-state Kalman filter equations simplify to:

[12] With a time-invariant system, the state-space matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ do not change.

$$\begin{aligned}
\breve{\mathbf{x}}_t &= \mathbf{A}\hat{\mathbf{x}}_{t-1} + \mathbf{B}\mathbf{u}_t, && (6.17)\\
\hat{\mathbf{x}}_t &= \breve{\mathbf{x}}_t + \mathbf{K}\left(\mathbf{z}_t - \mathbf{C}\breve{\mathbf{x}}_t\right). && (6.18)
\end{aligned}$$

Given the steady-state covariance, $\mathbf{P}$, the Kalman gain is found from

$$\mathbf{K} = \mathbf{P}\mathbf{C}^{\mathrm{T}} \left(\mathbf{C}\mathbf{P}\mathbf{C}^{\mathrm{T}} + \boldsymbol{\Sigma}_v \right)^{-1}. \tag{6.19}$$

Unfortunately, finding the steady-state covariance, $\mathbf{P}$, is non trivial. It requires solving a matrix Ricatti equation:

$$\mathbf{P} = \mathbf{A} \left(\mathbf{P} - \mathbf{P}\mathbf{C}^{\mathrm{T}} \left(\mathbf{C}\mathbf{P}\mathbf{C}^{\mathrm{T}} + \boldsymbol{\Sigma}_v \right)^{-1} \mathbf{C}\mathbf{P} \right) \mathbf{A}^{\mathrm{T}} + \mathbf{B}\boldsymbol{\Sigma}_w\mathbf{B}^{\mathrm{T}}. \tag{6.20}$$

7 *Summary*

- The Kalman filter is an on-the-fly (online) averaging filter.
- The Kalman gain weights the contribution of each new measurement.
- The Kalman filter stores only the estimated mean state and the estimated state covariance matrix.
- The Kalman filter is only applicable for linear systems.
- The Kalman filter requires the noise to be additive and Gaussian.

8 *Further reading*

- Dan Simon, "Kalman filtering", http://academic.csuohio.edu/simond/courses/eec644/kalman.pdf
- http://en.wikipedia.org/wiki/Kalman_filter

9 *Exercises*

1. What is the difference between *a priori* and *a posteriori*?

2. Can a Kalman filter be applied to a non-linear system?

3. A Kalman filter is applied to estimate the location of a robot using two state variables. How many floating point words of memory does the Kalman filter need to store?

4. A Kalman filter is applied to estimate the pose of a robot using ten state variables. How many floating point words of memory does the Kalman filter need to store?

5. Given the block diagram for a Kalman filter, Figure 6.4, write down the state-space equations.

6. Given the state-space equations (6.1), draw the block diagram for the system.

7. Given the state-space equations (6.1), draw the block diagram for a Kalman filter.

8. What does $\mathbf{z}_t - \mathbf{C}\check{\mathbf{x}}_t$ represent?

9. Under what conditions does the Kalman gain converge to a constant vector?

10. What is a steady state Kalman filter?

11. A Kalman filter can be implemented using the following equations:

$$\begin{aligned}
\check{\mathbf{x}}_t &= \mathbf{A}\hat{\mathbf{x}}_{t-1} + \mathbf{B}\mathbf{u}_t && (6.21)\\
\check{\mathbf{P}}_t &= \mathbf{A}\hat{\mathbf{P}}_{t-1}\mathbf{A}^{\mathrm{T}} + \mathbf{\Sigma}_w && (6.22)\\
\mathbf{K}_t &= \check{\mathbf{P}}_t\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\check{\mathbf{P}}_t\mathbf{C}^{\mathrm{T}} + \mathbf{\Sigma}_v\right)^{-1} && (6.23)\\
\hat{\mathbf{x}}_t &= \check{\mathbf{x}}_t + \mathbf{K}_t\left(\mathbf{z}_t - \mathbf{C}\check{\mathbf{x}}_t\right) && (6.24)\\
\hat{\mathbf{P}}_t &= \left(\mathbf{I} - \mathbf{K}_t\mathbf{C}\right)\check{\mathbf{P}}_t && (6.25)
\end{aligned}$$

Consider these equations and complete the table for a system with a three element state vector, a two element measurement vector, and a one element control vector.

Symbol	Description	Dimension
$\mathbf{z}_t$	*Measurement vector at time step t*	2×1
$\mathbf{u}_t$		
$\hat{\mathbf{x}}_t$		
$\breve{\mathbf{x}}_t$		
$\hat{\mathbf{P}}_t$		
$\mathbf{\Sigma}_w$		
$\mathbf{A}$		
$\mathbf{C}$		
$\mathbf{K}_t$		
$\mathbf{\Sigma}_v$		
$\mathbf{I}$		

7
Kalman filter examples

This lecture considers two example Kalman filter examples, with one and two state variables. Both are based on a probabilistic state-space model, where the state-space and output equations repeated from (6.2) are

$$\begin{aligned} \mathbf{x}_t &= \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t, \\ \mathbf{z}_t &= \mathbf{C}\mathbf{x}_t + \mathbf{v}_t. \end{aligned} \tag{7.1}$$

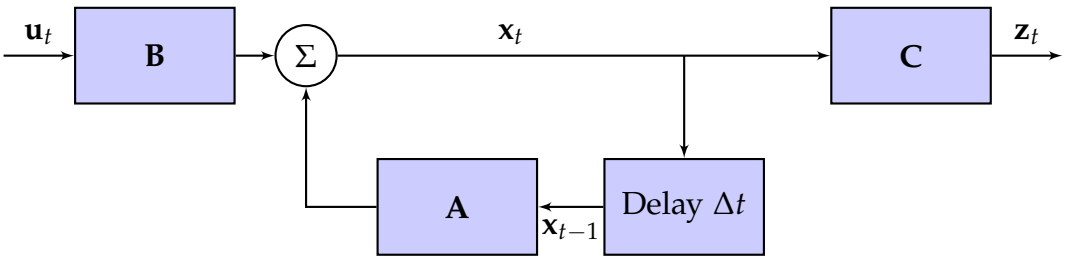


Figure 7.1: Discrete state-space system block diagram.

1 *State-space Kalman filter example 1*

Let's reconsider the problem of a Dalek moving in a straight line where the control input is speed (a scalar),

$$\mathbf{u}_t = \begin{bmatrix} v_t \end{bmatrix}. \tag{7.2}$$

The only required state is the Dalek's position x_t, thus the state vector is a scalar,

$$\mathbf{x}_t = \begin{bmatrix} x_t \end{bmatrix}. \tag{7.3}$$

Let's assume the measurement vector is the measured position

$$\mathbf{z}_t = \begin{bmatrix} z_t \end{bmatrix}. \tag{7.4}$$

Ignoring friction, a speed v_t will change the position to

$$x_t = x_{t-1} + v_t \Delta t, \tag{7.5}$$

where Δt is the time step duration. In practice, the position will be perturbed by wind gusts, potholes, wheel slippage, etc., so a better model is

$$x_t = x_{t-1} + v_t \Delta t + \tilde{x}_t, \tag{7.6}$$

where $\tilde{x}_t$ is a zero-mean Gaussian random variable that models the position uncertainty.

Combining (7.28) and (7.27), yields[1] the state equation

$$\begin{bmatrix} x_t \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \end{bmatrix} + \begin{bmatrix} \Delta t \end{bmatrix} v_t + \begin{bmatrix} \tilde{x}_t \end{bmatrix}. \tag{7.7}$$

The output equation[2] is

$$z_t = \begin{bmatrix} 1 \end{bmatrix} x_t + \tilde{z}_t, \tag{7.8}$$

where $\tilde{z}_t$ is the measurement noise.

Let's assume that the position sensor standard deviation is 0.1 m, i.e., $\sigma_{\tilde{z}} = 0.1\,\text{m}$. Since the measurement is a scalar, the measurement covariance matrix is also a scalar,

$$\begin{aligned} \mathbf{\Sigma}_v &= \mathrm{E}\left\{ \mathbf{v}\mathbf{v}^{\mathrm{T}} \right\}, && (7.9) \\ &= \mathrm{E}\left\{ \tilde{z}^2 \right\} = \sigma_{\tilde{z}}^2. && (7.10) \end{aligned}$$

Note, the error is assumed to be zero-mean, $\mathrm{E}\{\tilde{z}\} = 0$.

The process noise covariance matrix is given by

$$\begin{aligned} \mathbf{\Sigma}_w &= \mathrm{E}\left\{ \mathbf{w}\mathbf{w}^{\mathrm{T}} \right\}, && (7.11) \\ &= \begin{bmatrix} \mathrm{E}\left\{ \tilde{x}^2 \right\} \end{bmatrix}, && (7.12) \\ &= \sigma_{\tilde{x}}^2. && (7.13) \end{aligned}$$

Using the Kalman filter equations, noting that all quantities are scalars for this example, the predicted mean state estimate is

$$\check{x}_t = \hat{x}_{t-1} + v_t \Delta t, \tag{7.14}$$

and the predicted state variance is

$$\sigma_{\check{x}_t}^2 = \sigma_{\hat{x}_{t-1}}^2 + \sigma_{\tilde{x}}^2. \tag{7.15}$$

For the update step, the Kalman gain is

$$K_t = \frac{\sigma_{\hat{x}_{t-1}}^2 + \sigma_{\tilde{x}}^2}{\sigma_{\hat{x}_{t-1}}^2 + \sigma_{\tilde{x}}^2 + \sigma_{\tilde{z}}^2}, \tag{7.16}$$

[1] $\mathbf{A} = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{B} = \begin{bmatrix} \Delta t \end{bmatrix}$, $\mathbf{w}_t = \begin{bmatrix} \tilde{x}_t \end{bmatrix}$.

[2] $\mathbf{C} = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{v}_t = \begin{bmatrix} \tilde{z}_t \end{bmatrix}$.

the updated mean state estimate is

$$\hat{x}_t = \check{x}_t + K_t \left(z_t - \check{x}_t \right), \tag{7.17}$$

and the updated state variance is

$$\sigma^2_{\hat{x}_t} = \left(1 - K_t \right) \sigma^2_{\check{x}_t}. \tag{7.18}$$

Note, that if $v_t = 0$ and $\sigma^2_{\tilde{x}} = 0$, this example is equivalent to an averaging filter.

In practice, it is hard to estimate the covariance of the process noise since it depends on the unknown environment. We can approximate it by assuming that the control speed is noisy. Let's assume that the speed noise standard deviation is $\sigma_{\tilde{v}} = 0.1\,\mathrm{m/s}$. Using the propagation of uncertainty,

$$\begin{aligned}
\mathbf{\Sigma}_w &= \mathrm{E}\left\{ \left(\mathbf{B}\tilde{v} \right) \left(\mathbf{B}\tilde{v} \right)^{\mathrm{T}} \right\}, && (7.19)\\
&= \mathbf{B}\,\mathrm{E}\left\{ \tilde{v}^2 \right\} \mathbf{B}^{\mathrm{T}}, && (7.20)\\
&= \mathbf{B}\sigma^2_{\tilde{v}}\mathbf{B}^{\mathrm{T}}, && (7.21)\\
&= \left[\Delta t \right] \sigma^2_{\tilde{v}} \left[\Delta t \right], && (7.22)\\
&= \Delta t^2 \sigma^2_{\tilde{v}}. && (7.23)
\end{aligned}$$

A Python implementation of a Kalman filter for this example is shown in Listing 7.2 and the results are shown in Figure 7.2, Figure 7.3, and Figure 7.4.

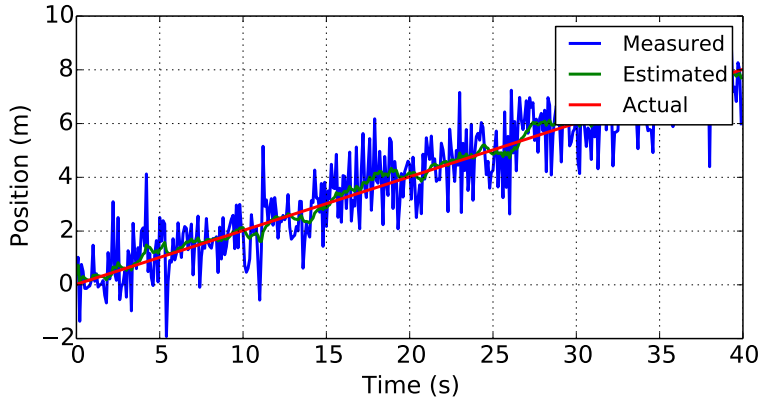


Figure 7.2: Demonstration of state-space Kalman filter for a simple linear robot. With a constant applied speed the robot position increases linearly.

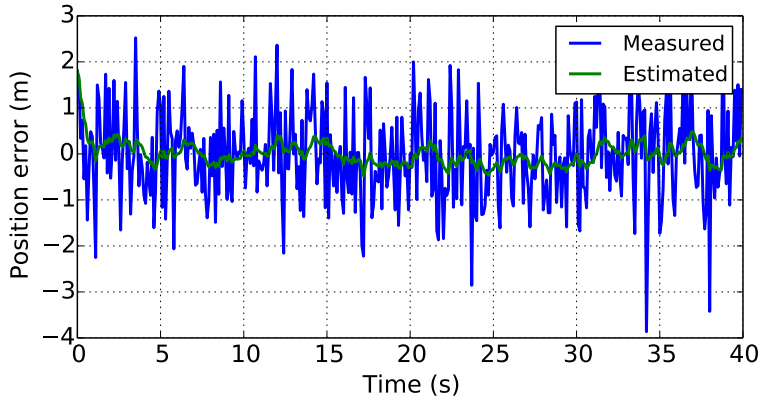


Figure 7.3: Demonstration of state-space Kalman filter for a simple linear robot showing the measurement and estimated errors.

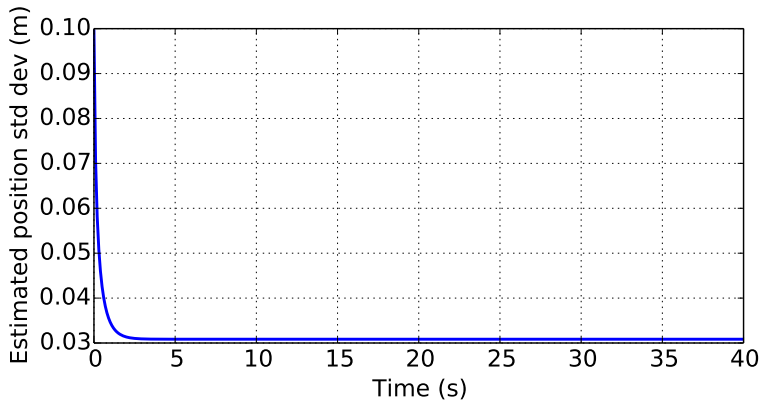


Figure 7.4: Demonstration of state-space Kalman filter for a simple linear robot showing the estimate position standard deviation.

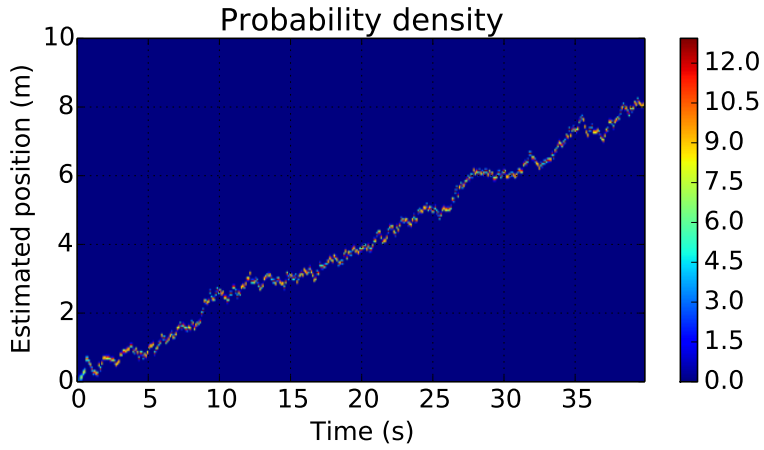


Figure 7.5: Demonstration of state-space Kalman filter for a simple linear robot showing the estimated probability density for the position.

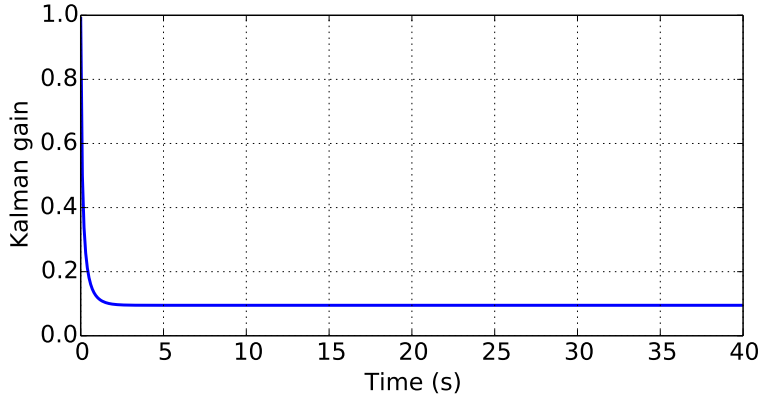


Figure 7.6: Demonstration of state-space Kalman filter for a simple linear robot showing the Kalman gain versus time.

2 State-space Kalman filter example 2

Let's now consider a Dalek moving in a straight line where the control input is acceleration,

$$\mathbf{u}_t = \begin{bmatrix} a_t \end{bmatrix}. \tag{7.24}$$

This will change the Dalek's speed, $v_t = \dot{x}_t$, as well as its position x_t. The required state vector needs two parameters,

$$\mathbf{x}_t = \begin{bmatrix} x_t \\ v_t \end{bmatrix}. \tag{7.25}$$

Again, let's assume that a sensor estimates the Dalek position,

$$\mathbf{z}_t = \begin{bmatrix} z_t \end{bmatrix}. \tag{7.26}$$

Ignoring friction and using simple kinematics, an acceleration a_t will change the speed to

$$v_t = v_{t-1} + a_t \Delta t + \tilde{v}_t, \tag{7.27}$$

where $\tilde{v}_t$ is a zero-mean Gaussian random variable that models the speed uncertainty. Integrating the speed, the position can be modelled by

$$x_t = x_{t-1} + v_t \Delta t + \frac{1}{2}\Delta t^2 a_t + \tilde{x}_t, \tag{7.28}$$

where $\tilde{x}_t$ is a zero-mean Gaussian random variable that models the measured position uncertainty. Combining (7.28) and (7.27), yields[3] the state equation

$$\begin{bmatrix} x_t \\ v_t \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ v_{t-1} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix} a_t + \begin{bmatrix} \tilde{x}_t \\ \tilde{v}_t \end{bmatrix}. \tag{7.29}$$

The output equation[4] is

$$z_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ v_t \end{bmatrix} + \tilde{z}_t, \tag{7.30}$$

where $\tilde{z}_t$ is the measurement noise.

Let's assume that the position sensor standard deviation is 0.1 m, i.e., $\sigma_{\tilde{z}} = 0.1$ m. Since the measurement is a scalar, the measurement covariance matrix is also a scalar,

$$\begin{aligned} \mathbf{\Sigma}_v &= \mathrm{E}\left\{\mathbf{v}\mathbf{v}^{\mathrm{T}}\right\}, && (7.31) \\ &= \mathrm{E}\left\{\tilde{z}^2\right\} = \sigma_{\tilde{z}}^2. && (7.32) \end{aligned}$$

[3] $\mathbf{A} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$, $\mathbf{B} = \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix}$, $\mathbf{w}_t = \begin{bmatrix} \tilde{x}_t \\ \tilde{v}_t \end{bmatrix}$.

[4] $\mathbf{C} = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $\mathbf{v}_t = \begin{bmatrix} \tilde{z}_t \end{bmatrix}$.

Note, the error is assumed to be zero-mean, $\mathrm{E}\{\tilde{z}\} = 0$.

The process noise covariance matrix is given by

$$\begin{aligned}
\boldsymbol{\Sigma}_w &= \mathrm{E}\left\{\mathbf{w}\mathbf{w}^{\mathrm{T}}\right\}, && (7.33)\\
&= \mathrm{E}\left\{\begin{bmatrix}\tilde{x}\\ \tilde{v}\end{bmatrix}\begin{bmatrix}\tilde{x} & \tilde{v}\end{bmatrix}\right\}, && (7.34)\\
&= \begin{bmatrix}\mathrm{E}\{\tilde{x}^2\} & \mathrm{E}\{\tilde{x}\tilde{v}\}\\ \mathrm{E}\{\tilde{v}\tilde{x}\} & \mathrm{E}\{\tilde{v}^2\}\end{bmatrix}. && (7.35)
\end{aligned}$$

In practice, this is hard to estimate since it depends on the unknown environment. We can approximate it by assuming that the control acceleration is noisy. Let's assume that the acceleration noise standard deviation is $\sigma_{\tilde{a}} = 0.1\,\mathrm{m/s^2}$. Using the propagation of uncertainty,

$$\begin{aligned}
\boldsymbol{\Sigma}_w &= \mathrm{E}\left\{(\mathbf{B}\tilde{a})(\mathbf{B}\tilde{a})^{\mathrm{T}}\right\}, && (7.36)\\
&= \mathbf{B}\,\mathrm{E}\left\{\tilde{a}^2\right\}\mathbf{B}^{\mathrm{T}}, && (7.37)\\
&= \mathbf{B}\sigma_{\tilde{a}}^2\mathbf{B}^{\mathrm{T}}, && (7.38)\\
&= \begin{bmatrix}\frac{1}{2}\Delta t^2 & \Delta t\end{bmatrix}\sigma_{\tilde{a}}^2\begin{bmatrix}\frac{1}{2}\Delta t^2\\ \Delta t\end{bmatrix}, && (7.39)\\
&= \begin{bmatrix}\frac{1}{4}\Delta t^4 & \frac{1}{2}\Delta t^3\\ \frac{1}{2}\Delta t^3 & \Delta t^2\end{bmatrix}\sigma_{\tilde{a}}^2. && (7.40)
\end{aligned}$$

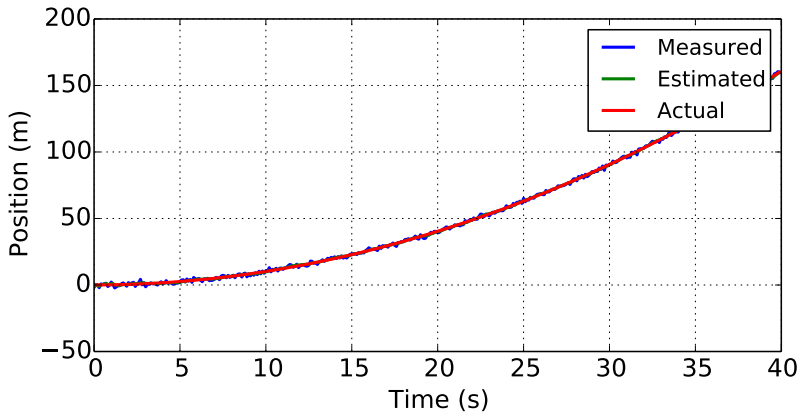


Figure 7.7: Demonstration of state-space Kalman filter for a simple linear robot. With a constant applied acceleration the robot position increases quadratically.

A Python implementation of a Kalman filter for this example is shown in Listing 7.3 and the results are shown in Figure 7.7 and Figure 7.8.

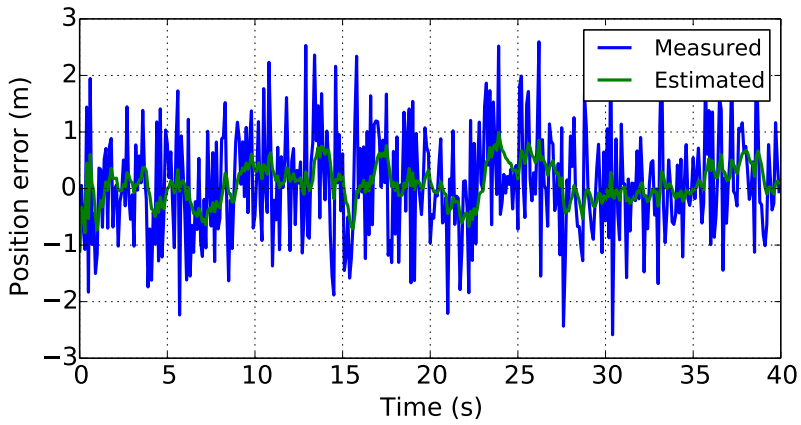


Figure 7.8: Demonstration of state-space Kalman filter for a simple linear robot showing the measurement and estimated position errors.

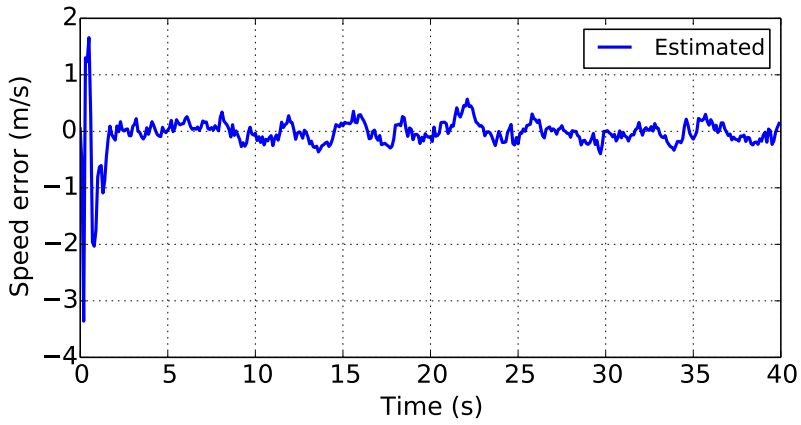


Figure 7.9: Demonstration of state-space Kalman filter for a simple linear robot showing the estimated speed errors.

3 *Practical considerations*

An optimal Kalman filter requires good estimates of the process and measurement noise covariance matrices. Unfortunately, the noise from many sensors is not Gaussian distributed.

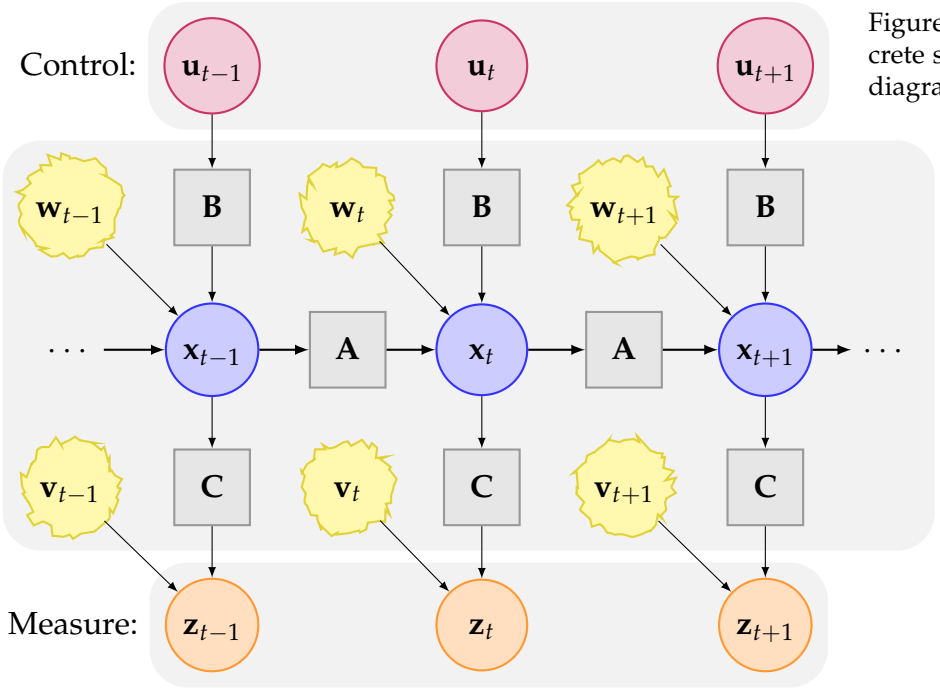


Figure 7.10: Probabilistic discrete state-space system block diagram.

```
from numpy import eye

class KalmanFilter(object):
    def __init__(self, x0, P0, A, B, C, Sigma_v, Sigma_w):
        """Create Kalman Filter with initial guesses of state, x, and
        covariance matrix, P0 for a constant linear state-space model
        described my matrices A, B, and C."""

        # A posteriori state and error covariance estimates
        self.x = x0
        self.P = P0

        self.K = 0

        # State-space matrices
        self.A = A
        self.B = B
        self.C = C

        # Process and measurement nose covariance matrices
        self.Sigma_v = Sigma_v
        self.Sigma_w = Sigma_w

    def predict(self, u):
        """Calculate a priori state estimate and error covariance matrix
        using control vector u"""

        self.x = self.A * self.x + self.B * u
        self.P = self.A * self.P * self.A.T + self.Sigma_w

    def update(self, z):
        """Calculate a posteriori state estimate and error
        covariance matrix using measurement vector z"""

        # Calculate Kalman gain
        C, P = self.C, self.P
        K = P * C.T * (C * P * C.T + self.Sigma_v) ** -1

        I = eye(len(self.x))

        self.x = self.x + K * (z - C * self.x)
        self.P = (I - K * C) * P
        self.K = K

    def estimate(self, u, z):
        """Estimate state given control input vector u and
        measurement vector z"""

        self.predict(u)
        self.update(z)
        return self.x
```

Listing 7.1: Python implementation of a Kalman filter.

```
from linear_robot import LinearRobot
from kalman_filter import KalmanFilter

# Constant speed
u = 0.2

robot = LinearRobot()

# Initial state variable and covariance estimates
x_init = np.matrix(0).T
P_init = np.matrix(1**2)

# Process noise covariance matrix (use smaller values if confident of model)
Sigma_w = np.matrix((0.01**2))
# Measurement noise covariance matrix (assuming no corelation)
Sigma_v = np.matrix((0.1**2))

# Use the same state-space matrices as the simulation
kf = KalmanFilter(x_init, P_init, robot.A, robot.B, robot.C, Sigma_v, Sigma_w)

for m in range(M):
    # Simulate robot
    x, z = robot.update(u)

    # Estimate robot state
    x_est = kf.estimate(u, z)
```

Listing 7.2: Python implementation of a Kalman filter for a simple linear robot with speed control.

```
from linear_robot2 import LinearRobot2
from kalman_filter import KalmanFilter

# Constant acceleration
u = 0.2

M = 400
x_history = np.zeros(M)
x_z_history = np.zeros(M)
x_est_history = np.zeros(M)

robot = LinearRobot2()

# Initial state variable and covariance estimates
x_init = np.matrix((0, 0)).T
P_init = np.matrix(((1**2, 0),
                    (0, 1**2)))

# Use the same state-space matrices as the simulation
kf = KalmanFilter(x_init, P_init, robot.A, robot.B, robot.C)

# Process noise covariance matrix (use smaller values if confident of model)
Sigma_w = np.matrix((0.01**2))
# Measurement noise covariance matrix (assuming no corelation)
Sigma_v = np.matrix((0.1**2))

for m in range(M):
    x, z = robot.update(u)

    x_est = kf.estimate(u, z, Sigma_v, Sigma_w)

    x_history[m] = x[0]
    x_z_history[m] = z
    x_est_history[m] = x_est[0]
```

Listing 7.3: Python implementation of a Kalman filter for a simple linear robot with acceleration control.

8
Bayes filters

The Kalman filter only works for linear systems with additive Gaussian process and sensor noise. Unfortunately, most systems are non-linear, so what can we do? To find the answer, we need a more general approach to the Kalman filter. To set the scene, let's start with a little problem...

> Half the robots in a room are evil. 2/3 of the evil robots are square; 1/4 of the good robots are square. If you meet a square robot, what are the chances that it is evil[1]?

[1] Problems such as this are solved using Bayes' theorem.

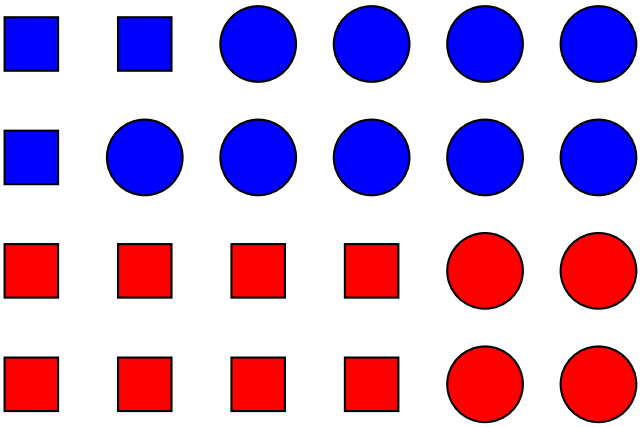


Figure 8.1: The blue robots are good; the red robots are evil. If you meet a square robot, what is the probability that it is evil?

A related problem is the estimation of a robot's state. Consider the measurement, $\mathbf{z}$, of some parameter of the robot's environment (the state). Since the sensor has noise, this can be modelled[2] as

$$\mathbf{z} = h(\mathbf{x}) + \mathbf{v}, \tag{8.1}$$

where h is the sensor model and $\mathbf{v}$ denotes the sensor noise. The goal is to estimate the state $\mathbf{x}$ given the measurement $\mathbf{z}$ and control action $\mathbf{u}$.

In this lecture, we see that the estimate of $\mathbf{x}$ can be greatly improved if we have some prior knowledge of $\mathbf{x}$ and of the sensor noise $\mathbf{v}$.

[2] The feedthrough effect of control actions are ignored. This is equivalent in the linear case to $\mathbf{D} = 0$.

1 Bayes' theorem

Bayes' theorem[3] is usually stated in probability form as

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}, \tag{8.2}$$

since

$$P(A \cap B) = P(A|B)\,P(B) = P(B|A)\,P(A). \tag{8.3}$$

Here $P(A|B)$ is the *conditional probability*[4] for A given B and $P(B|A)$ is the *conditional probability* for event B given event A.

The simplest way to solve the evil robot problem is to note that in Figure 8.1 there are 3 good square robots and 8 evil square robots. The probability of selecting an evil square robot is

$$P(E|S) = \frac{8}{8+3} = \frac{8}{11}. \tag{8.4}$$

Here E denotes an evil robot and S denotes a square robot. Initially, the prior probability of a robot being evil is 0.5 but with the additional knowledge that it is square increases the probability to $8/11 \approx 0.73$.

We can also solve the problem using Bayes' theorem. We know that the probability of a robot being evil is $P(E) = 1/2$. We also know the conditional probability that $1/4$ of the good robots are square, $P(S|\overline{E}) = 1/4$, and the conditional probability that $2/3$ of the evil robots are square, $P(S|E) = 2/3$. To use Bayes' theorem we need to determine $P(S)$, the probability that a robot is square. This can be determined from the law of total probability[5] ,

$$\begin{aligned} P(S) &= P(S|E)P(E) + P(S|\overline{E})P(\overline{E}), && (8.5)\\ &= \frac{2}{3} \times \frac{1}{2} + \frac{1}{4} \times \frac{1}{2}, && (8.6)\\ &= \frac{11}{24}. && (8.7) \end{aligned}$$

Now we can apply Bayes' theorem,

$$\begin{aligned} P(E|S) &= \frac{P(S|E)\,P(E)}{P(S)}, && (8.8)\\ &= \frac{\frac{2}{3} \times \frac{1}{2}}{\frac{2}{3} \times \frac{1}{2} + \frac{1}{4} \times \frac{1}{2}}, && (8.9)\\ &= \frac{\frac{1}{3}}{\frac{11}{24}} = \frac{8}{11} \approx 0.73. && (8.10) \end{aligned}$$

Notice that the additional information has allowed us to more reliably determine if a robot is evil.

[3] Bayes' rule is Bayes' theorem in odds form.

[4] Using conditional probabilities is important to improve estimates. A classic example is the Monty Hall game show problem: *Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others a donkey. You pick a door, say No.1, and the host, who knows what's behind the doors, opens another door, say No.3, which has a donkey. He then says to you, "Do you want to pick door No.2?" Is it to your advantage to switch your choice?*

[5] We can check this result by counting the proportion of squares in Figure 8.1.

1.1 Continuous random variables

For two continuous random variables X and Z, Bayes' theorem can also be stated as

$$f_{X|Z}(x|z) = \frac{f_{Z|X}(z|x)\, f_X(x)}{f_Z(z)}. \tag{8.11}$$

where

$f_{X|Z}(x|z)$ is called the posterior probability density,

$f_X(x)$ is called the prior probability density,

$f_{Z|X}(z|x)$ is called the likelihood,

$f_Z(z)$ is the probability density of the data (the evidence).

Equation (8.11) can be derived from the joint probability density of X and Z, $f_{X,Z}(x,z)$, since

$$f_{X|Z}(x|z) = \frac{f_{X,Z}(x,z)}{f_Z(z)}, \tag{8.12}$$

$$f_{Z|X}(z|x) = \frac{f_{X,Z}(x,z)}{f_X(x)}. \tag{8.13}$$

1.2 Discrete random variables

For two discrete random variables X and Z, Bayes' theorem can also be stated as

$$p_{X|Z}(x|z) = \frac{p_{Z|X}(z|x)\, p_X(x)}{p_Z(z)}. \tag{8.14}$$

Here $p_{X|Z}(x|z)$ is the conditional probability mass function of X given Z.

1.3 Generalised random variables

Perhaps in an attempt to generalise Bayes' theorem for continuous and discrete random variables or just because of sloppiness[6], Bayes' theorem is often written as

$$p(x|z) = \frac{p(z|x)\, p(x)}{p(z)}, \tag{8.15}$$

or as

$$p(x|z) = \eta p(z|x) p(x), \tag{8.16}$$

where $\eta = 1/p(z)$ is a normalising factor[7],

$$\eta = \frac{1}{p(z)} = \frac{1}{\int p(z|x)p(x)\mathrm{d}x}. \tag{8.17}$$

[6] The disadvantage is that $p(x)$ and $p(z)$ are different probability density (or mass) functions but appear to be the same.

[7] Found using the law of total probability. For discrete distributions, the integral is replaced with a summation.

2 State-space belief

A robot can manipulate its environment as well as sense it (see Figure 8.2). So, in general, a robot is interested in estimating its (and the environment's) state, $\mathbf{x}_t$, given all the past measurements $\mathbf{z}_{0:t}$ and past controls $\mathbf{u}_{0:t}$.

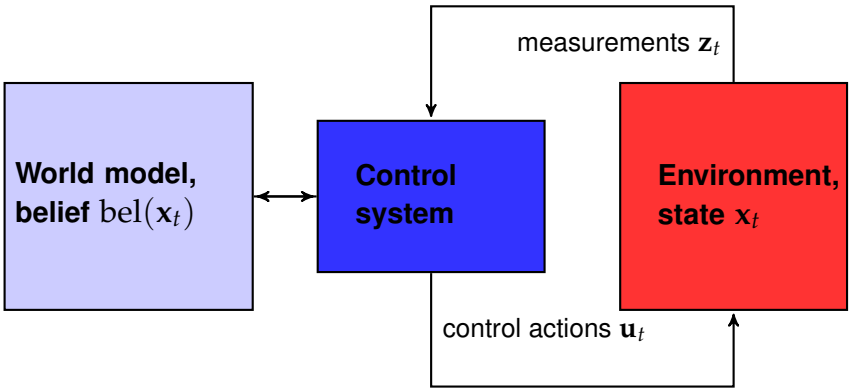


Figure 8.2: Robot environment interaction.

The belief of a system's state, $\mathbf{x}_t$, given all the past controls $\mathbf{u}_{0:t}$ and all the past measurements $\mathbf{z}_{0:t}$, is represented as a conditional probability distribution,

$$\text{bel}(\mathbf{x}_t) = p(\mathbf{x}_t|\mathbf{u}_{0:t}, \mathbf{z}_{0:t}). \tag{8.18}$$

This is a posterior distribution since it depends on measurements.

Prior information about the system is encoded by conditional probabilities[8]:

$p(\mathbf{x}_t|\mathbf{x}_{0:t-1}, \mathbf{u}_{0:t})$ The *state transition probability*, i.e., the probability of the state, $\mathbf{x}_t$, given all the previous states, $\mathbf{x}_{0:t-1}$ and the current and previous controls, $\mathbf{u}_{0:t}$.

$p(\mathbf{z}_t|\mathbf{x}_{0:t}, \mathbf{u}_{0:t})$ The *measurement likelihood*, i.e., the probability of the sensor measurement, $\mathbf{z}_t$, given the current and previous states, $\mathbf{x}_{0:t}$ and controls[9] $\mathbf{u}_{0:t}$.

[8] The Kalman filter assumes a deterministic system with additive Gaussian process and measurement noise.

[9] In general, these can be neglected. This is equivalent to $\mathbf{D} = 0$ in a state-space model.

3 Complete state assumption

The *complete state*[10] assumption assumes that no past measurement or control would provide additional information. This simplifies the conditional probabilities:

$\text{bel}(\mathbf{x}_t) = p(\mathbf{x}_t|\mathbf{u}_{0:t}, \mathbf{z}_{0:t}) = p(\mathbf{x}_t|\mathbf{u}_t, \mathbf{z}_t)$ (*state belief*).

$p(\mathbf{x}_t|\mathbf{x}_{0:t-1}, \mathbf{u}_{0:t}) = p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$ (*transition probability*).

$p(\mathbf{z}_t|\mathbf{x}_{0:t}, \mathbf{u}_{0:t}) = p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{u}_t)$ (*measurement likelihood*).

[10] Also called the Markov assumption.

4 Bayes filter (recursive Bayesian estimation)

Using Bayes' theorem and the assumption of *complete state*, the belief at time step t can be recursively estimated from the belief at the previous time step using a two step process[11] (see Figure 8.3):

$$\text{predict: } \breve{\text{bel}}(\mathbf{x}_t) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)\, \text{bel}(\mathbf{x}_{t-1}) \mathrm{d}\mathbf{x}_{t-1} \quad (8.19)$$

$$\text{update: } \text{bel}(\mathbf{x}_t) = \eta p(\mathbf{z}_t|\mathbf{x}_t) \breve{\text{bel}}(\mathbf{x}_t), \quad (8.20)$$

where η is a normalising factor and

$$\breve{\text{bel}}(\mathbf{x}_t) = p(\mathbf{x}_t|\mathbf{u}_t), \quad (8.21)$$

$$\text{bel}(\mathbf{x}_t) = p(\mathbf{x}_t|\mathbf{u}_t, \mathbf{z}_t). \quad (8.22)$$

[11] Just like the Kalman filter. Indeed, the Kalman filter is a special case of a Bayes filter.

The first step[12] predicts the belief from the belief at the previous time step using the state transition probability and the current control action, $\mathbf{u}_t$. The second step[13] updates the belief based on the new sensor measurement, $\mathbf{z}_t$.

[12] This propagates the belief density and is equivalent to a convolution when the model is linear.
[13] This used Bayes' theorem.

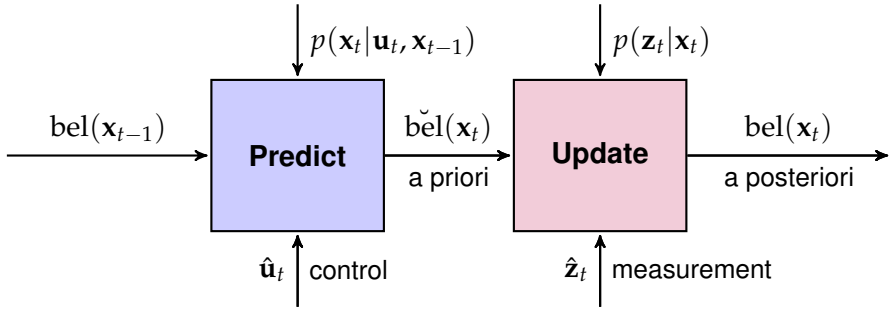


Figure 8.3: Bayes filter predict and update steps. $\breve{\text{bel}}(\mathbf{x}_t)$ is the predicted belief of the state $\mathbf{x}$ at time step t.

5 1-D Bayes filter example

Let's consider the Dalek position estimation problem again. We modelled the state transition with

$$x_t = x_{t-1} + v\Delta t + \tilde{x}, \tag{8.23}$$

where $\tilde{x}$ was an additive Gaussian random variable modelling the process noise. We can express this using a conditional probability distribution (see Figure 8.4),

$$p(x_t|x_{t-1}) = \frac{1}{\sqrt{2\pi\sigma_x^2}} \exp\left(-0.5\frac{(x_t - x_{t-1} - v\Delta t)^2}{\sigma_x^2}\right). \tag{8.24}$$

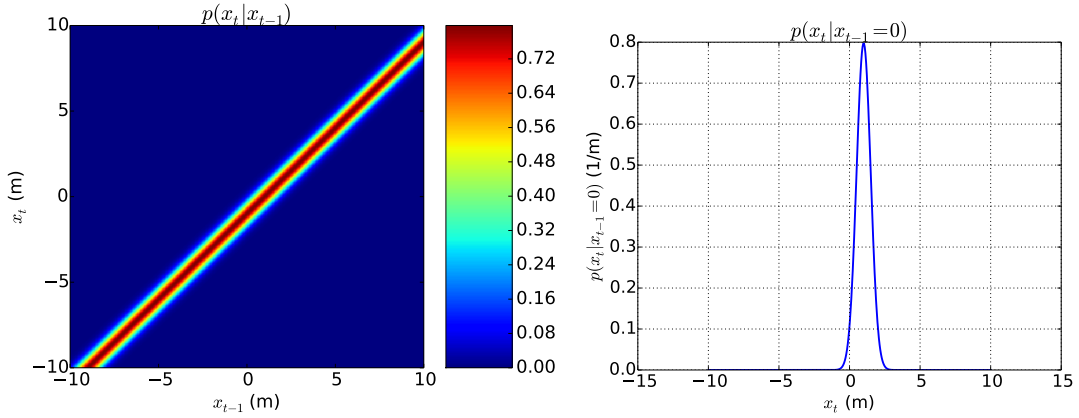


Figure 8.4: State transition probability distribution $p(x_t|x_{t-1})$ for $x_t = x_{t-1} + v\Delta t + \tilde{x}$, where $\tilde{x}$ is a Gaussian random variable.

Using the state transition condition probability we can propagate the belief using (8.19). This is shown in Figure 8.5 for a belief with multiple hypotheses.

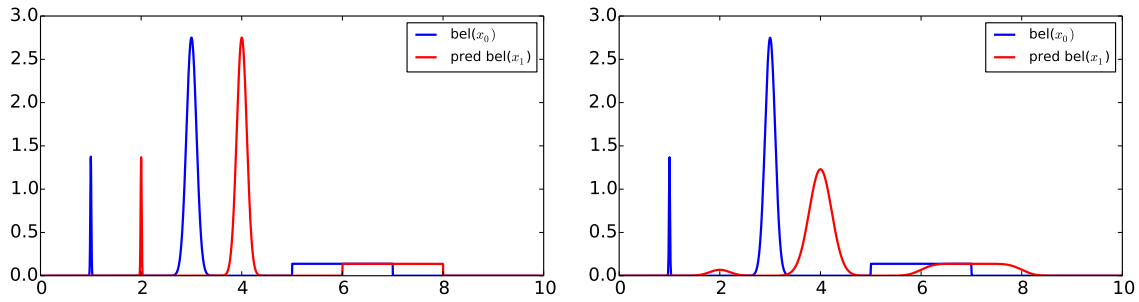


Figure 8.5: Belief propagation using the state transition conditional probability. The first example has $\sigma_x = 0.001$; the second uses $\sigma_x = 0.2$.

Similarly, we modelled[14] the sensor measurement with

$$z_t = x_t + \tilde{z}, \tag{8.25}$$

where $\tilde{z}$ was an additive Gaussian random variable representing the sensor noise. The equivalent conditional probability distribution is (see Figure 8.6),

$$p(z_t|x_t) = \frac{1}{\sqrt{2\pi\sigma_z^2}} \exp\left(-0.5\frac{(z_t - x_t)^2}{\sigma_z^2}\right). \tag{8.26}$$

[14] This is a linear model with $C = 1$.

So given a known position x_t we can find the likelihood of a sensor measurement, z_t.

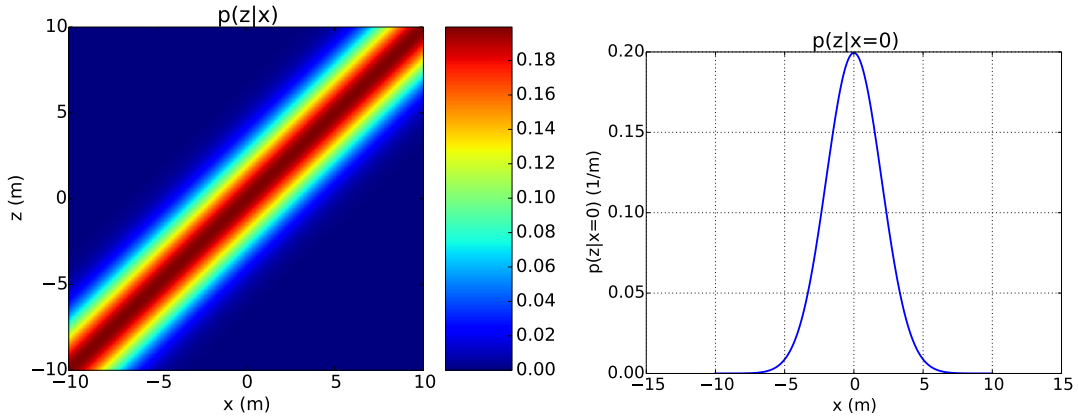


Figure 8.6: Sensor likelihood $p(z_t|x_t)$ for $z_t = x_t + \tilde{z}$,. where $\tilde{z}$ is a Gaussian random variable.

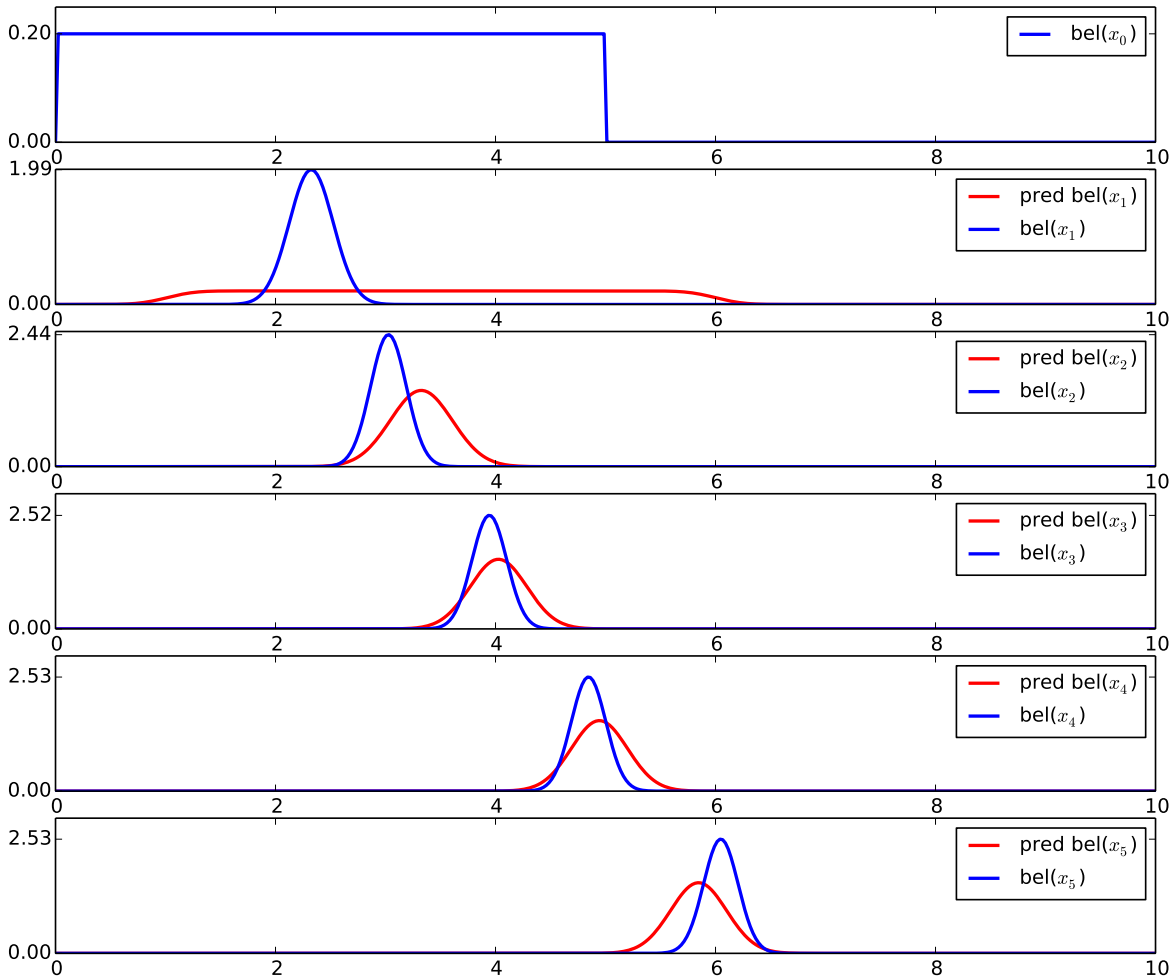


Figure 8.7: 1-D Bayes filter example for a robot moving with a mean speed of 1 m/s and $\Delta t = 1$ s. The sensor standard deviation, σ_z, is 0.5 m.

6 Bayes filter derivation

Let's say we are interested in predicting the state $\mathbf{x}_t$ given all the past measurements $\mathbf{z}_{0:t}$, assuming there are no controls. The belief (posterior density) is

$$\text{bel}(\mathbf{x}_t) = p(\mathbf{x}_t|\mathbf{z}_{0:t}). \tag{8.27}$$

Using Bayes' theorem the belief can be expressed as

$$\begin{aligned} \text{bel}(\mathbf{x}_t) &= \frac{p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{z}_{0:t-1})p(\mathbf{x}_t|\mathbf{z}_{0:t-1})}{p(\mathbf{z}_t|\mathbf{z}_{0:t-1})}, & (8.28)\\ &= \eta p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{z}_{0:t-1})p(\mathbf{x}_t|\mathbf{z}_{0:t-1}), & (8.29) \end{aligned}$$

where η is a normalising factor. Assuming complete state produces the conditional independence,

$$p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{z}_{0:t-1}) = p(\mathbf{z}_t|\mathbf{x}_t). \tag{8.30}$$

In other words, the previous measurements have no effect. The belief can be now expressed as

$$\text{bel}(\mathbf{x}_t) = \eta p(\mathbf{z}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{z}_{0:t-1}). \tag{8.31}$$

Using conditional probabilities,

$$p(\mathbf{x}_t|\mathbf{z}_{0:t-1}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{z}_{0:t-1})p(\mathbf{x}_{t-1}|\mathbf{z}_{0:t-1})\mathrm{d}\mathbf{x}_{t-1}. \tag{8.32}$$

Again assuming complete state,

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{z}_{0:t-1}) = p(\mathbf{x}_t|\mathbf{x}_{t-1}), \tag{8.33}$$

so

$$p(\mathbf{x}_t|\mathbf{z}_{0:t-1}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1})p(\mathbf{x}_{t-1}|\mathbf{z}_{0:t-1})\mathrm{d}\mathbf{x}_{t-1}. \tag{8.34}$$

Now since $p(\mathbf{x}_t|\mathbf{x}_{t-1})$ is a uniform density, (8.34) simplifies to

$$\begin{aligned} p(\mathbf{x}_t|\mathbf{z}_{0:t-1}) &= p(\mathbf{x}_{t-1}|\mathbf{z}_{0:t-1}), & (8.35)\\ &= \text{bel}(\mathbf{x}_{t-1}), & (8.36) \end{aligned}$$

Finally, using (8.36) with (8.31) produces the update step[15] of the Bayes filter

$$\text{bel}(\mathbf{x}_t) = \eta p(\mathbf{z}_t|\mathbf{x}_t)\,\text{bel}(\mathbf{x}_{t-1}). \tag{8.37}$$

[15] This derivation assumes no controls so there is no predict step.

7 General Bayes filter derivation

Let's say we are interested in predicting the state $\mathbf{x}_t$ given all the past measurements $\mathbf{z}_{0:t}$ and past controls $\mathbf{u}_{0:t}$. The belief (posterior density) is

$$\text{bel}(\mathbf{x}_t) = p(\mathbf{x}_t|\mathbf{z}_{0:t}, \mathbf{u}_{0:t}). \tag{8.38}$$

Using Bayes' theorem the belief can be expressed as

$$\begin{aligned} \text{bel}(\mathbf{x}_t) &= \frac{p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{z}_{0:t-1}, \mathbf{u}_{0:t})p(\mathbf{x}_t|\mathbf{z}_{0:t-1}, \mathbf{u}_{0:t})}{p(\mathbf{z}_t|\mathbf{z}_{0:t-1}, \mathbf{u}_{0:t})} && (8.39) \\ &= \eta p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{z}_{0:t-1}, \mathbf{u}_{0:t})p(\mathbf{x}_t|\mathbf{z}_{0:t-1}, \mathbf{u}_{0:t}) && (8.40) \end{aligned}$$

This can be simplified using the assumption of complete state. This assumption produces the *conditional independence*,

$$p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{z}_{0:t-1}, \mathbf{u}_{0:t}) = p(\mathbf{z}_t|\mathbf{x}_t), \tag{8.41}$$

and so the belief can be expressed as

$$\text{bel}(\mathbf{x}_t) = \eta p(\mathbf{z}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{z}_{0:t-1}, \mathbf{u}_{0:t}). \tag{8.42}$$

Using conditional probabilities,

$$p(\mathbf{x}_t|\mathbf{z}_{0:t-1}, \mathbf{u}_{0:t}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{z}_{0:t-1}, \mathbf{u}_{0:t})p(\mathbf{x}_{t-1}|\mathbf{z}_{0:t-1}, \mathbf{u}_{0:t})\mathrm{d}\mathbf{x}_{t-1}. \tag{8.43}$$

Again assuming complete state,

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{z}_{0:t-1}, \mathbf{u}_{0:t}) = p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t), \tag{8.44}$$

so

$$p(\mathbf{x}_t|\mathbf{z}_{0:t-1}, \mathbf{u}_{0:t}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{0:t})p(\mathbf{x}_{t-1}|\mathbf{z}_{0:t-1}, \mathbf{u}_{0:t})\mathrm{d}\mathbf{x}_{t-1}. \tag{8.45}$$

TODO: finish...

8 Probability notation

Probability notation is confusing and sloppily applied[16]. The notation is even inconsistent among statisticians. Part from the problem stems from the use of upper case for a random variable and lower case for their values. However, what if you want to use a Greek letter since not all Greek symbols have an upper case variant. Or what about a random matrix? There is also confusion since some random variables are discrete and some are continuous.

To avoid ambiguity, the consensus is to use the upper-case random variable as a subscript. For example, the conditional probability mass function[17] for a discrete random variable Z given another random variable X is

$$p_{Z|X}(z|x).$$

However, others notably Wikipedia[18], use the notation

$$p_Z(z|X = x).$$

Similarly, the conditional probability density function for a continuous random variable Z given another random variable X is

$$f_{Z|X}(z|x),$$

or alternatively

$$f_Z(z|X = x).$$

[16] See http://lingpipe-blog.com/2009/10/13/whats-wrong-with-probability-notation/

[17] This is the probability distribution for a discrete random variable; c.f., probability density function for a continuous random variable.

[18] https://en.wikipedia.org/wiki/Bayes'_theorem

9 Summary

1. Bayes filters represent the belief of a state by a probability distribution.
2. Using Bayes' theorem the posterior distribution is determined from the prior distribution and conditional probabilities describing the system stochastically.
3. The complete state (Markov) assumption assumes that no past measurement or control would provide additional information.
4. Using the complete state assumption, and an initial belief, the Bayes filter can estimate the belief of a state recursively.

10 Exercises

1. One third of the robots in a room are evil. 2/3 of the evil robots are square; 1/4 of the good robots are square. If you meet a square robot, what are the chances that it is evil?

2. One third of the robots in a room are evil. 1/3 of the evil robots are square; 3/4 of the good robots are square. If you meet a square robot, what are the chances that it is evil?

3. What is the difference between Figure 8.3 and Figure 6.3?

4. What is the advantage of the complete state (Markov) assumption?

5. Determine $p(z_t|x_t)$ for a linear system with a single measurement, z_t, and a single state, x_t assuming that the sensor noise is additive with a zero-mean Gaussian distribution and variance σ_v^2. Assume that the noise free sensor model is $z_t = Cx_t$.

6. Determine $p(x_t|x_{t-1}, u_t)$ for a linear system with a single control, u_t, and a single state, x_t assuming that the process noise is additive with a zero-mean Gaussian distribution and variance σ_w^2. Assume that the noise free state transition model is $x_t = Ax_{t-1} + Bu_t$.

7. Comment on using a lookup table to store $p(x_t|x_{t-1}, u_t)$.

9
Discrete Bayes filters

When the state-space is discrete, sensor fusion can be performed with a discrete Bayes filter. The discrete Bayes filter[1] replaces the integral in the prediction step (8.19) with a summation, so the two steps are:

[1] Also called hidden Markov models (HMM).

$$\text{predict: } \breve{\text{bel}}(\mathbf{x}_t) = \sum_{\mathbf{x}_{t-1}} p(\mathbf{x}_t|\mathbf{u}_t, \mathbf{x}_{t-1}) \, \text{bel}(\mathbf{x}_{t-1}), \quad (9.1)$$

$$\text{update: } \text{bel}(\mathbf{x}_t) = \eta p(\mathbf{z}_t|\mathbf{x}_t) \breve{\text{bel}}(\mathbf{x}_t). \quad (9.2)$$

This lecture looks at a couple of examples.

1 *Discrete Bayes filter example*

Consider a robot with a sensor that senses an open door with 60% probability and a closed door with 80% probability. Let's say the robot wants to estimate the state of the door, with two discrete states: open and closed.

Let's assume there is no process noise and no control so the door cannot transition to another state[2]. So the Bayes filter prediction step (9.1) is unnecessary and only the update step is required (9.2). For this example, the Bayes filter simplifies to:

[2] The conditional probability is a Dirac delta.

$$\text{bel}(x_t) = \eta p(z_t|x_t) \, \text{bel}(x_{t-1}). \quad (9.3)$$

If the robot has no prior knowledge of the initial door state, the initial beliefs[3] are:

[3] Note, these beliefs sum to 1.0 and are equally likely.

$$\begin{aligned} \text{bel}(x_0 = \text{open}) &= 0.5, \\ \text{bel}(x_0 = \text{closed}) &= 0.5. \end{aligned}$$

Now let's say that the robot door sensor returns is-closed and so the robot updates its beliefs using (9.3):

$$\begin{aligned} \text{bel}(x_1 = \text{open}) &= p(z_1 = \text{is-closed}|x_1 = \text{open}) \, \text{bel}(x_0 = \text{open}), \\ &= 0.4 \times 0.5 = 0.20, \\ \text{bel}(x_1 = \text{closed}) &= p(z_1 = \text{is-closed}|x_1 = \text{closed}) \, \text{bel}(x_0 = \text{closed}), \\ &= 0.8 \times 0.5 = 0.40. \end{aligned}$$

Note, these beliefs sum to 0.60 and not 1.0, so we need to normalise them,

$$\text{bel}(x_1 = \text{open}) = 0.20/0.60 = 0.333,$$
$$\text{bel}(x_1 = \text{closed}) = 0.40/0.60 = 0.667.$$

Thus the robot believes that the door is more likely to be closed.

As the robot makes more measurements, it updates its beliefs for the door states. Some example scenarios for the belief that the door is open are shown in Figure 9.1 and Figure 9.2.

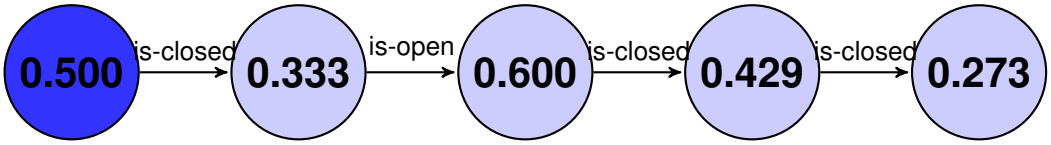


Figure 9.1: Beliefs that the door is open. In this example, the robot trusts the open measurements more than the closed measurements.

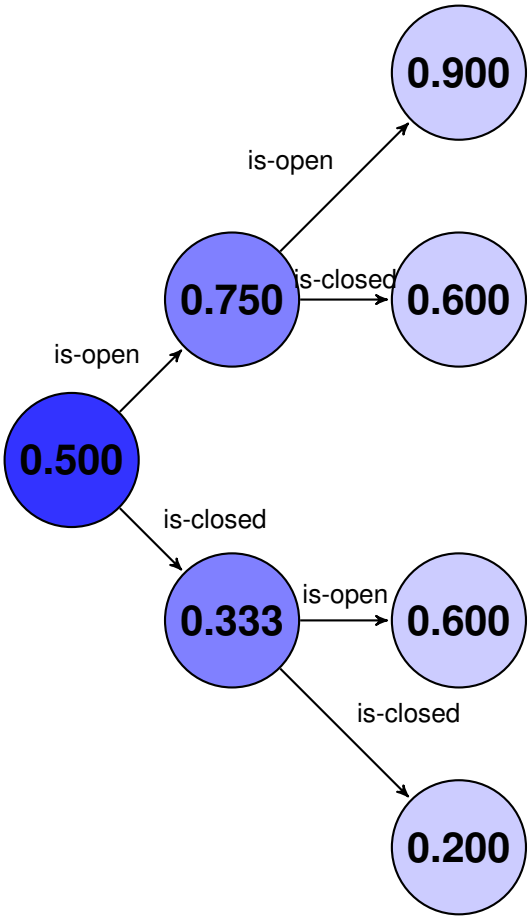


Figure 9.2: Beliefs that the door is open.

2 *Discrete Bayes filter with controls example*

Consider a robot with a manipulator with two control actions: push and do-nothing. The push action can open a door with 80% probability. It also has a sensor that senses an open door with 60% probability and a closed door with 80% probability. Let's say the robot wants to track the state of the door, with two discrete states: open and closed. Denoting the belief at time step t by x_t, the initial beliefs[4] are

[4] Note, these sum to 1.0 and are equally likely.

$$\begin{aligned} \text{bel}(x_0 = \text{open}) &= 0.5, \\ \text{bel}(x_0 = \text{closed}) &= 0.5. \end{aligned}$$

Now let's assume the robot outputs a push action, so $u_1 = \text{push}$. The predicted beliefs are:

$$\begin{aligned} \breve{\text{bel}}(x_1 = \text{open}) &= p(x_1 = \text{open}|u_1 = \text{push}, x_0 = \text{open})\,\text{bel}(x_0 = \text{open}) \\ &\quad + p(x_1 = \text{open}|u_1 = \text{push}, x_0 = \text{closed})\,\text{bel}(x_0 = \text{closed}) \\ &= 1 \times 0.5 + 0.8 \times 0.5 = 0.9 \\ \breve{\text{bel}}(x_1 = \text{closed}) &= p(x_1 = \text{closed}|u_1 = \text{push}, x_0 = \text{open})\,\text{bel}(x_0 = \text{open}) \\ &\quad + p(x_1 = \text{closed}|u_1 = \text{push}, x_0 = \text{closed})\,\text{bel}(x_0 = \text{closed}) \\ &= 0.0 \times 0.5 + 0.2 \times 0.5 = 0.1. \end{aligned}$$

Note, again these sum to 1.0. Finally, to update the beliefs for this time step, the robot needs to consider the sensor measurement. Let's assume that it says is-closed. Using the update step of the Bayes filter, the updated beliefs are:

$$\begin{aligned} \text{bel}(x_1 = \text{open}) &= p(z_1 = \text{is-closed}|x_1 = \text{open})\breve{\text{bel}}(x_1 = \text{open}), \\ &= 0.4 \times 0.9 = 0.36, \\ \text{bel}(x_1 = \text{closed}) &= p(z_1 = \text{is-closed}|x_1 = \text{closed})\breve{\text{bel}}(x_1 = \text{closed}), \\ &= 0.8 \times 0.1 = 0.08, \end{aligned}$$

Note, these sum to 0.44 and not 1.0, so we need to normalise them,

$$\begin{aligned} \text{bel}(x_1 = \text{open}) &= 0.36/0.44 = 0.818, \\ \text{bel}(x_1 = \text{closed}) &= 0.08/0.44 = 0.181. \end{aligned}$$

A python implementation of this discrete Bayes filter is shown in Listing 9.3. This creates a class called `DoorBelief`. To use it, first we import the class from the module, then create an instance of it.

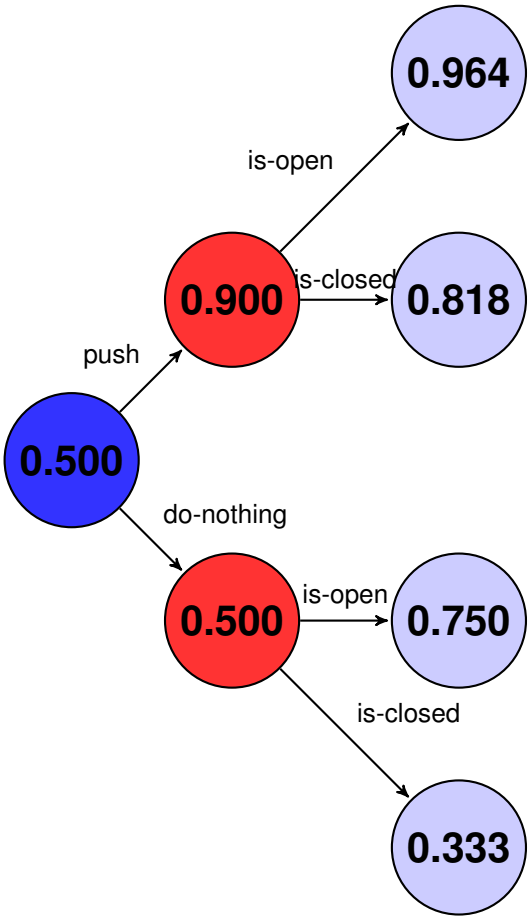


Figure 9.3: Beliefs that the door is open. The dark blue node is the previous belief, the red nodes are the predicted beliefs (given a control action), and light blue nodes are the updated beliefs (given a sensor measurement).

```
>>> from bayes_demo3 import DoorBelief
>>> door = DoorBelief()
>>> print(door)
Open 0.500  Closed 0.500
```

Note, since the robot has no idea of the state of the door, the open and closed beliefs are both 0.5.

Let's now update the beliefs by including a sensor measurement that the door is closed:

```
>>> door.update('do-nothing', 'is-closed')
>>> print(door)
Open 0.333  Closed 0.667
```

Now, the robot thinks that the door is more likely to be closed but it does not fully trust the sensor measurement. As we include more sensor measurements that the door is closed, the robot comes to believe that the door is more likely to be closed:

```
>>> door.update('do-nothing', 'is-closed')
>>> print(door)
Open 0.200  Closed 0.800
>>> door.update('do-nothing', 'is-closed')
>>> print(door)
Open 0.111  Closed 0.889
>>> door.update('do-nothing', 'is-closed')
>>> print(door)
Open 0.059  Closed 0.941
```

This sequence of events is shown in Figure 9.4.

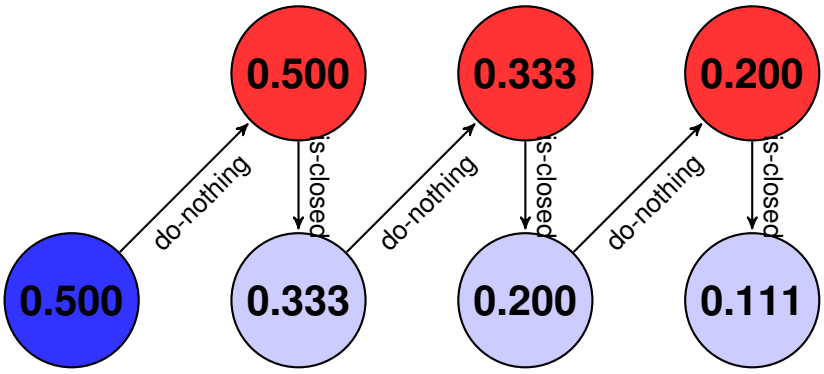


Figure 9.4: Beliefs that the door is open. The dark blue node is the initial belief, the red nodes are the predicted beliefs (given a control action), and light blue nodes are the updated beliefs (given a sensor measurement).

Now, let's consider the case where the robot tries to open the door by pushing on it (see Figure 9.5). In this case, it appears that the push has failed since the sensor keeps saying that it is closed. The robot starts assuming that it has opened it but as the sensor reports that it is closed, the robot becomes less sure.

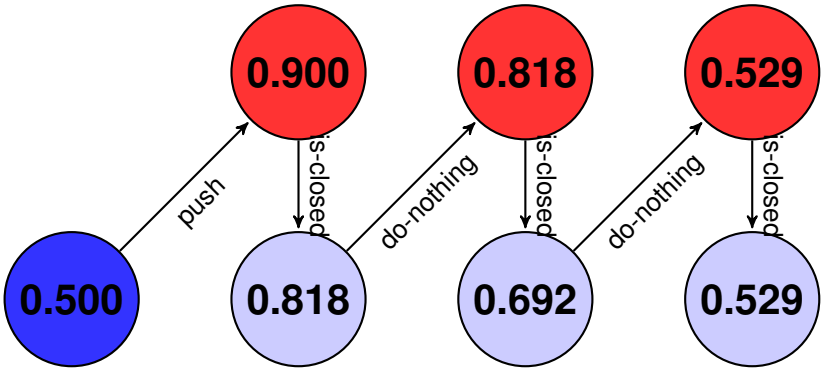


Figure 9.5: Beliefs that the door is open. The dark blue node is the initial belief, the red nodes are the predicted beliefs (given a control action), and light blue nodes are the updated beliefs (given a sensor measurement).

3 *Bayes filter for continuous states*

Bayes filters are straightforward when the states are discrete. But what about continuous densities, such as the distance a robot has moved? One approach[5], is to approximate the continuous posterior with a number of discrete states. The more states the better the approximation but at the expense of memory and CPU time. An example is shown in Figure 9.6 that tries to estimate the x-position of a robot given noisy sensor measurements. The initial beliefs are equiprobable but as more sensor measurements are made, the posterior becomes more tightly distributed around the best belief.

[5] Called a histogram filter.

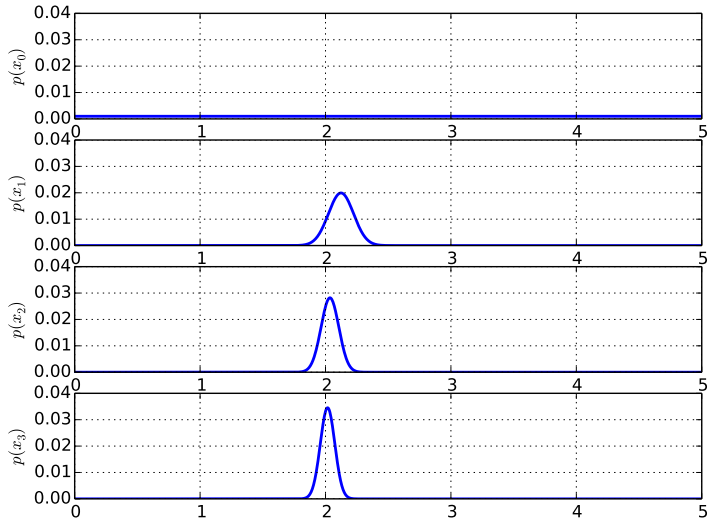


Figure 9.6: Plots of the posterior density for a position estimator using a discretisation of the continuous position. Each subplot represents the next time step. Note, the posterior becomes more tightly distributed since the robot is stationary.

4 Exercises

1. If a discrete Bayes filter stores a belief as a floating point number, how many floating point numbers are required to store the beliefs for three state variables, each with 1000 states?

2. If a discrete Bayes filter stores a belief as a floating point number, how many floating point numbers are required to store the beliefs for 100 state variables, each with 200 states?

3. If a discrete Bayes filter has 5 state variables, 100 belief states for each state variable, 100 sensor measurement states, and 10 control states, what are the required sizes of the state transition and measurement conditional probability matrices (hint see Listing 9.4)?

4. If a robot door sensor senses an open door with 60% probability and a closed door with 80% probability, what is the probability of the door being open if the sensor reports: is-closed, is-open, is-closed? Assume that the robot cannot manipulate the door, the robot does not expect that the door can change state, and that the initial belief for an open door is 0.5.

5. If a robot door sensor senses an open door with 80% probability and a closed door with 60% probability, what is the probability of the door being open if the sensor reports: is-closed, is-open, is-closed? Assume that the robot cannot manipulate the door, the robot does not expect that the door can change state, and that the initial belief for an open door is 0.7.

```
from numpy import zeros_like

class DiscreteBayesFilter(object):

    def __init__(self, beliefs0):
        """Create Discrete Bayes Filter with initial beliefs, beliefs0
           """
        self.beliefs = beliefs0


    def predict(self, u, state_transition_pc):
        """Calculate a priori beliefs"""

        predicted_beliefs = zeros_like(self.beliefs)

        # Iterate over the states
        for m, _ in enumerate(self.beliefs):

            for n, belief in enumerate(self.beliefs):
                predicted_beliefs[m] += state_transition_pc[m, u, n] *
                    belief

        self.beliefs = predicted_beliefs


    def update(self, z, sensor_pc):
        """Calculate a posteriori beliefs"""

        # Update beliefs
        for m, belief in enumerate(self.beliefs):
            self.beliefs[m] = sensor_pc[z, m] * belief

        # Normalise so beliefs sum to 1
        self.beliefs = self.beliefs / self.beliefs.sum()


    def estimate(self, u, z, state_transition_pc, sensor_pc):
        """Estimate a posteriori beliefs"""

        self.predict(u, state_transition_pc)
        self.update(z, sensor_pc)
```

Listing 9.1: Python implementation of a discrete Bayes filter using control and sensor information.

```
"""Bayes filtering example based on the simple example from Thrun,
Burgard, and Fox, "Probabilistic Robots", p 28."""
from __future__ import division
import numpy as np
from discrete_bayes_filter import DiscreteBayesFilter

class DoorBelief(object):

    belief_states = ['open', 'closed']
    num_belief_states = len(belief_states)
    open, closed = range(num_belief_states)

    control_states = ['do_nothing']
    num_control_states = len(control_states)
    do_nothing = range(num_control_states)

    sensor_states = ['is_open', 'is_closed']
    num_sensor_states = len(sensor_states)
    is_open, is_closed = range(num_sensor_states)

    # Conditional probabilities to characterise sensor noise
    # p(z_t | x_t)
    sensor_pc = np.zeros((num_sensor_states, num_belief_states))
    sensor_pc[is_open, open] = 0.6
    sensor_pc[is_closed, open] = 0.4
    sensor_pc[is_open, closed] = 0.2
    sensor_pc[is_closed, closed] = 0.8

    # Conditional probabilities for state transitions of door
    # p(x_t | u_t, x_{t-1})
    state_pc = np.zeros((num_belief_states, num_control_states,
       num_belief_states))
    state_pc[open, do_nothing, open] = 1
    state_pc[closed, do_nothing, open] = 0
    state_pc[open, do_nothing, closed] = 0
    state_pc[closed, do_nothing, closed] = 1

    def __init__(self):

        self.dbf = DiscreteBayesFilter(np.ones(self.num_belief_states) /
            self.num_belief_states)


    def __str__(self):

        beliefs = self.dbf.beliefs

        return 'Open %.3f  Closed %.3f' % (beliefs[self.open], beliefs[
           self.closed])


    def bayes_filter(self, u, z):

        self.dbf.estimate(self.do_nothing, z, self.state_pc, self.
           sensor_pc)
```

Listing 9.2: Python example of using a discrete Bayes filter using only sensor information.

```
from discrete_bayes_filter_demo3 import DoorBelief

door = DoorBelief()
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)
```

Listing 9.3: Python example of using a discrete Bayes filter using only sensor information.

```
"""Bayes filtering example based on the simple example from Thrun,
Burgard, and Fox, "Probabilistic Robots", p 28."""
from __future__ import division
import numpy as np
from discrete_bayes_filter import DiscreteBayesFilter

class DoorBelief(object):

    belief_states = ['open', 'closed']
    num_belief_states = len(belief_states)
    open, closed = range(num_belief_states)

    control_states = ['push', 'close', 'do_nothing']
    num_control_states = len(control_states)
    push, close, do_nothing = range(num_control_states)

    sensor_states = ['is_open', 'is_closed']
    num_sensor_states = len(sensor_states)
    is_open, is_closed = range(num_sensor_states)

    # Conditional probabilities to characterise sensor noise
    # p(z_t | x_t)
    sensor_pc = np.zeros((num_sensor_states, num_belief_states))
    sensor_pc[is_open, open] = 0.6
    sensor_pc[is_closed, open] = 0.4
    sensor_pc[is_open, closed] = 0.2
    sensor_pc[is_closed, closed] = 0.8

    # Conditional probabilities for state transitions of door
    # p(x_t | u_t, x_{t-1})
    state_pc = np.zeros((num_belief_states, num_control_states,
       num_belief_states))
    state_pc[open, push, open] = 1
    state_pc[closed, push, open] = 0
    state_pc[open, push, closed] = 0.8
    state_pc[closed, push, closed] = 0.2
    state_pc[open, do_nothing, open] = 1
    state_pc[closed, do_nothing, open] = 0
    state_pc[open, do_nothing, closed] = 0
    state_pc[closed, do_nothing, closed] = 1
    state_pc[open, close, open] = 0.2
    state_pc[closed, close, open] = 0.8
    state_pc[open, close, closed] = 0
    state_pc[closed, close, closed] = 1

    def __init__(self):

        self.dbf = DiscreteBayesFilter(np.ones(self.num_belief_states) /
            self.num_belief_states)


    def __str__(self):

        beliefs = self.dbf.beliefs

        return 'Open %.3f  Closed %.3f' % (beliefs[self.open], beliefs[
           self.closed])


    def bayes_filter(self, u, z):

        self.dbf.estimate(u, z, self.state_pc, self.sensor_pc)
```

Listing 9.4: Python example of using a discrete Bayes

```
from discrete_bayes_filter_demo4 import DoorBelief

door = DoorBelief()
print(door)

door.bayes_filter(door.push, door.is_closed)
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)

door.bayes_filter(door.do_nothing, door.is_closed)
print(door)
```

Listing 9.5: Python example of using a discrete Bayes filter using control and sensor information.

10
Gaussian Bayes filters

Discretisation of continuous state variables for Bayes filtering is memory and CPU intensive. An alternative approach is to parameterise the beliefs and to store the moments of the posterior (mean, variance, etc.). If the posterior has a single mode, the most common choice of probability distribution is the Gaussian distribution. This can be parameterised by its mean and variance.

There are many Gaussian Bayes filters. They all use recursive state estimation. The simplest is the Kalman filter but it only works for a linear system. There are two principle Kalman filters for non-linear systems, the extended Kalman filter (EKF) and the unscented Kalman filter (UKF).

Gaussian Bayes filters model the belief distribution of a random variable, X, with a Gaussian pdf. This distribution has two advantages:

1. It can be parameterised by two parameters, the mean, μ_X, and variance, σ_X^2. A multivariate Gaussian distribution can be modelled by a vector of means and a covariance matrix.
2. A linear mapping of a Gaussian random variable is also a Gaussian random variable but with a possibly different mean and variance, see Figure 10.1.

1 The Kalman filter revisited

The Kalman filter is a special case of a Bayes filter for a linear system with additive Gaussian noise and a Gaussian belief.

Consider a linear system with a state equation,

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t. \tag{10.1}$$

If the process noise, $\mathbf{w}_t$, has a Gaussian distribution, then so does the state transition conditional probability

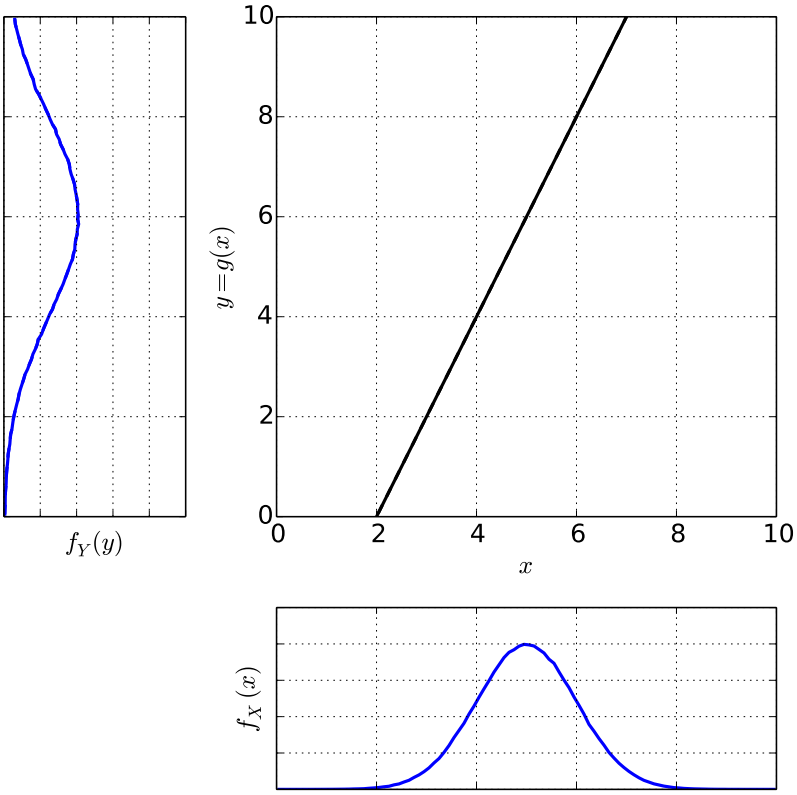


Figure 10.1: Error propagation with a linear function, $y = g(x) = 2x - 4$.

density. Thus, the (*a priori*) mean and covariance of the estimated state are

$$\mathrm{E}\left\{\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{u}_t\right\} = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t = \mu_{\mathbf{x}_t}, \tag{10.2}$$

$$\mathrm{E}\left\{\left(\mathbf{x}_t - \mu_{\mathbf{x}_t}\right)\left(\mathbf{x}_t - \mathbf{x}_t\right)^{\mathrm{T}} | \mathbf{x}_{t-1}, \mathbf{u}_t\right\} = \mathbf{\Sigma}_w, \tag{10.3}$$

and so the state transition conditional probability density is

$$p(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{u}_t) = \eta \exp\left(-\frac{1}{2}\left(\mathbf{x}_t - \mu_{\mathbf{x}_t}\right)^{\mathrm{T}} \mathbf{\Sigma}_w^{-1} \left(\mathbf{x}_t - \mu_{\mathbf{x}_t}\right)\right), \tag{10.4}$$

where

$$\eta = \frac{1}{\sqrt{(2\pi)^N \det \mathbf{\Sigma}_w}}. \tag{10.5}$$

Similarly, with a linear output equation,

$$\mathbf{z}_t = \mathbf{C}\mathbf{x}_t + \mathbf{v}_t, \tag{10.6}$$

Gaussian sensor noise, $\mathbf{v}_t$, results in a Gaussian measurement likelihood,

$$p(\mathbf{z}_t | \mathbf{x}_t) = \eta \exp\left(-\frac{1}{2}\left(\mathbf{z}_t - \mu_{\mathbf{z}_t}\right)^{\mathrm{T}} \mathbf{\Sigma}_v^{-1} \left(\mathbf{z}_t - \mu_{\mathbf{z}_t}\right)\right), \tag{10.7}$$

where

$$\mathrm{E}\left\{\mathbf{z}_t|\mathbf{z}_{t-1},\mathbf{u}_t\right\} = \mathbf{C}\mathbf{x}_t = \mu_{\mathbf{z}_t}, \tag{10.8}$$

$$\mathrm{E}\left\{\left(\mathbf{z}_t-\mu_{\mathbf{z}_t}\right)\left(\mathbf{z}_t-\mu_{\mathbf{z}_t}\right)^{\mathrm{T}}|\mathbf{z}_{t-1},\mathbf{u}_t\right\} = \mathbf{\Sigma}_v, \tag{10.9}$$

and

$$\eta = \frac{1}{\sqrt{(2\pi)^N \det \mathbf{\Sigma}_v}}. \tag{10.10}$$

2 *Gaussian filters for non-linear systems*

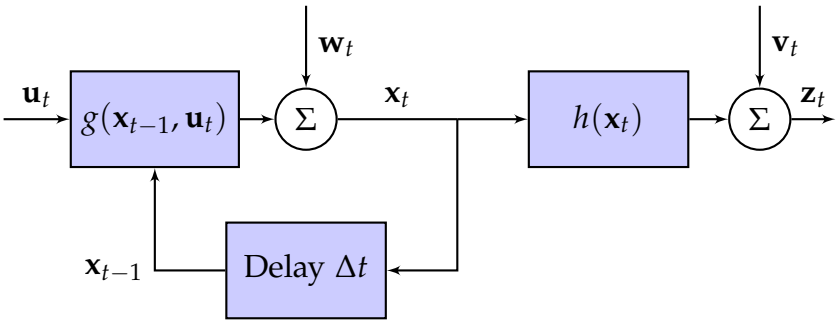


Figure 10.2: Discrete non-linear state-space system block diagram.

The equivalent state-space equations for a non-linear system (see Figure 10.2) are:

$$\mathbf{x}_t = g(\mathbf{x}_{t-1},\mathbf{u}_t) + \mathbf{w}_t, \tag{10.11}$$

$$\mathbf{z}_t = h(\mathbf{x}_t,\mathbf{u}_t) + \mathbf{v}_t. \tag{10.12}$$

A Kalman filter is not applicable to such a system since the mapping of a Gaussian random variable by a non-linear function produces a random variable with a non-Gaussian distribution (see Figure 10.3). In general, the resulting distribution cannot be parameterised by a mean vector and a covariance matrix.

2.1 *Extended Kalman filter (EKF)*

The EKF linearises the non-linear state-space equations (10.12) using the Jacobians of g and h, calculated at the state estimate, $\hat{\mathbf{x}}_t$, and control, $\mathbf{u}_t$,

$$\mathbf{A}_t = \left.\frac{\partial g}{\partial \mathbf{x}_t}\right|_{\hat{\mathbf{x}}_t,\mathbf{u}_t}, \tag{10.13}$$

$$\mathbf{B}_t = \left.\frac{\partial g}{\partial \mathbf{u}_t}\right|_{\hat{\mathbf{x}}_t,\mathbf{u}_t}, \tag{10.14}$$

$$\mathbf{C}_t = \left.\frac{\partial h}{\partial \mathbf{x}_t}\right|_{\hat{\mathbf{x}}_t,\mathbf{u}_t}, \tag{10.15}$$

$$\mathbf{D}_t = \left.\frac{\partial h}{\partial \mathbf{u}_t}\right|_{\hat{\mathbf{x}}_t,\mathbf{u}_t}. \tag{10.16}$$

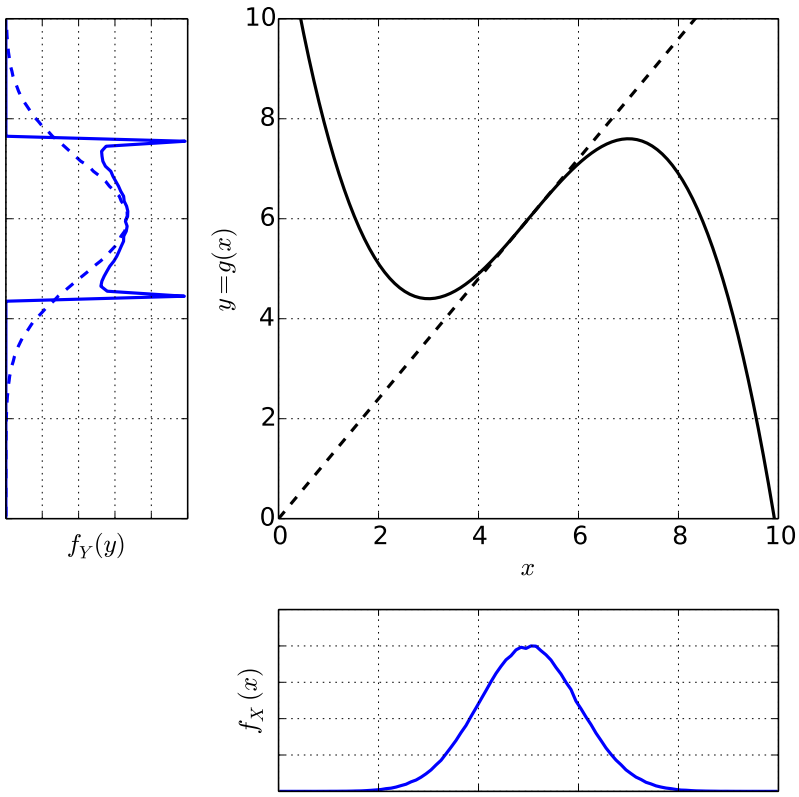


Figure 10.3: Error propagation with a non-linear function, $y = g(x) = 1.2x - 0.1(x-5)^3$. The dashed line is a linear approximation around $x = 5$.

2.2 *Unscented Kalman filter (UKF)*

The extended Kalman filter gives poor performance for highly non-linear systems. It also requires explicit calculation of derivatives[1].

As an alternative, the unscented Kalman filter (UKF) models the Gaussian distribution by a set of *sigma points* around the mean. These sigma points are then mapped through the non-linear system to characterise the resulting distribution, again approximated as a Gaussian.

[1] Jacobians for multidimensional systems.

3 *Multiple hypothesis tracking*

A Gaussian pdf has a single mode and thus can only track a single hypothesis. For multiple hypotheses, a mixture (weighted sum) of Gaussians can be employed; one for each hypothesis. The resultant PDF is

$$f_X(x) = \sum_i \frac{w_i}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu_i}{\sigma_i}\right)^2\right). \qquad (10.17)$$

An example is shown in Figure 10.4.

Using a mixture of Gaussians, a multiple hypothesis Kalman filter (MHKF) or a multiple hypothesis extended Kalman filter (MHEKF) can be created.

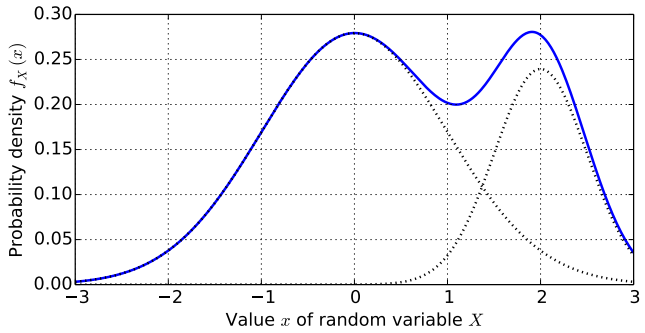


Figure 10.4: A weighted sum of two Gaussian distributions: $\mu_1 = 0$, $\sigma_1^2 = 1$, $w_1 = 0.3$, $\mu_2 = 2$, $\sigma_2^2 = 0.5$, $w_2 = 0.7$.

4 *Information filters*

There are two ways to represent multivariate Gaussian distributions:

moments parameterisation mean vector and covariance matrix:

$$f_{\mathbf{X}}(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^N \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})\right), \tag{10.18}$$

where $\boldsymbol{\Sigma}$ is the covariance matrix and $\boldsymbol{\mu} = \mathrm{E}\{\mathbf{x}\}$ is the mean of $\mathbf{x}$.

canonical parameterisation information vector and information matrix:

$$f_{\mathbf{X}}(\mathbf{x}) = \frac{\exp\left(-\frac{1}{2}\boldsymbol{\nu}^{\mathrm{T}}\boldsymbol{\Lambda}^{-1}\boldsymbol{\nu}\right)}{\sqrt{(2\pi)^N \det(\boldsymbol{\Lambda}^{-1})}} \exp\left(-\frac{1}{2}\mathbf{x}^{\mathrm{T}}\boldsymbol{\Lambda}\mathbf{x} + \mathbf{x}^{\mathrm{T}}\boldsymbol{\nu}\right), \tag{10.19}$$

where $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$ is the (Fisher) information matrix and $\boldsymbol{\nu} = \boldsymbol{\Lambda}\boldsymbol{\mu}$ is the information vector (scaled mean).

The Kalman filter is derived when applying a Bayes filter with the moments parameterisation. The information filter is derived when applying a Bayes filter with the canonical parameterisation of a Gaussian. They can be considered duals of each other.

5 *The Kalman filter family*

Here are some variations of the Kalman filter (see Figure 10.5):

Kalman filter (KF) Uses Gaussian pdf assumption and linear system model.

Extended Kalman filter (EKF) Uses Gaussian pdf assumption and linear approximation of a non-linear system model (using Taylor series).

Unscented Kalman filter (UKF) Another Kalman filter for non-linear systems but with a different linear approximation (a stochastic linearisation).

Information filter (IF) This is like a Kalman filter but uses a canonical parameterisation of a Gaussian[2]). This can simplify the update equations[3].

Extended information filter (EIF) This is similar to an extended Kalman filter but using an information filter.

Multi-hypothesis extended Kalman filter (MHEKF) This represents posteriors with a mixture (or sums) of Gaussians to represent multiple modes.

[2] Instead of a mean vector and covariance matrix an information vector and information matrix is used. The (Fisher) information matrix is the inverse of the covariance matrix and the information vector is the product of the inverse covariance matrix and mean vector.

[3] Global uncertainty is simply a zero information matrix.

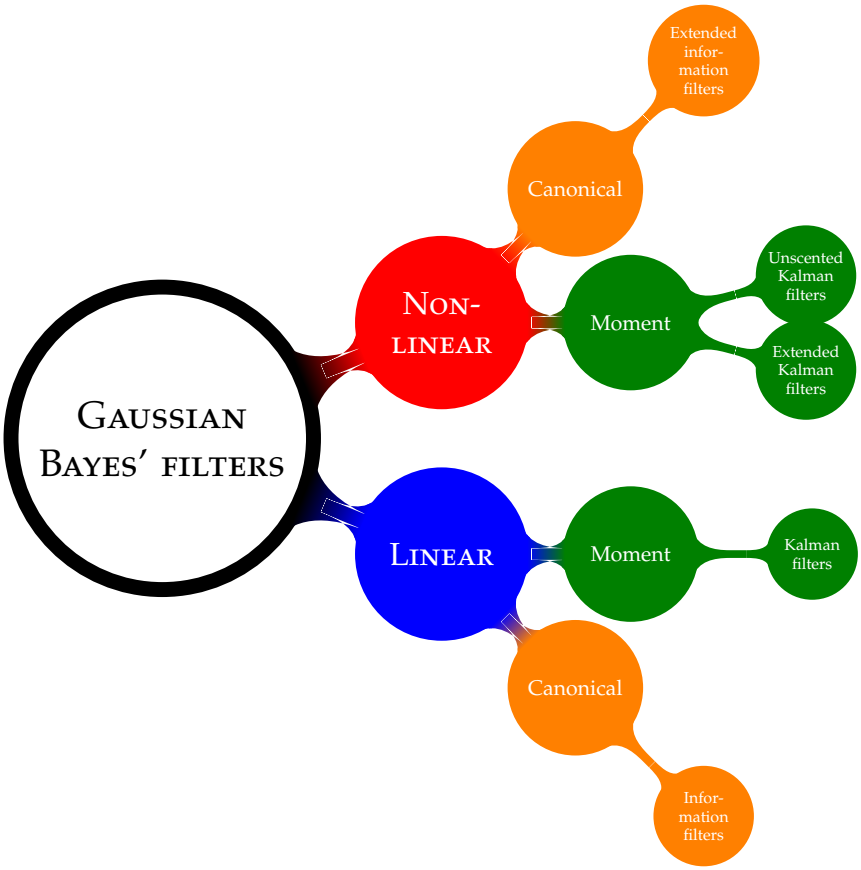


Figure 10.5: Categories of Gaussian Bayes' filters.

6 Bayes filters to Kalman filters

A Bayes filter represents a model of a discrete-time continuous system by the *state transition probability* density, $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$, and the *measurement likelihood*, $p(\mathbf{z}_t|\mathbf{x}_t)$. Its belief of the system state is represented by an arbitrary probability density, $\text{bel}(\mathbf{x}_t) = p(\mathbf{x}_t|\mathbf{u}_{0:t}, \mathbf{z}_{0:t})$.

A simpler, but less general, probabilistic system model assumes zero-mean additive Gaussian[4] process and measurement noise:

[4] This assumption is not required for particle filters.

$$\mathbf{x}_t = g(\mathbf{x}_{t-1}, \mathbf{u}_t) + \mathbf{w}_t, \tag{10.20}$$
$$\mathbf{z}_t = h(\mathbf{x}_t, \mathbf{u}_t) + \mathbf{v}_t, \tag{10.21}$$

so the state transition probability density is

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) = \eta \exp\left(-\frac{1}{2}\left(\mathbf{x}_t - \mu_{\mathbf{x}_t}\right)^{\mathrm{T}} \boldsymbol{\Sigma}_w^{-1} \left(\mathbf{x}_t - \mu_{\mathbf{x}_t}\right)\right), \tag{10.22}$$

where

$$\mu_{\mathbf{x}_t} = g(\mathbf{x}_{t-1}, \mathbf{u}_t), \tag{10.23}$$

and the measurement likelihood is

$$p(\mathbf{z}_t|\mathbf{x}_t) = \eta \exp\left(-\frac{1}{2}\left(\mathbf{z}_t - \mu_{\mathbf{z}_t}\right)^{\mathrm{T}} \boldsymbol{\Sigma}_v^{-1} \left(\mathbf{z}_t - \mu_{\mathbf{z}_t}\right)\right), \tag{10.24}$$

where

$$\mu_{\mathbf{z}_t} = h(\mathbf{x}_t, \mathbf{u}_t). \tag{10.25}$$

In both (10.22) and (10.24), η is a normalising constant.

Both the extended and unscented Kalman filters use this approximation and, in addition, represent the belief by a Gaussian density[5]. However, in general, a Gaussian belief density is a poor approximation for a non-linear system.

[5] This only requires a mean vector and covariance matrix to parameterise.

The Kalman filter requires a linear model of the system, so

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t, \tag{10.26}$$
$$\mathbf{z}_t = \mathbf{C}\mathbf{x}_t + \mathbf{D}\mathbf{u}_t + \mathbf{v}_t. \tag{10.27}$$

With a linear model, a Gaussian belief always maps to another Gaussian belief at the next time step.

7 Linear mapping of a random variable

Consider the linear mapping between a random variable X to another random variable Y, where

$$Y = aX + b. \tag{10.28}$$

Here a is a scale factor and b is an offset. The mean of Y is related to the mean of X by

$$\begin{aligned} \mu_Y &= \mathrm{E}\{Y\}, && (10.29)\\ &= \mathrm{E}\{aX\} + b, && (10.30)\\ &= a\,\mathrm{E}\{X\} + b, && (10.31)\\ &= a\mu_X + b. && (10.32) \end{aligned}$$

The variance of Y is related[6] to the variance of X by

$$\begin{aligned} \sigma_Y^2 &= \mathrm{Var}\{Y\}, && (10.33)\\ &= \mathrm{E}\left\{(Y-\mu_Y)^2\right\}, && (10.34)\\ &= \mathrm{E}\left\{(aX - b - (a\mu_X + b))^2\right\}, && (10.35)\\ &= a^2\,\mathrm{E}\left\{(X-\mu_X)^2\right\}, && (10.36)\\ &= a^2\sigma_X^2. && (10.37) \end{aligned}$$

[6] This relation is called the propagation of uncertainty.

If $f_X(x)$ is Gaussian[7] then it can be characterised by the mean and variance and so with a linear mapping, $f_Y(y)$ will also be Gaussian. However, in general, there is no simple mapping of the pdf, $f_Y(y)$, given the pdf, $f_X(x)$.

[7] With other distributions, the mapping of higher order statistics are required.

Consider the linear mapping between a random vector[8] $\mathbf{X}$ to another random vector $\mathbf{Y}$

$$\mathbf{Y} = \mathbf{A}\mathbf{X} + \mathbf{b}, \tag{10.38}$$

[8] To avoid confusion between vectors and matrices, lower case is often used to denote a random variable vector.

where $\mathbf{A}$ is a transfer matrix and $\mathbf{b}$ is an offset vector. The mean of $\mathbf{Y}$ is related to the mean of $\mathbf{X}$ by

$$\mu_{\mathbf{Y}} = \mathbf{A}\mu_{\mathbf{X}} + \mathbf{b}. \tag{10.39}$$

The covariance[9] of $\mathbf{Y}$ is related to the covariance of $\mathbf{X}$ by

$$\mathbf{\Sigma}_{\mathbf{Y}} = \mathbf{A}\,\mathbf{\Sigma}_{\mathbf{X}}\,\mathbf{A}^{\mathrm{T}}. \tag{10.40}$$

[9] A covariance matrix has variances on the diagonal and correlations off the diagonal.

Even if the errors of $\mathbf{X}$ are uncorrelated so that $\mathbf{\Sigma}_{\mathbf{X}}$ is diagonal, then in general the covariance $\mathbf{\Sigma}_{\mathbf{Y}}$ is full.

8 Non-linear mapping of a random variable

Consider a non-linear mapping of a Gaussian random variable X,

$$Y = g(X). \tag{10.41}$$

An approximate approach to characterise the distribution of Y is to linearise the mapping, $g(x)$, of a value x of X around the mean, μ_X, of X,

$$Y \approx \frac{\mathrm{d}}{\mathrm{d}X} g(X)\Big|_{X=\mu_X} (X - \mu_X) + g(\mu_X). \tag{10.42}$$

When the system is non-linear and multidimensional,

$$\mathbf{Y} = g(\mathbf{X}), \tag{10.43}$$

then a linear approximation can be employed around the mean, $\boldsymbol{\mu}_{\mathbf{X}}$,

$$\mathbf{Y} \approx \mathbf{J}\,(\mathbf{X} - \boldsymbol{\mu}_{\mathbf{X}}) + g(\boldsymbol{\mu}_{\mathbf{X}}), \tag{10.44}$$

where $\mathbf{J}$ is a Jacobian matrix (evaluated at $\mathbf{X} = \boldsymbol{\mu}_{\mathbf{X}}$),

$$\mathbf{J} = \begin{bmatrix} \frac{\partial Y_1}{\partial X_1} & \cdots & \frac{\partial Y_1}{\partial X_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial Y_m}{\partial X_1} & \cdots & \frac{\partial Y_m}{\partial X_n} \end{bmatrix}. \tag{10.45}$$

With this approximation, the covariance of $\mathbf{Y}$ is related to the covariance of $\mathbf{X}$ by

$$\boldsymbol{\Sigma}_{\mathbf{Y}} = \mathbf{J}\,\boldsymbol{\Sigma}_{\mathbf{X}}\,\mathbf{J}^{\mathrm{T}}. \tag{10.46}$$

9 Derivation of Kalman filter as special case of a Bayes filter

In the following derivation, a single state variable, x_t, with a single control action, u_t, and single measurement, z_t is assumed.

The initial Gaussian belief distribution is

$$\mathrm{bel}(x_0) = \frac{1}{\sqrt{2\pi\sigma_{\hat{x}_0}^2}} \exp\left(-0.5\frac{(x_0 - \hat{x}_0)^2}{\sigma_{\hat{x}_0}^2}\right), \tag{10.47}$$

where $\hat{x}_0$ is the initial guess with variance $\hat{x}_0$.

Using the state transition conditional probability distribution, $p(x_1|x_0, u_1)$, the predicted belief distribution is

$$\breve{\text{bel}}(x_1) = \int p(x_1|x_0, u_1)\,\text{bel}(x_0)\text{d}x_0. \tag{10.48}$$

With additive Gaussian process noise, the state transition can be described by a function g,

$$p(x_1|x_0, u_1) = \frac{1}{\sqrt{2\pi\sigma_w^2}} \exp\left(-0.5\frac{(x_1 - g(x_0, u_1))^2}{\sigma_w^2}\right). \tag{10.49}$$

Assuming a linear system,

$$g(x_0, u_1) = Ax_0 + Bu_1, \tag{10.50}$$

then

$$p(x_1|x_0, u_1) = \frac{1}{\sqrt{2\pi\sigma_w^2}} \exp\left(-0.5\frac{(x_1 - Ax_0 - Bu_1)^2}{\sigma_w^2}\right). \tag{10.51}$$

Thus (10.57) can be expressed as

$$\breve{\text{bel}}(x_1) = \int \frac{1}{\sqrt{2\pi\sigma_w^2}} \exp\left(-0.5\frac{(x_1 - Ax_0 - Bu_1)^2}{\sigma_w^2}\right) \frac{1}{\sqrt{2\pi\sigma_{\hat{x}_0}^2}} \exp\left(-0.5\frac{(x_0 - \hat{x}_0)^2}{\sigma_{\hat{x}_0}^2}\right) \text{d}x_0. \tag{10.52}$$

Solving the integral gives

$$\breve{\text{bel}}(x_1) = \frac{\sqrt{2\pi\sigma_w^2\sigma_{\hat{x}_0}^2}}{\sigma_{\breve{x}_1}^2} \exp\left(-0.5\frac{(x_1 - \breve{x}_1)^2}{\sigma_{\breve{x}_1}^2}\right), \tag{10.53}$$

where

$$\breve{x}_1 = A\hat{x}_0 + Bu_1, \tag{10.54}$$

$$\sigma_{\breve{x}_1}^2 = A^2\sigma_{\hat{x}_0}^2 + \sigma_w^2. \tag{10.55}$$

Using Bayes' theorem, the posteriori belief distribution is

$$\text{bel}(x_1) = \eta p(z_1|x_1)\breve{\text{bel}}(x_1). \tag{10.56}$$

Assuming additive Gaussian sensor noise, the measurement likelihood is

$$p(z_1|x_0) = \frac{1}{\sqrt{2\pi\sigma_v^2}} \exp\left(-0.5\frac{(z_1 - h(x_0))^2}{\sigma_v^2}\right). \tag{10.57}$$

Assuming a linear system,

$$h(x_0) = Cx_0, \tag{10.58}$$

then

$$p(z_1|x_1) = \frac{1}{\sqrt{2\pi\sigma_v^2}} \exp\left(-0.5\frac{(z_1 - Cx_1)^2}{\sigma_v^2}\right). \quad (10.59)$$

Using this, the posterior belief distribution is given by

$$\text{bel}(x_1) = \eta \frac{1}{\sqrt{2\pi\sigma_v^2}} \exp\left(-0.5\frac{(z_1 - Cx_1)^2}{\sigma_v^2}\right) \frac{\sqrt{2\pi\sigma_w^2\sigma_{\hat{x}_0}^2}}{\sigma_{\check{x}_1}^2} \exp\left(-0.5\frac{(x_1 - \check{x}_1)^2}{\sigma_{\check{x}_1}^2}\right), \quad (10.60)$$

or

$$\text{bel}(x_1) = \eta' \exp\left(-0.5\frac{(x_1 - \hat{x}_1)^2}{\sigma_{\hat{x}_1}^2}\right), \quad (10.61)$$

where

$$\hat{x}_1 = \check{x}_1 + \frac{C\sigma_{\check{x}_1}^2 (z_1 - C\check{x}_1)}{C^2\sigma_{\check{x}_1}^2 + \sigma_v^2}, \quad (10.62)$$

$$\sigma_{\hat{x}_1}^2 = \frac{\sigma_v^2\sigma_{\check{x}_1}^2}{C^2\sigma_{\check{x}_1}^2 + \sigma_v^2}, \quad (10.63)$$

$$\eta' = \frac{1}{\sqrt{2\pi\sigma_{\hat{x}_1}^2}}. \quad (10.64)$$

This result is the same as that given by (6.12).

10 Summary

- Kalman filters are a class of Bayes filter. They are recursive and thus require the complete state (Markov) assumption.
- Kalman filters track only the mean and covariance of the estimated state and thus assume a Gaussian pdf.
- When the system is non-linear, the resulting pdf is no longer Gaussian.
- A non-linear system can be linearised around the mean with a Taylor series; this is the basis of the extended Kalman filter.
- The unscented Kalman filter uses a different linearisation based on sigma points.
- A Gaussian distribution can be parameterised in terms of an information vector and matrix; this leads to the information filters.
- Multiple hypotheses can be tracked using a weighted sum of Gaussians (one per hypothesis).

11 Exercises

1. A random variable X is mapped to a variable Y by $Y = 3X + 2$. If the mean of X is 1 and the standard-deviation is 2, determine the mean and standard deviation of Y. Are any assumptions required?

2. A random variable X is mapped to a random variable Y by $Y = X^2 + 2$. If the mean of X is 1 and the standard-deviation is 2, determine the mean and standard deviation of Y. Are any assumptions required?

3. A Gaussian random variable X with mean 2 and variance 3 is mapped to a random variable Y by $Y = 2X - 1$. What is the pdf of Y?

4. A Gaussian random variable X with mean 2 and variance 3 is mapped to a random variable Y by $Y = 2X^2 - 1$. Is the pdf of Y Gaussian?

5. A simple robot has the state transition equation,

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} v\Delta t \cos\theta_t, \\ v\Delta t \sin\theta_t, \\ 0 \end{bmatrix}, \tag{10.65}$$

   where x and y denote the Robot's location and θ denotes its heading. If the measurement vector is a linear function of the robot's state, what sort of Kalman filter would you recommend to estimate the robot's state?

6. If for the previous example the measurement vector is a non-linear function of the robot's state, what sort of Kalman filter would you recommend to estimate the robot's state?

7. How many parameters are required to describe a bivariate Gaussian distribution?

8. How many parameters are required to describe a trivariate Gaussian distribution?

9. If there are 1000 state variables, how many parameters does an extended Kalman filter need to track?

11
Non-parametric Bayes filters

An alternative to Gaussian techniques for Bayes filters are nonparametric filters. They approximate the posterior by a finite number of values. There are two common nonparametric filters:

Histogram filter: This approximates the posterior with a histogram and so is equivalent to a discrete Bayes filter.

Particle filter: This uses importance sampling. The particles are spaced more closely near modes in the posterior distribution.

1 Particle filters

Imagine that you are lost in the pitch black countryside. All you can see are the lights of farmhouses[1] in the distance. You have a phone but no GPS or map. So how can you find out where you are?

[1] Did I say that all the farmers had loaded shotguns and were suspicious of strangers?

One approach is to phone all your friends and relations and tell them to spread themselves randomly around the countryside at random angles. You then phone them up and tell them the approximate distance and bearing to the nearest farmhouse you can see. They then perform the same exercise but since they have a map and GPS they can accurately determine the correct range and distance to the nearest farmhouse they can see. Using the information you gave them and their own measurement, they can assign a weight to the consistency of the measurements. If the weight is small then they are in the wrong location or facing in the wrong direction. If the weight is high, then they could have the same pose as you (or the same pose with respect to another farmhouse). You then tell the people whose measurements have a low weight to hurry to a new place

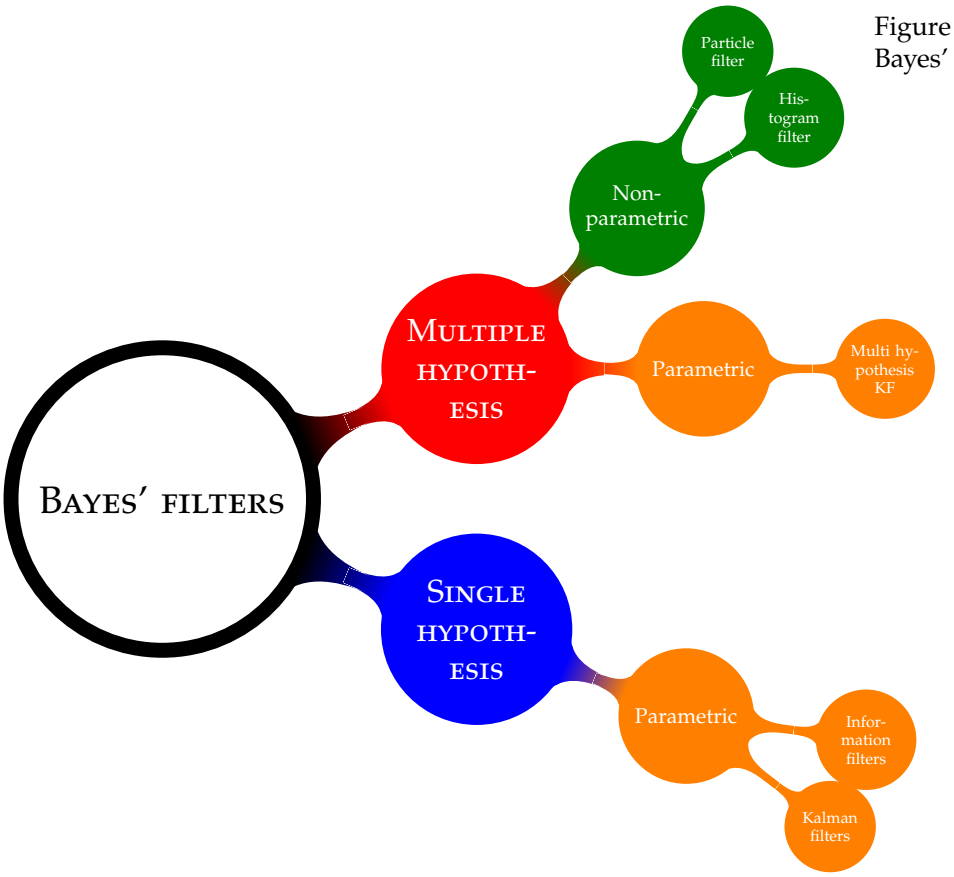


Figure 11.1: Categories of Bayes' filters.

in the countryside, preferably closer to the people with high weights. You then walk along for a while and tell everyone to do the same. After a while, you stop, estimate the range and bearing to the nearest farmhouse, phone you friends and get them to reestimate their weights, etc. The process then repeats until hopefully you find all your friends and relatives huddled around you.

This algorithm is analogous to a particle filter where each of your friends and relatives is a particle. Since they have no prior knowledge of where you are, they need to spread themselves out uniformly over the countryside—this models the initial belief that you could be anywhere. As you wander along they wander themselves at a similar speed. Then using the measurements you give them, they collectively reestimate the posterior density for their belief of your location[2].

[2] And orientation.

1.1 *Particle filter algorithm*

1. Generate initial distribution of particles, each with unit weight.
2. Update each particle using transition model $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$.
3. Weight each particle using evidence likelihood $p(\mathbf{z}_t|\mathbf{x}_t)$.
4. Resample particles, by sampling from posterior distribution, and reset weights to unity.

1.2 *Particles*

As mentioned in Section 3, a multi-modal probability density can be approximated by the superposition of Gaussian distributions,

$$f_X(x) \approx \sum_{i=0}^{I-1} \frac{w_i}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{1}{2}\left(\frac{x-\mu_i}{\sigma_i}\right)^2\right). \qquad (11.1)$$

The tricky part is how to determine μ_i, σ_i, w_i and I given $f_X(x)$.

A simpler problem is to choose $w_i = 1$ and $\sigma_i = \sigma$, so that

$$f_X(x) \approx \sum_{i=0}^{I-1} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left(\frac{x-\mu_i}{\sigma_i}\right)^2\right). \qquad (11.2)$$

The solution is to choose μ_i to be random samples from the desired density, $f_X(x)$, and to make I large enough

to achieve the desired accuracy. We then determine σ to get the best fit[3]. Each sample is called a particle. Examples of using this method to approximate a Gaussian distribution are shown in Figure 11.2 and Figure 11.3 for 25 and 250 particles. Notice how the random sampling selects more particles around the mode and fewer in the tails of the distribution. Also note how more particles better approximate the desired distribution.

The key concept is that the particles are samples from the desired density. The sample statistics of the particles approximate the population statistics of the desired density; the more particles the better the approximation.

[3] This is an aspect of kernel density estimation.

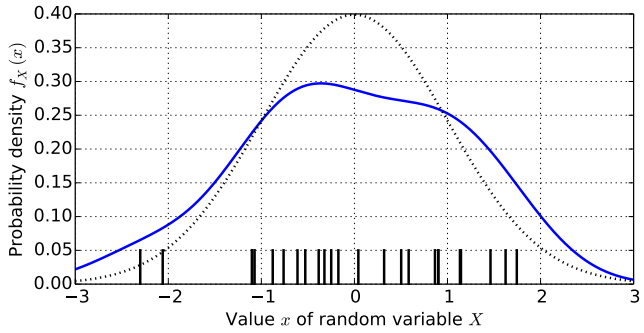


Figure 11.2: Approximation of a Gaussian distribution with 25 uniformly weighted particles.

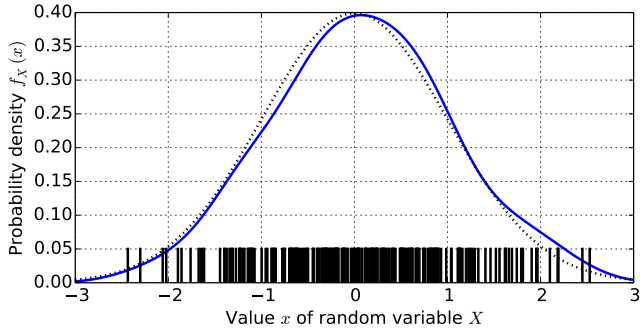


Figure 11.3: Approximation of a Gaussian distribution with 250 uniformly weighted particles.

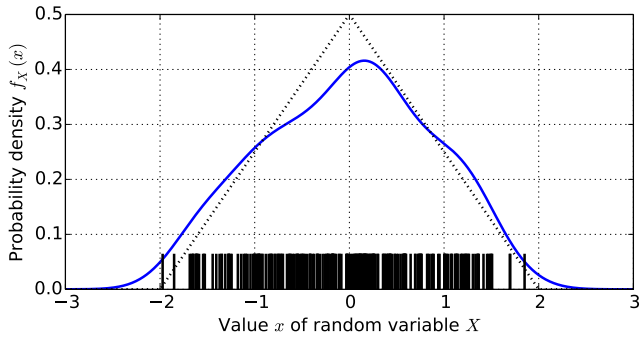


Figure 11.4: Approximation of a triangle distribution with 250 uniformly weighted particles.

1.3 Particle filtering

Particle filters use a set of M particles to represent the belief posterior density, $\text{bel}(\mathbf{x}_t)$, at each time step t:

$$\mathcal{X}_t = \left\{ \mathbf{x}_t^{(0)}, \mathbf{x}_t^{(1)}, \cdots, \mathbf{x}_t^{(M-1)} \right\}. \tag{11.3}$$

Initially, the particles are chosen using *inverse transform sampling* to approximate the initial belief distribution.

Then for each iteration, the predict and update steps of a Bayes filter are applied:

$$\text{predict: } \breve{\text{bel}}(\mathbf{x}_t) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)\, \text{bel}(\mathbf{x}_{t-1}) \mathrm{d}\mathbf{x}_{t-1}, \tag{11.4}$$

$$\text{update: } \text{bel}(\mathbf{x}_t) = \eta p(\mathbf{z}_t|\mathbf{x}_t) \breve{\text{bel}}(\mathbf{x}_t). \tag{11.5}$$

The prediction step is achieved by selecting a new set of particles to represent $\breve{\text{bel}}(\mathbf{x}_t)$,

$$\breve{\mathcal{X}}_t = \left\{ \breve{\mathbf{x}}_t^{(0)}, \breve{\mathbf{x}}_t^{(1)}, \cdots, \breve{\mathbf{x}}_t^{(M-1)} \right\}, \tag{11.6}$$

where each new particle is sampled from the state transition probability density, conditioned on the current input, $\mathbf{u}_t$, and each previous particle, $\mathbf{x}_{t-1}^{(m)} \in \mathcal{X}_{t-1}$,

$$\breve{\mathbf{x}}_t^{(m)} \sim p\left(\mathbf{x}_t | \mathbf{x}_{t-1}^{(m)}, \mathbf{u}_t \right). \tag{11.7}$$

The update step creates a set of weights, one per particle,

$$\mathcal{W}_t = \left\{ w_t^{(0)}, w_t^{(1)}, \cdots, w_t^{(M-1)} \right\}, \tag{11.8}$$

where the weights are selected using the measurement likelihood,

$$w_t^{(m)} = p\left(\mathbf{z}_t | \breve{\mathbf{x}}_t^{(m)} \right). \tag{11.9}$$

1.4 Resampling

While the set of weights, $\mathcal{W}_t$, and set of particles, $\mathcal{X}_t = \breve{\mathcal{X}}_t$ represent the new belief[4], $\text{bel}(\mathbf{x}_t)$, some of the weights may be close to zero and so a resampling step is performed to create a new set of particles, $\mathcal{X}_t$, sampled from $\breve{\mathcal{X}}_t$ using $\mathcal{W}_t$. This step is called *importance sampling* or *resampling*.

The resampling step draws particles from $\breve{\mathcal{X}}_t$ (with replacement) with a probability given by $\mathcal{W}_t$. This can result in many duplicate particles in $\mathcal{X}_t$ but they will be

[4] For example, the estimated mean state can be calculated using a weighted average of the particles:

$$\bar{\mathbf{x}}_t = \frac{\sum_{m=0}^{M-1} w_t^{(m)} \mathbf{x}_t^{(m)}}{\sum_{m=0}^{M-1} w_t^{(m)}}. \tag{11.10}$$

dispersed when $\breve{\mathcal{X}}_{t+1}$ is formed. Note, that the new set of particles approximate the posterior density without the weights, for example, for a 1-D case

$$f_X(x) \approx \sum_{m=0}^{M-1} \frac{w^{(m)}}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\breve{x}^{(m)}}{\sigma}\right)^2\right) \quad (11.11)$$

$$\approx \sum_{m=0}^{M-1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-x^{(m)}}{\sigma}\right)^2\right) \quad (11.12)$$

The resampling step can be avoided with naïve particle filter implementations. However, many of the particles can develop weights close to zero and thus do not contribute to the posterior density. Thus many more particles are required.

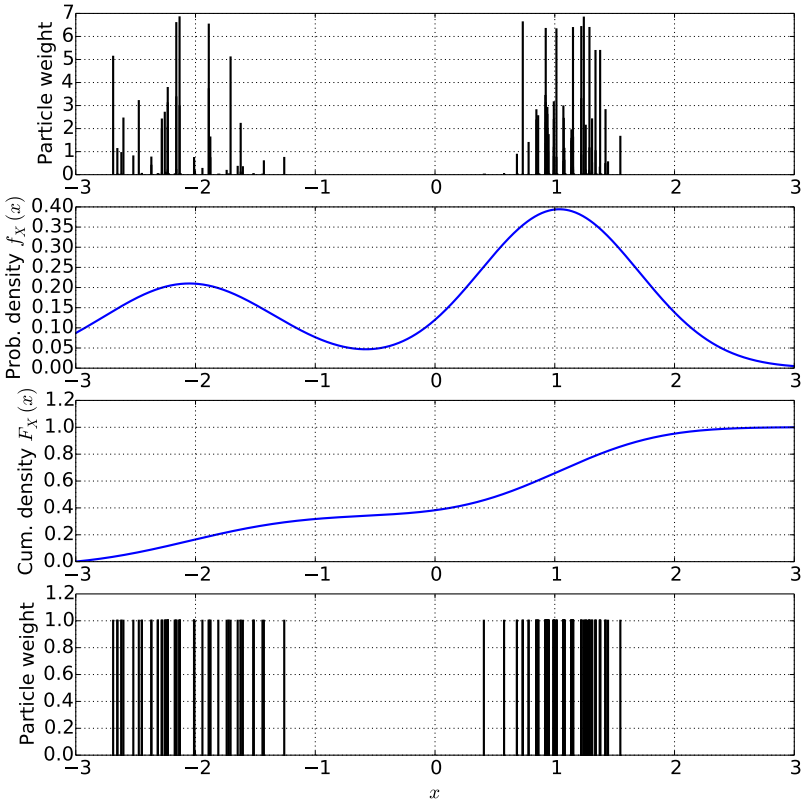


Figure 11.5: Demonstration of particle resampling. Note, many of the particles in the top plot have negligible weights and do not contribute to the belief.

1.5 *Practical considerations*

In practice, the dimensionality of $p\left(\mathbf{x}_t|\mathbf{x}_{t-1}^{(m)}, \mathbf{u}_t\right)$ is too high for it to be used as a lookup table. So again the state transition is often considered to be deterministic

with additive process noise. The resultant state transition conditional probability density is

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) = f_{\mathbf{W}}\left(\mathbf{x}_t - g(\mathbf{x}_{t-1}, \mathbf{u}_t)\right). \tag{11.13}$$

where $f_{\mathbf{W}}(\mathbf{w}_t)$ is the process noise probability density. Similarly, the measurement is considered to be corrupted by additive noise so the measurement likelihood is

$$p(\mathbf{z}_t|\mathbf{x}_t) = f_{\mathbf{V}}\left(\mathbf{z}_t - h(\mathbf{x}_t, \mathbf{u}_t)\right), \tag{11.14}$$

where $f_{\mathbf{V}}(\mathbf{v}_t)$ is the measurement noise probability density.

With these approximations, the particle filter algorithm for each time step is:

1. Create a set of particles, $\check{\mathcal{X}}_t$, to represent the predicted belief where the particles are sampled according to

$$\check{\mathbf{x}}_t^{(m)} \sim f_{\mathbf{W}}\left(\mathbf{x}_t - g(\mathbf{x}_{t-1}^{(m)}, \mathbf{u}_t)\right). \tag{11.15}$$

   This is equivalent to

$$\check{\mathbf{x}}_t^{(m)} = g(\mathbf{x}_{t-1}^{(m)}, \mathbf{u}_t) + \mathbf{w}_t, \tag{11.16}$$

   where $\mathbf{w}_t$ is a process noise random vector sampled from $f_{\mathbf{W}}(\mathbf{w})$.

2. Create a set of particle weights, $\mathcal{W}_t$, where

$$w_t^{(m)} = f_{\mathbf{V}}\left(\mathbf{z}_t - h(\check{\mathbf{x}}_t^{(m)}, \mathbf{u}_t)\right). \tag{11.17}$$

3. Resample the particles to create $\mathcal{X}_t$, where the m^{th} particle is drawn (with replacement) from $\check{\mathcal{X}}$ with a probability proportional to $w_t^{(m)}$.

1.6 What can go wrong?

1. Have too few particles to accurately represent the belief posterior[5]. But even with a large number of particles, none may occur in the vicinity of the correct state. This is called the *particle deprivation problem*. It can occur as the result of variance in the random sampling—an unlucky sequence wipes out the particles near the true state.
2. Choose 'bad' particles in the resampling process. Any sampling variance is amplified during resampling. In practice, the resampling is not performed every iteration.

[5] There are techniques to reduce the number of particles, such as Rao-Blackwellisation to partition the belief as used by the gmapping SLAM algorithm.

3. Have a poor model of the system.
4. Have a poor model of the process and measurement errors.
5. Have an ill-conditioned system, say due to insufficient measurements.

2 *Density extraction*

Density extraction concerns estimation of the posterior density from the particles. There is no simple solution, especially when the state space has many dimensions. There are a number of approaches:

1. Determine the (weighted) mean and variance of the particles and construct a Gaussian density distribution. This forces the posterior to have a single mode.
2. Approximate the posterior with a histogram.
3. Use kernel density estimation[6] where each particle 'location' is replaced by a Gaussian distribution; these are summed to form a continuous estimate of the posterior. Examples are shown in Figure 11.6 and Figure 11.7.
4. Use k-means but this requires an estimate of the number of modes.

[6] http://en.wikipedia.org/wiki/Kernel_density_estimation

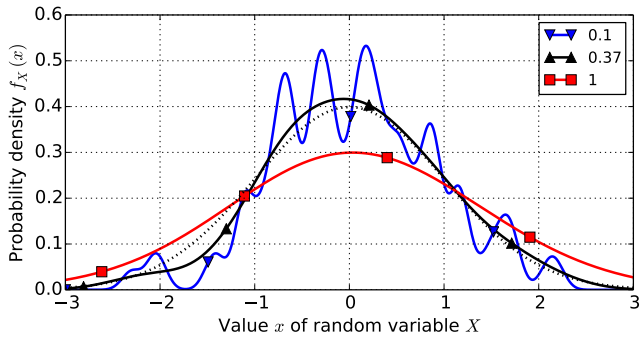


Figure 11.6: Kernel density estimation, 100 particles. 0.37 is the value for σ derived using Silverman's rule of thumb.

3 *Example*

Consider a robot exploring a 2-D world. Let's assume it moves at constant speed and so its state vector is $\mathbf{x}_t = (x_t, y_t, \theta_t)^{\mathrm{T}}$, where x_t, y_t is its position and θ_t is its heading at time step t. Let's also assume that it has

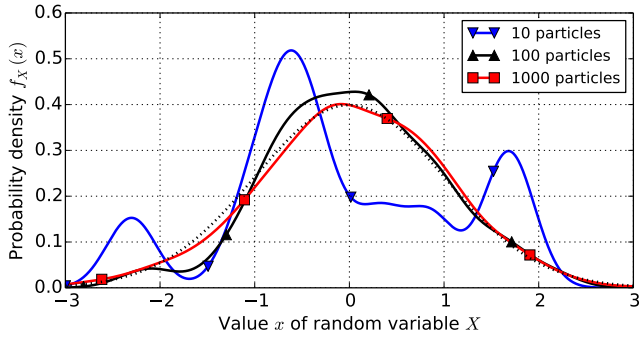


Figure 11.7: Kernel density estimation, 10, 100, 1000 particles.

no idea of its position but it has a sensor that can measure the range and bearing to a beacon. It knows where the beacons are but they all look the same.

One way for the robot to estimate its position and heading is to uniformly distribute a large number of particles[7] throughout the possible state space[8]. Let's assume that the robot's world is bounded by $-\frac{X}{2} \leq x \leq \frac{X}{2}$ and $-\frac{Y}{2} \leq y \leq \frac{Y}{2}$ and its heading angle is bounded by $-\pi \leq \theta \leq \pi$. The m^{th} particle is initialised by $\mathbf{x}_0^{(m)} = \left(x_0^{(m)}, y_0^{(m)}, \theta_0^{(m)}\right)^{\mathrm{T}}$, where

$$x_0^{(m)} = \text{uniform}\left(-\frac{X}{2}, \frac{X}{2}\right), \tag{11.18}$$

$$y_0^{(m)} = \text{uniform}\left(-\frac{Y}{2}, \frac{Y}{2}\right), \tag{11.19}$$

$$\theta_0^{(m)} = \text{uniform}\left(-\pi, \pi\right). \tag{11.20}$$

[7] You can think of each particle as behaving as a model of the robot.

[8] If the robot had some idea of where it was it could distribute more particles there.

The robot moves around making measurements. For each measurement, we need to update the weights of the particles based on the measurements. For each particle it determines the closest beacon and then compares the distance and bearing to the beacon with the distance and bearing measured by the robot. The range and bearing[9] measured by the robot is

$$r_r = \sqrt{(X(\mathbf{x}_t) - x_t)^2 + (Y(\mathbf{x}_t) - y_t)^2} + n_r, \tag{11.21}$$

$$\phi_r = \theta_t - \tan^{-1}\frac{Y(\mathbf{x}_t) - y_t}{X(\mathbf{x}_t) - x_t} + n_\phi, \tag{11.22}$$

and the range and bearing measured by the m^{th} particle are

$$r_r^{(m)} = \sqrt{(X(\mathbf{x}_t^{(m)}) - x_t)^2 + (Y(\mathbf{x}_t^{(m)}) - y_t)^2}, \tag{11.23}$$

$$\phi_r^{(m)} = \theta_t^{(m)} - \tan^{-1}\frac{Y(\mathbf{x}_t^{(m)}) - y_t}{X(\mathbf{x}_t^{(m)}) - x_t}. \tag{11.24}$$

[9] Note, the measurements are a non-linear function of $\mathbf{x}_t$.

In these equations $(X(\mathbf{x}_t), Y(\mathbf{x}_t))$ is the coordinate of the beacon closest to the robot with state $\mathbf{x}_t$, n_r represents range measurement noise, and n_ϕ represents bearing measurement noise. Note, that there is no measurement noise for the particles; they know where they and the beacons are.

The particle weights are chosen using the joint probability density of the range and bearing errors. If the range and bearing errors are independent, then the joint probability density is simply the product of the range and bearing probability densities. This simplifies the weight calculation to

$$w_t^{(m)} = f_R\left(r_t - r_t^{(m)}\right) f_\Phi\left(\phi_t - \phi_t^{(m)}\right), \tag{11.25}$$

where $f_R(r)$ is the range error probability density and $f_\Phi(\phi)$ is the bearing error probability density. Note, it is not necessary to get the amplitude scaling correct since the weights will be normalised.

A Python implementation of this problem is shown in Listing 11.1. Note how the particles are initialised with a random pose, selected uniformly over the search space. Initially the weights are set to 1 and are updated using (11.25). When resampling takes place (see Listing 11.2), the weights are reset to 1.

The robot tries to move in a straight line with constant speed. That is until it hits an obstacle, whereupon it falls over, gets up, and resumes in a random direction, see Listing 11.3. Note that the same code is used to implement both robot and particle, see Listing 11.3.

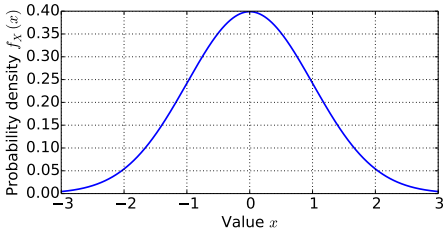


Figure 11.8: Gaussian pdf $f_X(x)$.

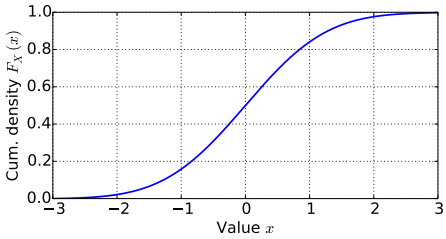


Figure 11.9: Gaussian cdf $F_X(x)$.

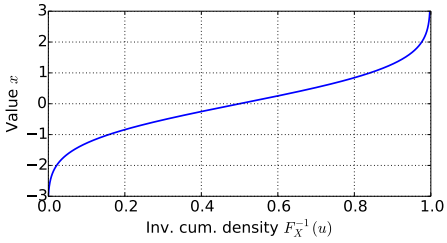


Figure 11.10: Inverse Gaussian cdf $F_X^{-1}(u)$.

4 *Inverse transform sampling*

Let's say we have a density $f_X(x)$ and we wish to draw samples, x_i, from this density. The common approach is called *inverse transform sampling*. It requires the density, $f_X(x)$, integrated to obtain the cumulative density, $F_X(x)$. This is a monotonically increasing function from 0 to 1. Uniformly distributed random numbers, u_i, (with a range of 0 to 1) are then mapped through the inverse of this cumulative density,

$$x_i = F_X^{-1}(u_i). \tag{11.26}$$

In general, obtaining analytic expressions for F_X^{-1} is difficult. In practice, interpolation is used with a discrete probability distribution.

5 Summary

- Histogram filters approximate the posterior density with a histogram[10].
- Particle filters assume the complete state (Markov) approximation.
- Particles are randomly sampled from the initial belief distribution.
- Bayes filtering of the particles modifies the particle weights to update the posterior distribution.
- The particles are resampled (based on importance sampling using the particle weights) and the weights are set back to unity.
- Particle filters do not work out of the box; they require tuning for the problem.
- Despite the mathematics, particle filters are simple to program.

[10] The probability density function is replaced with a probability mass function.

6 Further reading

1. Simple explanation of a particle filter for a non-linear 1-D system, https://www.youtube.com/watch?v=aUkBa1zMKv4

7 Exercises

1. What is the point of resampling?
2. What can go wrong with resampling?
3. How should the particles be initially distributed if you have some prior information?
4. If you need approximately 100 particles per state variable, how many particles are required if the state space has ten dimensions?
5. What is the advantage of a particle filter over a Kalman filter?
6. Are there any restrictions on the state transition model for a particle filter?

7. Do the measurements have to have Gaussian noise for a particle filter?
8. How can the posterior density be determined from the particles?
9. How does a particle filter support multiple hypotheses?

```
# Particle filter example for simple robot using range and
# bearing measurements to known beacons.
# Michael P. Hayes, UCECE (roughly based on an example from Martin J.
    Laubach)
from world import World
from robot import Robot
from particle_filter import ParticleFilter

world_map = ('     @      ',
             ' @          ',
             '            ',
             '           @',
             '      @     ',
             '          @ ',
             '@            ',
             ' @          ',
             '            ',
             '      @     ')

PARTICLE_COUNT = 5000
ROBOT_SPEED = 0.2
RESAMPLE_PERIOD = 1

class Demo(object):

    def setup(self, world):
        self.robot = Robot(world.random_pose())

        particles = []
        for m in range(PARTICLE_COUNT):
            particles.append(Robot(world.random_pose()))

        self.pf = ParticleFilter(particles)

    def loop(self, world, iter):
        robot, pf = self.robot, self.pf

        # Move robot and particles
        robot.transition(1, ROBOT_SPEED, world)
        pf.predict(1, ROBOT_SPEED, world)

        # Update particle weights based on measurements
        z = robot.measure(world)
        pf.update(z, world)

        # Display robot, particles, and estimate of robot position
        world.show_particles(pf.particles, pf.weights)
        estimate, confidence = pf.mean_and_confidence()
        world.show_robot(Robot(estimate),
                         'black' if confidence > 0.95 else 'grey')
        world.show_robot(robot, 'green')

        # Periodically resample particles and set weights back to unity
        if RESAMPLE_PERIOD and (iter % RESAMPLE_PERIOD == 0):
            pf.resample()

world = World(world_map, Demo(), random_seed=2)
world.show()
```

Listing 11.1: Python particle filter to solve lost in the countryside problem.

```
# Michael P. Hayes, UCECE
from numpy import ones
from bisect import bisect_left
from random import uniform
from copy import copy

class ParticleFilter(object):

    def __init__(self, particles):
        """Create particle filter with supplied particles.
        The particles require a state attribute and transition, weight,
        and mean methods"""

        self.particles = particles
        self.weights = ones(len(self.particles))

    def predict(self, dt, u, *args):
        """Perform prediction step using state transition and control
           vector u"""

        for p in self.particles:
            p.transition(dt, u, *args)

    def update(self, z, *args):
        """Perform update step using weights and measurement vector z"""

        for m, p in enumerate(self.particles):
            self.weights[m] *= p.weight(z, *args)

    def resample(self):
        """Resample particles in proportion to their weights"""

        cum_weights = self.weights.cumsum()
        cum_weights /= cum_weights[-1]

        new_particles = []
        for _ in self.particles:
            m = bisect_left(cum_weights, uniform(0, 1))
            p = self.particles[m]
            # Create new particle with same state as particle p
            new_particles.append(copy(p))

        for m, p in enumerate(new_particles):
            self.particles[m] = p

        self.weights = ones(len(self.particles))

    def mean_and_confidence(self):
        """Return mean particle state and confidence"""

        return self.particles[0].mean_and_confidence(self.particles,
                                                     self.weights)

    def is_degenerate(self):

        w = self.weights
        w = w / sum(w)

        return 1 / sum(w ** 2) < 0.5 * len(w)
```

Listing 11.2: Python particle filter class.

```
# Michael P. Hayes, UCECE
from __future__ import absolute_import
from __future__ import division
from numpy import sqrt, exp, pi, cos, sin, radians, degrees, array,
    arctan2
from random import uniform, gauss
from pose import Pose

SPEED_SIGMA = 0.1
HEADING_SIGMA = 1.0
RANGE_SENSOR_SIGMA = 1.0
BEARING_SENSOR_SIGMA = 10.0
HEADING_SENSOR_SIGMA = 30.0

def gauss_pdf(z, sigma):

    return exp(-0.5 * (z / sigma) ** 2) / (sqrt(2 * pi) * sigma)

def angle_wrap(a):
    """Wrap angle into range -180 <= a < 180"""
    if a < -180 : a += 360
    elif a > 180 : a -= 360
    return a

class Robot(object):
    def __init__(self, pose):
        """Create Robot with pose (x, y, h)"""

        self.x = pose.x
        self.y = pose.y
        self.h = pose.h

    def __repr__(self):
        return "(%.2f, %.2f, h=%.1f)" % (self.x, self.y, self.h)

    @property
    def state(self):
        """Return robot state (its pose)"""

        return Pose(self.x, self.y, self.h)

    def transition(self, dt, speed, world):
        """Perform robot state transition"""

        speed += gauss(0, SPEED_SIGMA)
        self.h += gauss(0, HEADING_SIGMA)

        while True:
            r = radians(self.h)
            dx = cos(r) * speed * dt
            dy = sin(r) * speed * dt
            if world.is_free(self.x + dx, self.y + dy):
                break
            self.h = uniform(0, 360)

        self.x += dx
```

Listing 11.3: Python robot class.

```
        self.y += dy

    def range(self, world):
        """Find distance to nearest beacon"""

        return world.distance_to_nearest_beacon(self.state)

    def bearing(self, world):
        """Find bearing to nearest beacon"""

        return angle_wrap(world.angle_to_nearest_beacon(self.state) -
            self.h)

    def measure_range(self, world):
        """Return range to nearest beacon"""

        r = self.range(world)
        return r + gauss(0, RANGE_SENSOR_SIGMA)

    def measure_bearing(self, world):

        b = self.bearing(world)
        return b + gauss(0, BEARING_SENSOR_SIGMA)

    def measure_heading(self):

        return self.h + gauss(0, HEADING_SENSOR_SIGMA)

    def measure(self, world):

        return self.measure_range(world), self.measure_bearing(world)

    def range_weight(self, r, world):

        return gauss_pdf(r - self.range(world), RANGE_SENSOR_SIGMA)

    def bearing_weight(self, b, world):

        return gauss_pdf(angle_wrap(b - self.bearing(world)),
            BEARING_SENSOR_SIGMA)

    def weight(self, z, world):
        """Calculate weight given measurement vector z"""

        r, b = z
        return self.range_weight(r, world) * self.bearing_weight(b,
            world)

    def mean_and_confidence(self, particles, weights):
        """Return mean particle position and confidence"""

        x = array([p.x for p in particles])
        y = array([p.y for p in particles])
        a = radians(array([p.h for p in particles]))

        m_x = (x * weights).sum() / weights.sum()
        m_y = (y * weights).sum() / weights.sum()

        # Calculate weighted mean heading in Cartesian coords to avoid
            wrap
        m_h = degrees(arctan2((sin(a) * weights).sum(),
                              (cos(a) * weights).sum()))

        # Count particles in the immediate vicinity of estimate
```

12
Motion models

Kinematics describes the effect of control actions on a robot *configuration*. For a rigid mobile robot, the configuration can be described by 3-D Cartesian coordinates and three Euler angles (pitch, roll, and yaw). If we constrain the robot to a plane, the configuration is reduced to three variables, 2-D location and an angle (called heading or bearing). This reduced configuration is called the robot *pose*,

$$\mathbf{x}_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix}. \tag{12.1}$$

Traditionally, kinematics has been considered deterministic but there is always uncertainty due to control noise or unmodelled exogenous effects. These uncertainties can be considered using *probabilistic kinematics*[1].

[1] This collapses to deterministic kinematics when the modelled standard deviations are zero.

1 Probabilistic motion models

Probabilistic motion models are described by a conditional density $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$. They have two uses:

1. To determine the belief of the next pose[2] given the belief of the current pose and the control output.
2. To sample from when propagating the particles in a particle filter.

[2] Say for a histogram filter.

There are two common mobile robot motion models:

Velocity motion model This only considers the control outputs, assumed to be velocity commands given to the robot motors.

Odometry motion model This requires odometry sensors[3] It is more accurate than the velocity motion model but since it requires measurements it cannot be used for motion planning.

[3] Usually by having encoders to measure the revolution of the robot's wheels. Alternatively, IMUs are used with GPS (when outside) or light sensors, such as: monocular cameras, stereo cameras, laser scanners, time-of-flight cameras, and structured light cameras.

2 Velocity motion model

The velocity motion model[4] assumes there are two control outputs:

Translational speed [5] v

Rotational speed [6] ω

The control vector[7] at time step t is thus

$$\mathbf{u}_t = \begin{bmatrix} v_t \\ \omega_t \end{bmatrix}. \tag{12.2}$$

[4] For a differential drive robot.

[5] A positive value implies a forward motion.

[6] A positive value implies an anti-clockwise rotation.

[7] In the following the time dependence of the speed controls are dropped to simplify the notation.

2.1 Deterministic model

In the following we assume that there is no process error, for example, there is no wheel slippage. There are three cases to consider.

Pure translation, $\omega = 0$ Here the robot moves along a straight path and the motion model is

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} v\Delta t \cos\theta_{t-1} \\ v\Delta t \sin\theta_{t-1} \\ 0 \end{bmatrix}. \tag{12.3}$$

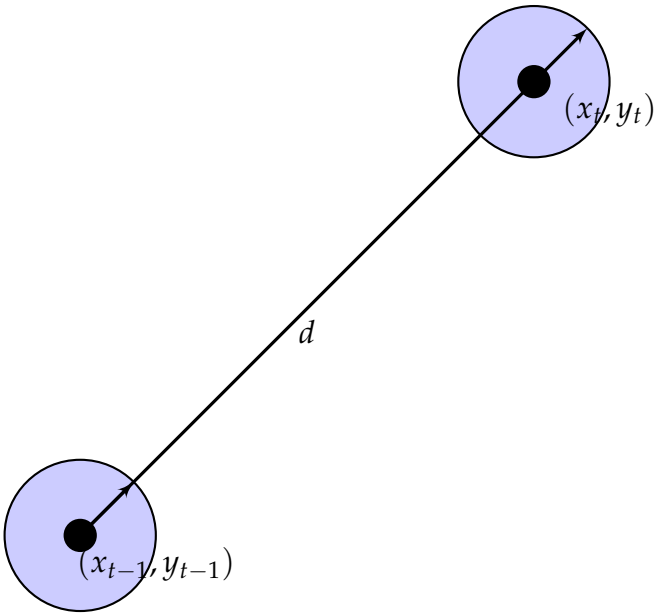


Figure 12.1: Velocity model for a zero angular speed. In this example, $\theta_t = \pi/4$. The distance travelled is $d = v\Delta t$.

Pure rotation, $v = 0$ Here the robot will rotate on the spot, and the motion model is

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \omega_t \Delta t \end{bmatrix}. \tag{12.4}$$

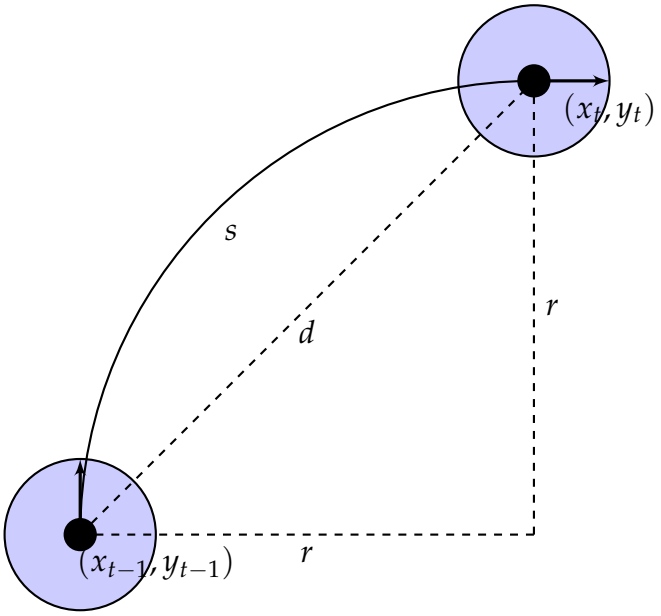


Figure 12.2: Velocity model for non-zero translational and angular speeds. In this example, $\theta_t = 0$. The distance travelled along the arc of radius, $r = v/\omega$, is $s = v\Delta t$. The Cartesian distance travelled is $d = 2r \tan \frac{\omega \Delta t}{2}$.

Translation with rotation Here the robot moves on a circle of radius $r = |v/\omega|$ so the next pose[8] is given by

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} -\frac{v}{\omega} \sin \theta_{t-1} + \frac{v}{\omega} \sin (\theta_{t-1} + \omega \Delta t) \\ \frac{v}{\omega} \cos \theta_{t-1} - \frac{v}{\omega} \cos (\theta_{t-1} + \omega \Delta t) \\ \omega \Delta t \end{bmatrix}. \tag{12.5}$$

The distance travelled along the arc between poses is $s_t = |v| \Delta t$.

The motion model (12.5) can be summarised as

$$\mathbf{x}_t = \mathbf{x}_{t-1} + g(\mathbf{x}_{t-1}, \mathbf{u}_t). \tag{12.6}$$

However, it cannot be expressed[9] as

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{A}_t \mathbf{u}_t, \tag{12.7}$$

where $\mathbf{A}$ is a state transition matrix. This means we cannot use a standard Kalman filter. We either need to linearise the motion model and apply the EKF or UKF or use a histogram or particle filter.

[8] In practice, the updated pose heading needs to be computed modulo 2π.

[9] Due to the reciprocal of ω and trig. functions.

2.2 *Error model*

A real robot is subject to noise and the actual speeds will differ from the desired ones. This is modelled by adding an error vector, $\mathbf{u}_\epsilon$, to the desired control vector, $\mathbf{u}_t$. $\mathbf{u}_\epsilon = (v_\epsilon, \omega_\epsilon)^{\mathrm{T}}$ denotes the values of a zero-mean random variable vector $(V_\epsilon, \Omega_\epsilon)^{\mathrm{T}}$.

Assuming that the translation and angular speed errors are independent then their joint probability density simplifies to

$$f_{V_\epsilon,\Omega_\epsilon}(v_\epsilon, \omega_\epsilon|\mathbf{u}_t) = f_{V_\epsilon}(v_\epsilon|\mathbf{u}_t) f_{\Omega_\epsilon}(\omega_\epsilon|\mathbf{u}_t). \tag{12.8}$$

Here $f_{V_\epsilon}(v|\mathbf{u}_t)$ and $f_{\Omega_\epsilon}(\omega|\mathbf{u}_t)$ describe suitable zero-mean probability density functions, say a Gaussian or a triangle distribution. The tricky thing is how to choose the standard deviations of the distributions. A common approach is to assume that the error increases with the desired control, i.e.,

$$\begin{bmatrix} \sigma_V \\ \sigma_\Omega \end{bmatrix} = \begin{bmatrix} \alpha_1 & \alpha_2 \\ \alpha_3 & \alpha_4 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}. \tag{12.9}$$

Here α_i are robot-specific error parameters[10] and $\mathbf{u}_t$ $(v, \omega)^{\mathrm{T}}$.

[10] These are smaller for more accurate robots.

2.3 *Inverse motion model*

Bayes filters[11] require $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$ in the predict step, since

[11] Except particle filters.

$$\breve{\mathrm{bel}}(\mathbf{x}_t) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)\, \mathrm{bel}(\mathbf{x}_{t-1}) \mathrm{d}\mathbf{x}_{t-1}. \tag{12.10}$$

To evaluate $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$ we need an inverse motion model, where the estimated speeds errors for the pose to change from $\mathbf{x}_{t-1}$ to $\mathbf{x}_t$ are found using

$$\mathbf{u}_\epsilon = g^{-1}(\mathbf{x}_t - \mathbf{x}_{t-1}) - \mathbf{u}_t, \tag{12.11}$$

where g^{-1} is the inverse of (12.6) and $\mathbf{u}_\epsilon = (v_\epsilon, \omega_\epsilon)^{\mathrm{T}}$. Then we can use the joint probability density for the errors to yield

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) = f_{V_\epsilon,\Omega_\epsilon}(v_\epsilon, \omega_\epsilon|\mathbf{u}_t). \tag{12.12}$$

2.4 *Sampling from the conditional probability*

Particle filters need to sample from $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$ during the transition of the particles from $\mathbf{x}_{t-1}$ to $\mathbf{x}_t$. This is achieved by sampling[12] v_ϵ and ω_ϵ from the joint error density, $f_{V_\epsilon,\Omega_\epsilon}(v_\epsilon, \omega_\epsilon|\mathbf{u}_t)$ (see (12.8)), and then mapping these speeds through the forward motion model (12.6) to find the change in pose,

$$\mathbf{x}_t = \mathbf{x}_{t-1} + g(\mathbf{x}_{t-1}, \mathbf{u}_t + \mathbf{u}_\epsilon), \tag{12.13}$$

where $\mathbf{u}_\epsilon = (v_\epsilon, \omega_\epsilon)^{\mathrm{T}}$.

[12] If the distributions are uniform or Gaussian, then the math functions `rand/uniform` and `randn` can be used. Otherwise it is necessary to use inverse transform sampling with the inverse of the desired cumulative density.

3 *Odometry motion model*

Most wheeled robots have sensors on the wheels to estimate the distance each wheel travels. From this the pose of the robot can be estimated. These measurements are relative to the starting pose of the robot and are subject to drift.

Odometry is really a form of measurement and so should be incorporated in the update step of a Bayesian filter. Unfortunately, to achieve this requires augmenting the robot's state with velocity and thus the estimation problem becomes harder. In practice, a sleight-of-hand is employed to approximate the odometry as a control output!

The odometry model uses relative motion information from the robot's odometry. These measurements are in the local coordinate frame of the robot whereas the robot's pose are in global coordinates. Denoting local measurements with a check accent, the local pose of a robot on a plane is

$$\check{\mathbf{x}}_t = \left(\check{x}_t, \check{y}_t, \check{\theta}_t\right)^T. \tag{12.14}$$

Using two of these local poses, $\check{\mathbf{x}}_t$ and $\check{\mathbf{x}}_{t-1}$, the odometry motion model infers the the probability of the transition of $\mathbf{x}_{t-1}$ to $\mathbf{x}_t$. Thus the contrived control input is two local poses,

$$\mathbf{u}_t = \begin{pmatrix} \check{\mathbf{x}}_{t-1} \\ \check{\mathbf{x}}_t \end{pmatrix}. \tag{12.15}$$

The algorithm decomposes[13] the pose change into a rotation, ϕ_1, a translation, d, and another rotation, ϕ_2,

$$(\phi_1, d, \phi_2)^T = \mathcal{D}\left(\mathbf{x}_t - \mathbf{x}_{t-1}\right). \tag{12.16}$$

[13] Denoted by the operator $\mathcal{D}$.

The local poses are decomposed in a similar manner

$$\left(\check{\phi}_1, \check{d}, \check{\phi}_2\right)^T = \mathcal{D}\left(\check{\mathbf{x}}_t - \check{\mathbf{x}}_{t-1}\right). \tag{12.17}$$

The conditional probability for the new pose given the previous pose and the odometry is

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) \approx f_{\Phi_1}(\phi_1 - \check{\phi}_1) f_D(d - \check{d}) f_{\Phi_2}(\phi_2 - \check{\phi}_2), \tag{12.18}$$

where the standard deviations of the probability densities are determined using

$$\sigma_{\Phi_1} = \alpha_1 |\phi_1| + \alpha_2 d, \tag{12.19}$$
$$\sigma_D = \alpha_3 \left(|\phi_2| + |\phi_2|\right) + \alpha_4 d, \tag{12.20}$$
$$\sigma_{\Phi_2} = \alpha_1 |\phi_2| + \alpha_2 d, \tag{12.21}$$

where α_i are constants.

Once the conditional probability $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$ has been determined, it can be sampled from to determine the new state of a particle (see Section 1.3 in Chapter 11).

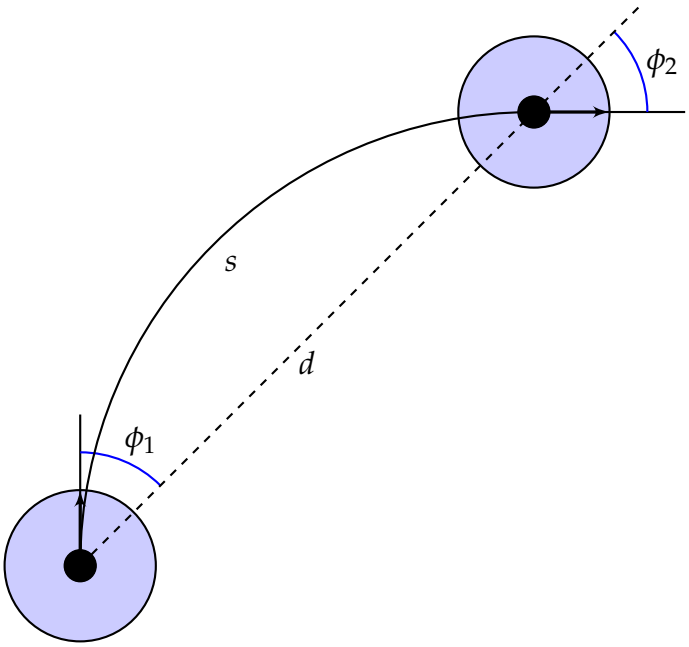


Figure 12.3: The robot motion is approximated by two rotations and a translation.

4 *Map-based motion models*

If the robot has an occupancy map, it can use it to constrain its position since it must always be in a free space. A map based motion model with a map, $\mathbf{M}$, has a conditional state transition probability, $p\left(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t, \mathbf{M}\right)$. In

practice, determining this is difficult. An approximation is to factor the map-based motion model into two components,

$$p\left(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t, \mathbf{M}\right) = \eta \frac{p\left(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t\right) p\left(\mathbf{x}_t|\mathbf{M}\right)}{p(\mathbf{x}_t)}, \quad (12.22)$$

where η is a normaliser. The second term expresses the consistency of the pose with the map. For example, $p\left(\mathbf{x}_t|\mathbf{M}\right) = 0$ if the robot pose is placed in an occupied grid cell.

5 Mathematics

This section gives more detailed mathematics for the velocity model derivation, linearised velocity model, inverse velocity model, and odometry model decomposition.

5.1 Velocity model derivation

Assuming a constant translation speed v and rotational speed ω, the robot moves on a circle of radius, r,

$$r = \left|\frac{v}{\omega}\right|. \quad (12.23)$$

Note, when the rotational speed, ω, is zero then the circle has an infinite radius corresponding to a straight line. The centre of the circle is given by

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} x - r\sin\theta \\ y + r\cos\theta \end{bmatrix}. \quad (12.24)$$

Thus

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} x_{t-1} - r\sin\theta_{t-1} \\ y_{t-1} + r\cos\theta_{t-1} \end{bmatrix} = \begin{bmatrix} x_t - r\sin\theta_t \\ y_t + r\cos\theta_t, \end{bmatrix} \quad (12.25)$$

and so

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} x_{t-1} - r\sin\theta_t + r\sin\left(\theta_{t-1} + \omega\Delta t\right) \\ y_{t-1} + r\cos\theta_t - r\cos\left(\theta_{t-1} + \omega\Delta t\right). \end{bmatrix} \quad (12.26)$$

5.2 Linearised velocity motion model

Using the double angle formulae, (12.5) can be expanded as

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} -\frac{v}{\omega}\sin\theta_{t-1} + \frac{v}{\omega}\left(\sin\theta_{t-1}\cos\omega\Delta t + \cos\theta_{t-1}\sin\omega\Delta t\right) \\ \frac{v}{\omega}\cos\theta_{t-1} - \frac{v}{\omega}\left(\cos\theta_{t-1}\cos\omega\Delta t - \sin\theta_{t-1}\sin\omega\Delta t\right) \\ \omega\Delta t \end{bmatrix}. \tag{12.27}$$

If $\omega\Delta t \ll 1$ then[14]

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} \approx \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} -\frac{v}{\omega}\sin\theta_{t-1} + \frac{v}{\omega}\left(\sin\theta_{t-1} + \omega\Delta t\cos\theta_{t-1}\right) \\ \frac{v}{\omega}\cos\theta_{t-1} - \frac{v}{\omega}\left(\cos\theta_{t-1} - \omega\Delta t\sin\theta_{t-1}\right) \\ \omega\Delta t \end{bmatrix}, \tag{12.28}$$

and so

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} \approx \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} v\Delta t\cos\theta_{t-1} \\ v\Delta t\sin\theta_{t-1} \\ \omega\Delta t \end{bmatrix}. \tag{12.29}$$

This is now a linear model[15] given θ_{t-1},

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} \approx \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} \Delta t\cos\theta_{t-1} & 0 \\ \Delta t\sin\theta_{t-1} & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}. \tag{12.30}$$

Another thing to note is this motion model assumes the speeds can instantaneously jump between time steps. It also assumes that the robot velocity changes without delay. In practice, the effect of the velocity jump is reduced by using a smaller time step Δt.

[14] Using $\sin\alpha \approx \alpha$ and $\cos\alpha \approx 1$ when $\alpha \ll 1$.

[15] It assumes the robot has followed a straight path instead of a curved path.

5.3 Inverse velocity model

Estimates of the effective speeds given the difference between two poses can be found naïvely[16] from

$$\hat{w} = \frac{\theta_t - \theta_{t-1}}{\Delta t}, \tag{12.31}$$

$$\hat{v} = \frac{d\hat{w}}{2\tan\left(\frac{\hat{\omega}\Delta t}{2}\right)}, \tag{12.32}$$

where

$$d = \sqrt{(y_t - y_{t-1})^2 + (x_t - x_{t-1})^2}. \tag{12.33}$$

[16] There are more accurate but more complicated inverse models.

5.4 Odometry model decomposition

The odometry model decomposition denoted by $\mathcal{D}(\mathbf{x}_t - \mathbf{x}_{t-1})$ is given by

$$\phi_1 = \tan^{-1}\frac{y_t - y_{t-1}}{x_t - x_{t-1}} - \theta_{t-1}, \tag{12.34}$$

$$d = \sqrt{(y_t - y_{t-1})^2 + (x_t - x_{t-1})^2}, \tag{12.35}$$

$$\phi_2 = \theta_t - \theta_{t-1} - \phi_1. \tag{12.36}$$

6 Summary

1. The odometry motion model fudges the odometry measurements as a control input to avoid having to augment the robot pose with velocities.
2. The odometry measurement uses differences between local pose estimates to determine the conditional probability for the pose transition.
3. The velocity motion model is better for path planning.
4. These motion models have ignored dynamics; thus a fast moving robot or heavily loaded robot will behave differently.
5. A map can assist a motion model.

7 Exercises

1. What is the advantage of the odometry motion model over the velocity model?
2. Is the velocity motion model linear?
3. Is the odometry motion model linear?
4. How do the motion models handle dynamics?
5. What is the principle of the odometry motion model?
6. What are the two ways a probabilistic motion model is used?
7. Why cannot the odometry model be used by a path planner?

8. If the joint probability density for the speeds is Gaussian, how would you choose $\mathbf{x}_t$ given $\mathbf{u}_t = (v, \omega)^{\mathrm{T}}$ and $\mathbf{x}_{t-1}$ for a particle filter?

9. When is the inverse velocity motion model needed?

10. Comment on the validity of

$$f_{V_\epsilon,\Omega_\epsilon}(v_\epsilon, \omega_\epsilon|\mathbf{u}_t) = f_{V_\epsilon}(v_\epsilon|\mathbf{u}_t) f_{\Omega_\epsilon}(\omega_\epsilon|\mathbf{u}_t).$$

11. Why are the densities in the above equation conditional on $\mathbf{u}_t$?

13
Mapping

Robots use maps to model the world. There are a number of problems:

Size The world is large and thus maps require a lot of storage and computing.

Noise Sensors and actuators are noisy and this complicates map making.

Perceptual ambiguity Many places look the same.

Cycles A robot can traverse many paths throughout its world.

1 Maps

A map is a list of objects and their locations. There are two classes of map representation:

Feature based where each entry corresponds to a feature. For example, a feature might be a road and thus the map is a network of roads.

Location based where each entry corresponds to a location. For example, a topographic map. In general, these are volumetric maps and thus can be wasteful of memory. Location based maps, such as an occupancy grid map indexed by robot position make it easy to find paths.

There are many different map implementations, using grids (multidimensional arrays), graphs, etc. There is no universally 'best' implementation.

2 *Occupancy maps*

Occupancy maps are useful to determine if a location is occupied or free; this is useful for robot navigation since it cannot move into an occupied location. Most occupancy maps use grids to discretise the world.

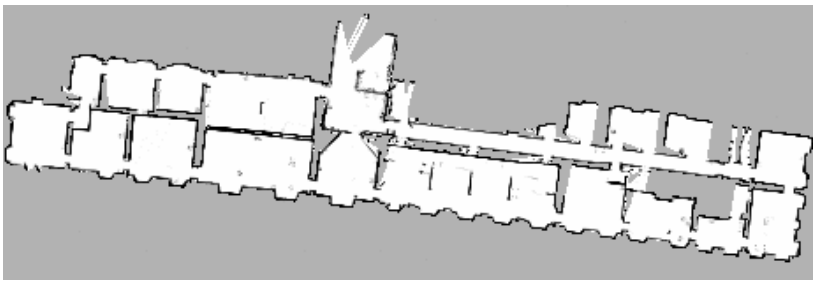


Figure 13.1: Occupancy grid example. White is free, black is occupied, grey is unsure. ROS occupancy values are given by $(255 - v)/255$ where v is the pixel value (in the range 0 to 255).

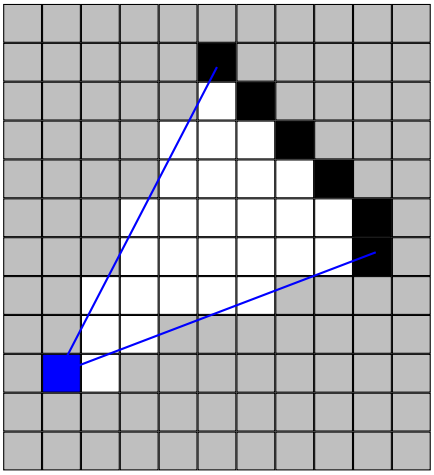


Figure 13.2: Occupancy grid scanning. The robot is at the blue square with position (x, y) and orientation θ. The blue lines denote the limits of the perceptual field of the sensor. The black cells at range r are considered occupied; the white cells within the scanning region are considered free; gray cells are unknown.

When generating an occupancy grid map, $\mathbf{M}$, the goal is to generate a posterior probability (belief) given all the sensor measurements from known locations[1]:

$$\text{bel}\left(\mathbf{M}_t\right) = p\left(\mathbf{M}_t | \mathbf{z}_{0:t}, \mathbf{x}_{0:t}\right). \tag{13.1}$$

Most occupancy grid maps are 2-D floor plans and thus they can be interpreted as a 2-D array of grid cells,

$$\mathbf{M} = \left\{M_{ij}\right\}, \tag{13.2}$$

where M_{ij} is a discrete random variable for the grid cell with indices i and j mapped to world coordinates.

Due to the large size of an occupancy grid it is infeasible to calculate the posterior map probability. The problem is simplified by assuming that each grid cell is independent, so that

$$\text{bel}\left(\mathbf{M}_t\right) \approx \Pi_{ij} \text{bel}\left(M_{ijt}\right). \tag{13.3}$$

[1] Note, the control information, $\mathbf{u}_{0:t}$, is not required since the path is specified by $\mathbf{x}_{0:t}$.

Note, $p(M_{ijt})$ denotes the probability that grid cell M_{ij} is free at time-step t. Thus a discrete Bayes filter is used.

To further reduce the complexity, occupancy grids often use a binary value for each grid cell: 0 means free, 1 means occupied. Thus a 100×100 occupancy grid has 10,000 cells and can represent 2^{10000} maps. The belief of each cell can be estimated with a discrete binary Bayes filter.

2.1 Occupancy grid algorithm

A discrete Bayes filter algorithm[2] to calculate the occupancy grid probabilities is shown in Algorithm 1.

[2] Note, there is no predict step since the map is assumed to be static.

for *each measurement* $\mathbf{z}_t$ *at pose* $\mathbf{x}_t$**:**
  for *each cell* M_{ij} *in perceptual field of* $\mathbf{z}_t$**:**
    $p(M_{ij}) = p(M_{ij}) \times \frac{p(\mathbf{z}_t|M_{ij},\mathbf{x}_t)}{p(\mathbf{z}_t,\mathbf{x}_t)}$

Algorithm 1: Occupancy grid algorithm for the posterior belief. $p(M_{ij})$ is the probability for cell M_{ij} being occupied (initialised with the prior that M_{ij} is occupied, usually 0.5); $p(\mathbf{z}_t|M_{ij}, \mathbf{x}_t)$ is the likelihood of measurement $\mathbf{z}_t$ given M_{ij} is occupied.

The measurements assign probabilities using:

$$p(\mathbf{z}_t|M_{ij}, \mathbf{x}_t) = \begin{cases} p_{\text{occupied}} & \text{if grid cell at measured range} \\ p_{\text{free}} & \text{if grid cell closer than measured range} \\ 0.5 & \text{if grid cell beyond measured range} \end{cases} \tag{13.4}$$

The values of p_{occupied} and p_{free} depends on the amount of evidence a measurement carries. If the measurements were noise free, then $p_{\text{occupied}} = 1$ and $p_{\text{free}} = 0$.

for *each measurement* $\mathbf{z}_t$ *at pose* $\mathbf{x}_t$**:**
  for *each cell* M_{ij} *in perceptual field of* $\mathbf{z}_t$**:**
    $l(M_{ij}) = l(M_{ij}) + \log \frac{p(\mathbf{z}_t|M_{ij},\mathbf{x}_t)}{p(\mathbf{z}_t|\overline{M}_{ij},\mathbf{x}_t)}$

Algorithm 2: Occupancy grid algorithm using log odds for the posterior belief, c.f., Algorithm 1. $l(M_{ij})$ is the log odds for cell M_{ij} being occupied (usually initialised with a prior of 0) and $p(\mathbf{z}_t|\overline{M}_{ij}, \mathbf{x}_t)$ is the likelihood of measurement $\mathbf{z}_t$ given M_{ij} is free.

In practice, log odds are used to reduce numerical instabilities when $p \approx 0$ or $p \approx 1$,

$$l = \log \frac{p}{1-p}. \tag{13.5}$$

The probability is easily found from the log odds using

$$p = 1 - \frac{1}{1 + \exp(l)}. \tag{13.6}$$

p	l
0.99	4.9
0.9	2.2
0.5	0
0.1	−2.2
0.01	−4.9

Table 13.1: Mapping between probabilities and log odds.

2.2 *Loop closure*

Grid maps are difficult to correct when there is a steady drift or when it has jumps. For example, consider the case where a robot navigates in a loop back to its known starting position. If the odometry is inaccurate, then the map of its path will not form a closed curve, see Figure 13.3. Fixing up the map is a tricky problem.

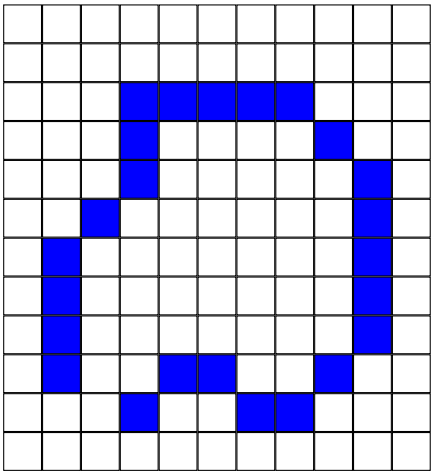


Figure 13.3: The loop closure problem with a grid map. A robot explores its environment and then returns to its starting point; however, due to errors in localisation the robot's path (denoted by blue cells) is not closed.

3 *Topological maps*

Topological maps (or graph based maps) store the map of the environment using a graph data structure. In the graph, each vertex represents robot and landmark poses, while the edges represent constraints on the relative poses of the two nodes. Since topological maps do not rely on metric measurements, they avoid some of the difficulties scaling to large environments. Additional advantages include:

- Efficient and compact representation of the map.

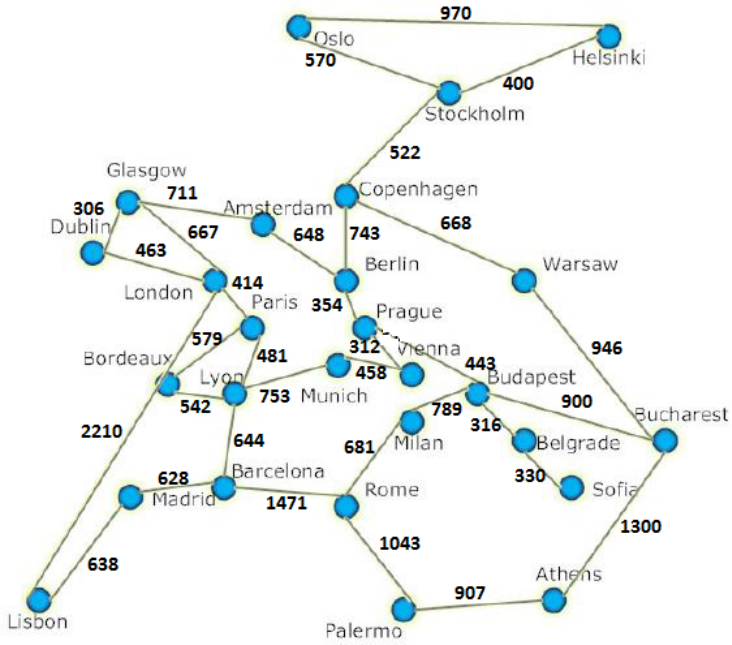


Figure 13.4: Example topological map showing some of the cities in Europe; the edges indicate the distances between the cities.

- Logical organisation for tasks such as path planning. The map is stored as a graph and this makes it possible to use existing graph algorithms, such as finding the shortest path between non-adjacent vertices.

- Loop closure on a local graph-based map is automatic and straight forward, as it just involves adding an edge between the start and end vertices. When a global map is required, a geometric spanner (t-spanner) can be used to close distant gaps introduced by the loop closure

- If two places appear similar, then loop closure is attempted resulting in a data association failure.

4 *Hybrid maps*

Hybrid maps have been developed to take advantage of the complementary strengths that grid and topological maps provide. For example, the map may be partitioned as independent local maps and these are stored in a graph structure. This is called sub-mapping. To produce a global map, the Dijkstra projection is used to find the pose of each submap relative to a single reference frame. However, this only works if there are no loops, otherwise a non-linear least squares optimisation is required.

5 Sensor fusion

Using multiple sensors to generate a map is difficult since different sensors respond to the environment differently. For example, a laser scanner can 'see' through glass whereas an ultrasonic sensor cannot.

The popular solution to generating a map with different sensor modalities is to create a separate maps for each sensor and then fuse the maps together. For example, consider an occupancy grid. Denoting $\mathbf{M}^k$ as the map generated with the k^{th} sensor, the pessimistic estimate for the combined map is

$$p(M_{ij}) = \max_k p(M_{ij}^k). \tag{13.7}$$

If any sensor says a cell is occupied then so will the combined map.

6 Dynamic environments

One approach to deal with dynamic environments is to create both short-term and long-term maps. The short term map contains recent sensor data, and as it does not contain significant odometry error, it is used for obstacle avoidance and localisation. Once a short-term map has matured, it is then added to the long-term map and used for navigation and path-planning.

7 Mapping algorithm derivation

Using Bayes' theorem, the probability that a map cell is occupied given a measurement is

$$P(M|Z) = \frac{P(Z|M)P(M)}{P(Z)}, \tag{13.8}$$

and the the probability that a map cell is free given a measurement is

$$P(\overline{M}|Z) = \frac{P(Z|\overline{M})P(\overline{M})}{P(Z)}. \tag{13.9}$$

Using odds,

$$O(M) = \frac{P(M)}{P(\overline{M})} \tag{13.10}$$

and so Bayes' theorem for odds is

$$O(M|Z) = \frac{P(M|Z)}{P(\overline{M}|Z)} = \frac{P(Z|M)P(M)}{P(Z|\overline{M})P(\overline{M})}, \quad (13.11)$$

$$= \lambda(Z|M)O(M), \quad (13.12)$$

where

$$\lambda(Z|M) = \frac{P(Z|M)}{P(Z|\overline{M})}. \quad (13.13)$$

Using log odds,

$$\log O(M|Z) = \log \lambda(Z|M) + \log O(M). \quad (13.14)$$

There are three cases to consider, where R is the measured range and D is the distance to the map grid cell of interest:

1. If the measured range is less than the map cell range then we cannot infer anything,

$$\log \lambda(R < D|M) = \log 1 = 0. \quad (13.15)$$

2. If the measured range is the same as map cell range, then it is likely that the map cell is occupied. If we assume that $P(R = D|M) = 0.06$ and $P(R = D|\overline{M}) = 0.005$, then

$$\log \lambda(R < D|M) = \log \frac{0.06}{0.005} = \log 12 = 2.5. \quad (13.16)$$

3. If the measured range is further than the map cell range, then it is likely that the map cell is free. If we assume that $P(R > D|M) = 0.2$ and $P(R > D|\overline{M}) = 0.9$, then

$$\log \lambda(R < D|M) = \log \frac{0.2}{0.9} = \log 0.22 = -1.5. \quad (13.17)$$

8 Summary

- Occupancy grids are simple to implement and view.
- Occupancy grids provide a dense representation of the environment.
- Occupancy grids provide an explicit representation of occupied and free space, useful for path planning.
- Occupancy grids can be created with a prior map, say a floorplan.

- A rectangular grid is not an efficient representation for non-rectangular environments.
- There is a trade-off between grid resolution and computational complexity.
- Graph-based maps are efficient and compact.
- Loop-closure is difficult for grid-based maps.
- Loop-closure is automatic for graph-based maps.

9 *Exercises*

1. What is the difference between a feature-based and a location-based map?
2. What is the difference between a grid map and a topological map?
3. If an occupancy grid cell has a value of 0.8, what would you concur?
4. If an occupancy grid cell has a value of 0.5, what would you concur?
5. How is an occupancy grid obtained?
6. If a robot has not viewed a region in the occupancy grid, what cell values do you expect there?
7. What are log odds?
8. How should maps from different sensors be combined?
9. What are short-term maps primarily used for?
10. Why is loop closure difficult with a grid map?

14 Localisation

Localisation is a position estimation problem. The task is to establish a correspondence between a map (described in a global coordinate system) with the robot's local coordinate system.

The pose of a robot cannot be sensed directly[1]; instead it must be inferred directly. Due to ambiguities, it requires the integration of multiple measurements. For example, consider being a robot in the corridor of a local hospital. How would you determine which wing your were in or on which floor you were on?

[1] There is no noise-free pose sensor.

The goal of localisation is to estimate the robot pose given a map and sensor measurements. This can be expressed as a conditional probability[2],

$$p(\mathbf{x}_t|\mathbf{z}_{0:t}, \mathbf{u}_{0:t}, \mathbf{M}), \tag{14.1}$$

[2] This defines the probability of having a specified pose, given a map and the measurement and control history.

where $\mathbf{M}$ denotes the map. Prior information is incorporated using a Bayes filter. Note, the conditional probability for the measurements also includes the map, $p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{M})$.

Localisation assumes that the robot's pose is known but, in practice, this needs to be estimated as well (simultaneous mapping and localisation, SLAM).

1 Localisation problems

There are a number of localisation problems, including:

local versus global localisation

position tracking — assumes the initial pose is known

global localisation — the initial pose is unknown

kidnapped robot problem — variant of global localisation where a robot may be repositioned

static versus dynamic environments

passive versus active sensing

multiple robot mapping — teams of robots can bias each others belief if knowledge of the relative location of the robots is known

2 *Features*

Features are a useful data reduction technique for localisation. Millions of range sensor measurements can be condensed to a few hundred features.

Localisation algorithms rely on features that can be used as *landmarks*. Features usually correspond to distinct objects in the physical world; for example, doors, windows, tree trunks, lamp-posts.

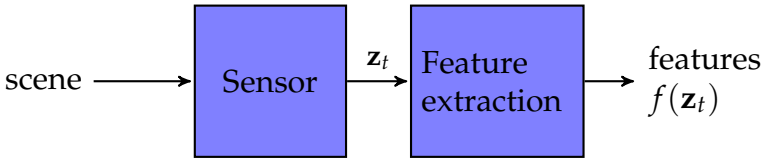


Figure 14.1: Feature extraction from measurement data, $\mathbf{z}_t$. Ideally, the features correspond to landmarks in the scene.

The features resulting from a measurement, $\mathbf{z}_t$, at time step t can be represented by[3]

$$f(\mathbf{z}_t) = \left(\mathbf{f}_t^0, \mathbf{f}_t^1, \mathbf{f}_t^2, \cdots\right)^T, \tag{14.2}$$

where

$$\mathbf{f}_t^i = \left(r_t^i, \phi_t^i, s_t^i\right)^T. \tag{14.3}$$

Here an observed feature is parameterised by a range, r, and bearing ϕ, from the robot heading. It may also consist of a signature s; this may be as simple scalar, such as the average colour, or a complicated descriptor vector (such as used by computer vision feature detectors, SIFT, SURF, etc).

The number of features identifiable at each time step is variable. Usually the features are considered to have independent measurement errors so that can be processed individually.

Whereas observed features are usually parameterised in terms of range, bearing, and signature, the features in the map consist of a location coordinate, (x, y), and a signature, s.

[3] In this section f refers to a feature and not a probability density.

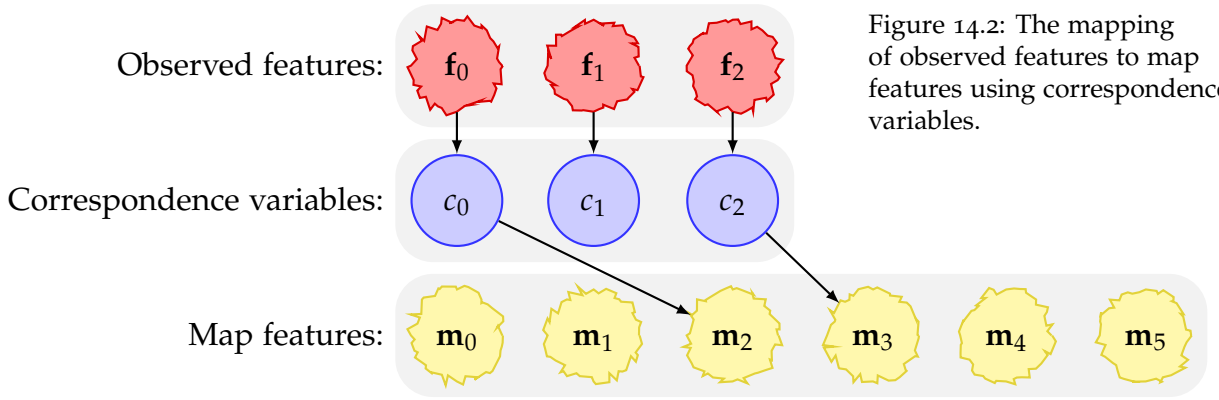


Figure 14.2: The mapping of observed features to map features using correspondence variables.

2.1 *Correspondence variables*

During localisation, features are associated with features in a feature-based map. The map features usually use a Cartesian coordinate in addition to a signature. The mapping between an observed feature, $\mathbf{f}_t^i$, and a map entry, $\mathbf{m}_j$, in a feature-based map is performed using correspondence variables[4], $c_t^i = j$.

The measurement model using features is

$$p(f(\mathbf{z}_t)|\mathbf{x}_t, \mathbf{c}_t, \mathbf{M}), \tag{14.4}$$

where $\mathbf{c}_t$ is a vector of correspondence variables. In practice, the feature measurement probabilities are assumed independent, so

$$p(f(\mathbf{z}_t)|\mathbf{x}_t, \mathbf{c}_t, \mathbf{M}) = \prod_i p(f(\mathbf{z}_t)^i|\mathbf{x}_t, c_t^i, \mathbf{M}). \tag{14.5}$$

[4] A negative value, such as -1, can be used to indicated there is no correspondence to a feature in the map.

2.2 *Feature likelihood model*

The probability that an observed feature matches a map feature is usually decomposed into a product of three distributions:

$$p(f(\mathbf{z}_t)^i|\mathbf{x}_t, c_t^i, \mathbf{M}) \approx f_1\left(\tilde{r}_t^j - r_t^i\right) f_2\left(\tilde{\phi}_t^j - \phi_t^i\right) f_3\left(\tilde{s}_t^j - s_t^i\right). \tag{14.6}$$

In other words, the probability depends on how close the measured range, bearing, and signature of the i^{th} observed feature match the expected range, bearing, and signature for the j^{th} map feature. The expected range and bearing of the j^{th} map feature from the current robot pose $\mathbf{x}_t = (x_t, y_t, \theta_t)$ are:

$$\tilde{r}_t^j = \sqrt{(x^j - x_t)^2 + (y^j - y_t)^2}, \tag{14.7}$$

$$\tilde{\phi}_t^j = \tan^{-1}\frac{y^j - y_t}{x^j - x_t} - \theta_t. \tag{14.8}$$

Note, $j = c_t^i$.

Not surprisingly, the feature likelihood model, $p(f(\mathbf{z}_t)^i | \mathbf{x}_t, c_t^i, \mathbf{M})$, is non-linear and the noise is not additive Gaussian!

2.3 *Feature matching*

The simplest localisation algorithms assume all features are uniquely identifiable. However, many features look the same, so more sophisticated algorithms use maximum likelihood (ML) to estimate the correspondences[5],

$$\hat{\mathbf{c}}_t = \arg\max_{\mathbf{c}_t} p(f(\mathbf{z}_t) | \mathbf{c}_{0:t-1}, \mathbf{c}_t, f(\mathbf{z}_{0:t-1}), \mathbf{u}_{0:t}, \mathbf{M}). \quad (14.9)$$

This estimate is conditioned on the prior correspondences, $\mathbf{c}_{0:t-1}$, and thus the algorithm assumes they are correct. This allows the filter to be updated incrementally but it can make the estimates diverge.

If the maximum likelihood is below a threshold, then no correspondence is assumed.

[5] The arg max operator finds the value of $\mathbf{c}_t$ that maximises the likelihood.

3 *Localisation algorithms*

There are many approaches to the localisation problem. The algorithms can be categorised by:

- The map representation: grids or graphs.
- The use of features or dense measurements.
- The measurement noise model: Gaussian or a general distribution.
- The representation of the posterior pose distribution: Gaussians, histograms, or particles.

Most localisation algorithms employ the *Markov assumption* (complete state assumption) so that only the previous state of the robot is required and not the entire history. For robot localisation, the Markov assumption is a good approximation but fails if a measurement bias is introduced, say if the robot crashes or a wheel gets worn.

The advantage of having a multimodal pose posterior is that multiple hypotheses are supported. Thus a robot can recover from ambiguous situations, say when it crashes into something and the odometry is lost.

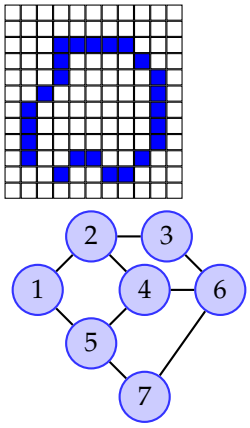


Figure 14.3: Location-based maps can be represented by grids or graphs.

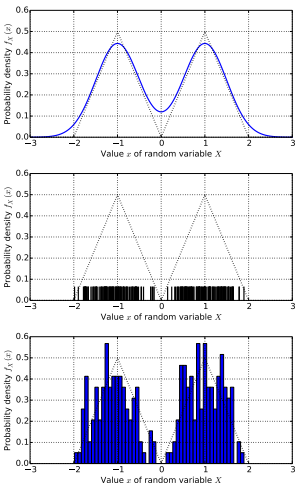


Figure 14.4: Posteriors can be represented mixtures of Gaussians, particles, and histograms.

3.1 *EKF localisation*

EKF localisation algorithms use a continuous representation of the robot pose (x, y, θ) and linearised motion and sensor models. They are precise and efficient but need to start with an initially known position.

The disadvantage of EKF localisation is that its pose beliefs are unimodal (single hypothesis). The multiple hypothesis tracking (MHT) algorithm overcomes this using a mixture of Gaussians to support multiple hypotheses[6].

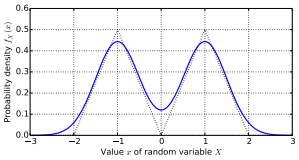


Figure 14.5: Posterior represented by a mixture of Gaussians.

[6] Analogous to multiple hypothesis EKF (MHEKF).

3.2 *Monte-Carlo localisation (MCL)*

Monte-Carlo localisation (MCL) represents the pose posterior using particles. MCL is popular since it is easy to implement and it can approximate nearly any distribution[7].

By adding random particles, the kidnapped robot problem can be solved. However, many additional particles are required and this slows the particle filter.

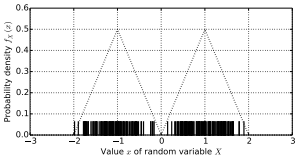


Figure 14.6: Posterior represented by particles.

[7] AMCL (Adapative Monte-Carlo Localisation) uses KLD sampling.

3.3 *Grid localisation*

Grid localisation uses a histogram filter so multiple hypotheses can be tracked. Usually a fixed partitioning of pose space is employed, typically, 15 cm $\times$ 15 cm for location and 5° for heading. A finer representation yields better results but is slower.

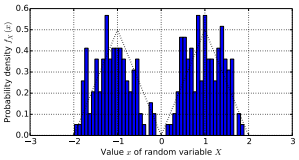


Figure 14.7: Posterior represented by a histogram.

4 *Summary*

1. Localisation involves estimating the robot's pose, given a motion model, a sensor model, control outputs, sensor measurements, and a map.
2. There are many localisation algorithms, based on different Bayes filters to represent the posterior pose distribution, e.g., EKF, histogram filter, particle filter.
3. Features greatly reduce the quantity of measurement data to process. However, a lot of information is lost using features[8].
4. As computers have become faster, there has been less reliance on features for localisation. Instead, dense measurements are used for state-of-the-art localisation algorithms.

[8] You can probably recognise where you are by looking around. However, if you only looked at the locations of doors, your localisation will be more ambiguous.

5. Position tracking requires a known initial pose.

5 *Exercises*

1. What sort of Bayesian filters are commonly used for localisation?
2. What is the difference between position tracking and global localisation?
3. What is the kidnapped robot problem?
4. What is the relationship of a landmark to a feature?
5. What is a correspondence variable?
6. How are features usually parameterised?
7. What is the essential difference between a sensor feature and a map feature?
8. What is a feature likelihood model?
9. How are features matched?
10. What is the MCL algorithm?
11. What is a disadvantage of using features compared to dense sensor measurements?
12. What is an advantage of using features compared to dense sensor measurements?

15 SLAM

Simultaneous localisation and mapping (SLAM) is difficult to solve in practice, since an unbiased map is required for localisation, while an accurate pose estimation is needed to build a map. Moreover, errors in the map will propagate into the localisation estimate and vice-versa.

1 SLAM problems

There are two variants of SLAM:

online SLAM problem This involves estimating the posterior of the current robot pose along with the map,

$$p(\mathbf{x}_t, \mathbf{M} | \mathbf{z}_{0:t}, \mathbf{u}_{0:t}), \tag{15.1}$$

given all the past measurements and controls. This is called online SLAM since it only estimates variables that persist at time step t. Note, many algorithms assume the complete state (Markov) approximation and so discard past measurements and controls.

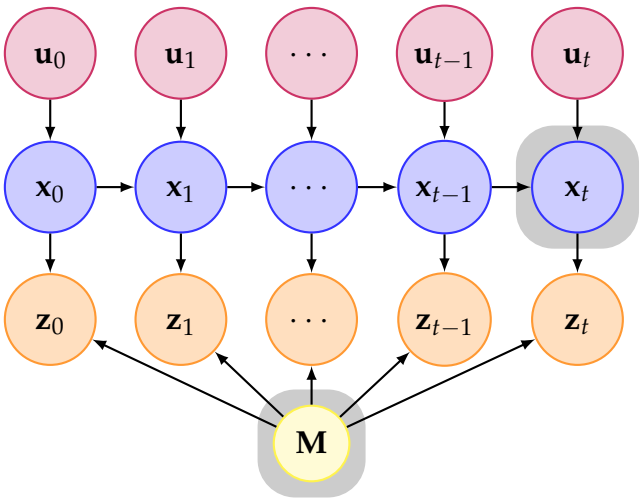


Figure 15.1: Graphical model of the online SLAM problem. The goal is to estimate the posterior of the current robot pose along with the map.

full SLAM problem This involves estimating the posterior over the entire pose history[1], $\mathbf{x}_{0:t}$, along with the map,

$$p(\mathbf{x}_{0:t}, \mathbf{M} | \mathbf{z}_{0:t}, \mathbf{u}_{0:t}). \tag{15.2}$$

[1] In other words, at each time step all the previous pose beliefs are re-estimated using the updated MAP.

The online SLAM problem can be obtained from the full SLAM problem by integrating over past poses,

$$p(\mathbf{x}_t, \mathbf{M} | \mathbf{z}_{0:t}, \mathbf{u}_{0:t}) = \int \int \cdots \int p(\mathbf{x}_{0:t}, \mathbf{M} | \mathbf{z}_{0:t}, \mathbf{u}_{0:t}) d\mathbf{x}_0 d\mathbf{x}_1 \cdots d\mathbf{x}_2. \tag{15.3}$$

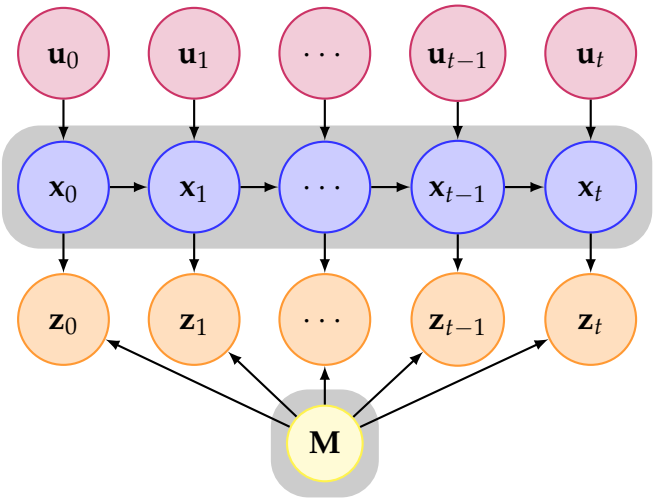


Figure 15.2: Graphical model of the full SLAM problem. The goal is to estimate the posterior of **all** robot poses along with the map.

2 *Bayes filter*

The Bayes filter for the SLAM problem[2] is:

$$p(\mathbf{x}_t, \mathbf{M}_t | \mathbf{z}_{0:t}, \mathbf{u}_{0:t}) = \eta p(\mathbf{z}_t | \mathbf{x}_t, \mathbf{M}_t) \times \int p(\mathbf{x}_t | \mathbf{u}_t, \mathbf{x}_{t-1}) p(\mathbf{x}_{t-1}, \mathbf{M}_t | \mathbf{z}_{0:t-1}, \mathbf{u}_{0:t-1}) d\mathbf{x}_{t-1}. \tag{15.4}$$

[2] This formulation assumes dense sensor measurements and not features. For features, replace $\mathbf{z}_{0:t}$ with $f(\mathbf{z}_{0:t})$ and add a correspondence vector history, $\mathbf{c}_{0:t}$.

The complete state assumption simplifies the Bayes filter to

$$p(\mathbf{x}_t, \mathbf{M}_t | \mathbf{z}_{0:t}, \mathbf{u}_{0:t}) = \eta p(\mathbf{z}_t | \mathbf{x}_t, \mathbf{M}_t) \int p(\mathbf{x}_t | \mathbf{u}_t, \mathbf{x}_{t-1}, \mathbf{M}_t) d\mathbf{x}_{t-1}. \tag{15.5}$$

However, even with this approximation, the posterior is intractable since it involves a probability distribution over a continuous space.

3 SLAM algorithms

Like localisation, there is a plethora of SLAM algorithms, due to the choice of map representation, choice of posterior representation for the pose and map, sensor noise model, etc.

The chief difficulty of applying a Bayes filter to the SLAM problem is the dimensionality of the state space; not only does it need to track the robot pose, it also needs to track the state of the environment.

Using a particle filter for both the pose and environment is infeasible since the required number of particles grows exponentially with the dimension of the state space. To reduce the dimensionality of the problem, two popular SLAM algorithms, FastSLAM and Gmapping, use Rao-Blackwellized particle filters.

Rao-Blackwellized particle filters (RBPFs) exploit the idea that knowledge of the robot's true path allows landmark positions to be conditionally independent. Thus the posterior distribution can be factorised. A particle filter can then be used to track the robot's pose and an EKF can be employed to track each map feature.

4 Summary

1. Online SLAM estimates $p(\mathbf{x}_t, \mathbf{M} | \mathbf{z}_{0:t}, \mathbf{u}_{0:t})$.
2. Full SLAM estimates $p(\mathbf{x}_{0:t}, \mathbf{M} | \mathbf{z}_{0:t}, \mathbf{u}_{0:t})$.
3. Mapping estimates $p(\mathbf{M} | \mathbf{x}_{0:t}, \mathbf{z}_{0:t}, \mathbf{u}_{0:t})$.
4. Localisation estimates $p(\mathbf{x}_t | \mathbf{z}_{0:t}, \mathbf{u}_{0:t}, \mathbf{M})$.
5. Many SLAM algorithms use a particle filter to estimate the pose and EKFs to estimate the map features.

5 Exercises

1. Why is SLAM difficult?
2. Why do SLAM algorithms often use a particle filter for the robot's pose?
3. Why do SLAM algorithms not use a particle filter for the map?
4. What is the difference between the online and full SLAM algorithms?

5. What Bayes filtering is used by the gmapping algorithm?

16 Navigation

There are many algorithms for robot navigation. There are two types:

Local navigation applies to navigation over short distances, often less than the maximum range of the obstacle detecting sensor(s). The primary application is obstacle avoidance. Current sensor data is used and thus it handles dynamic environments.

Global navigation is also known as *path planning*. It is employed for travelling between two locations and requires the use of a map.

Some navigation algorithms are probabilistic; others are deterministic.

Most robots use a set of navigation algorithms for motion planning. These execute at different rates, global path planners (e.g. A*, 0.1 Hz), mid-level path deformation (e.g. elastic band, 5 Hz), and collision / obstacle avoidance algorithms (20 Hz).

1 Local navigation algorithms

Local navigation is usually reactive. There are three common approaches:

Potential-field based where each obstacle has an obstacle 'force field' for repelling the robot, and the goal has a attraction field[1]

Dynamics based where the algorithm consider the robot's dynamics in calculating a solution. (e.g., Velocity Obstacles, Dynamic Window Approach, and Trajectory Rollout).

Sampling based where various collision free states are sampled and then combined, (e.g., Reachability graph, Probabilistic roadmaps).

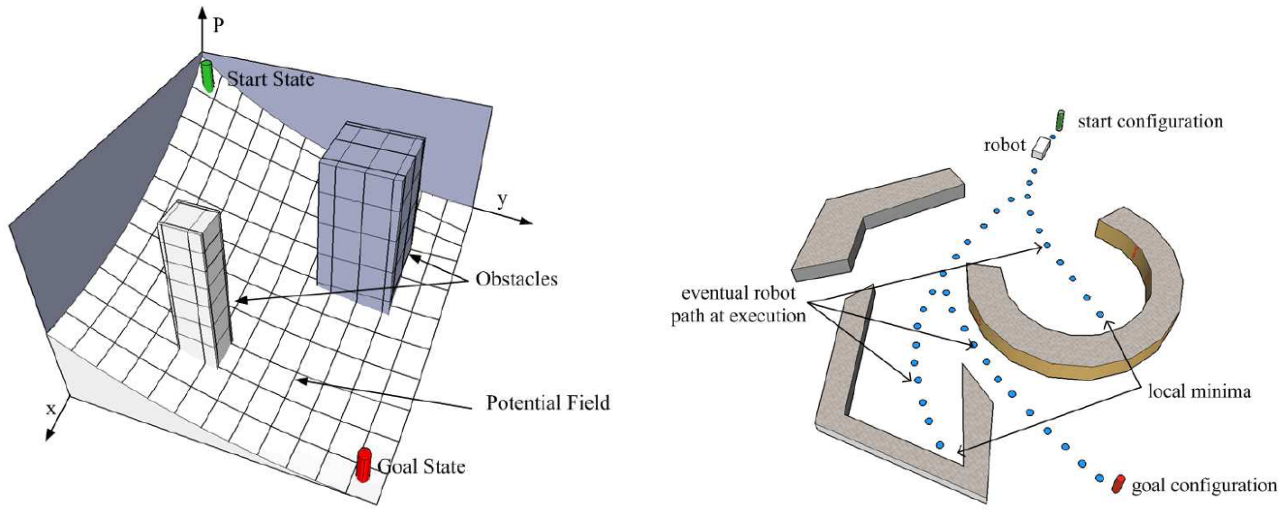
[1] A similar approach is Vector-field histograms (VFH) and Virtual Force Field. VFH also considers dynamics.

1.1 *Potential fields*

This models the robot as a charged particle in a potential field. Goals provide an attractive force; obstacles provide repulsive forces. The robot tries to move from a high potential at the starting point to the low potential at the goal (see Figure 16.1(a)).

Potential fields are simple to implement but:

- A robot can get stuck in a local minima due to U-shaped obstacles, see Figure 16.1(b).
- A robot struggles to pass through narrow gaps (the large contribution to the resultant force by the obstacles can make the robot reverse).
- A robot can oscillate in the presence of obstacles and in narrow passages.



(a) (b)

Figure 16.1: (from S. Petti, PhD thesis, 2007).

1.2 *Vector field histogram (VFH)*

This is one of the most popular algorithms. It achieves real-time obstacle avoidance by maintaining a 2-D histogram around the robot using range-bearing sensors such as a laser scanner. This provides a statistical representation of obstacles and is useful for inaccurate sensor data and fusion of multiple sensor readings.

There are three steps:

1. For each new sensor reading, only the 2-D histogram cell at the measured range and bearing is updated[2].

[2] Unlike with the creation of an occupancy grid where the data is projected to find free cells.

2. The 2-D histogram is transformed into a 1-D polar histogram, where the magnitude of each sector represents a measure of obstacle density in that direction.

3. The polar histogram is then smoothed and thresholded; the resulting steering angle is found by finding an appropriate 'valley' of the polar histogram with which the robot can be aligned.

One of the benefits of VFH is that it is relatively robust to bad sensor measurements because readings are averaged out in the data reduction processes. Another key advantage with VFH is its low computational requirement. However, the VFH method requires a manual tuning of the threshold value (which decides whether a sector in the polar histogram is a valley).

There are a number of extensions to the VFH algorithm, including VFH+ and VFH*. VFH+ incorporates the robot dynamics and uses a four-stage data reduction process to smooth the trajectory. VFH* uses the A* algorithm to avoid dead-end paths.

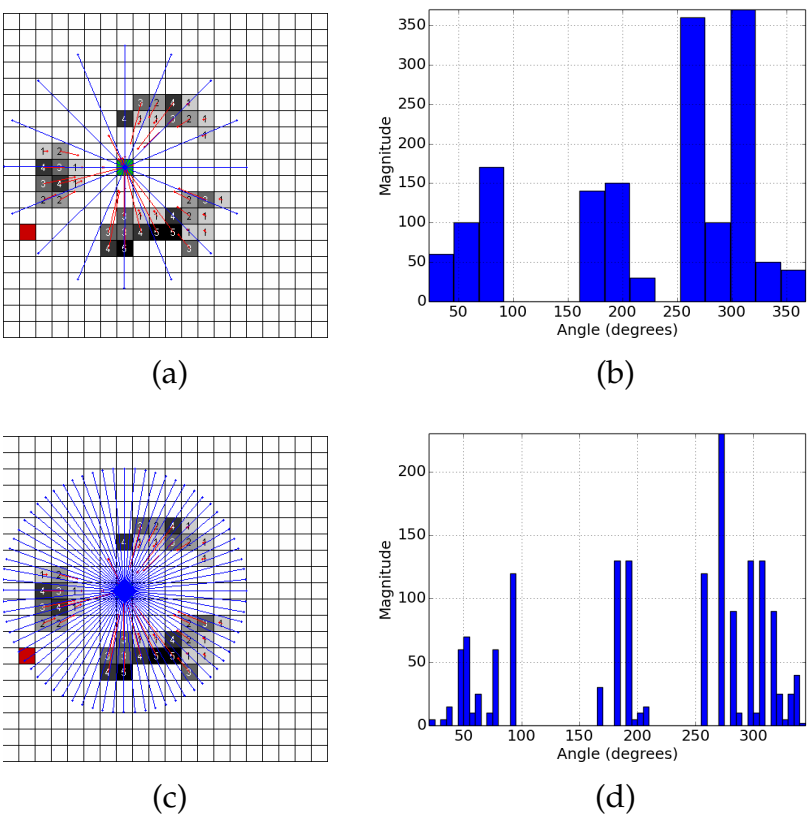


Figure 16.2: Conversion of Cartesian histogram to polar histogram for VFH: (a), (b) 22.5° sampling; (c), (d) 5° sampling. The green square denotes the robot; the red square the goal.

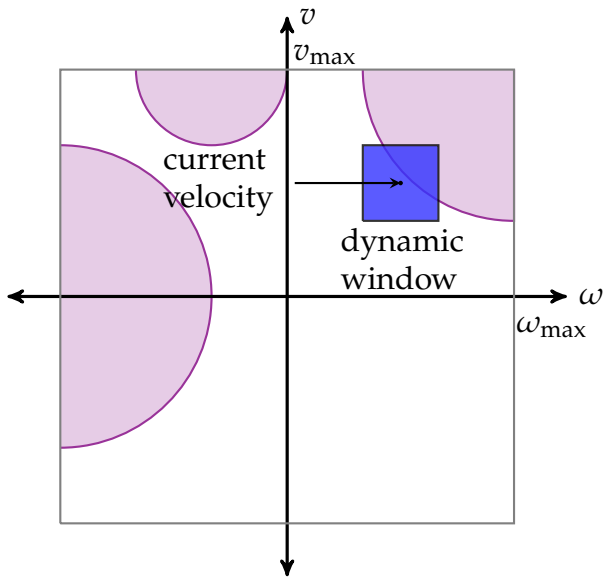


Figure 16.3: Dynamic window approach for a non-holonomic robot with differential drive. The blue box denotes the dynamic window surrounding the current velocity vector. The outer box denotes the limits of the robot's velocity. The violet regions indicate inadmissible velocities that will result in obstacle crashes.

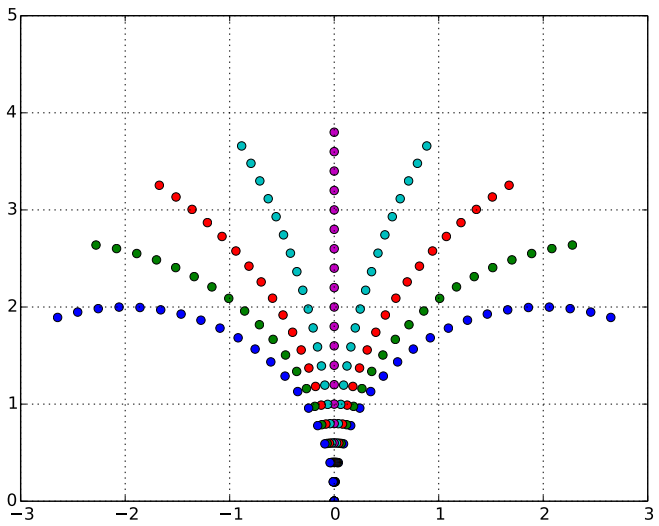


Figure 16.4: Path trajectories for $v = 2\,\text{m/s}$ and ω =-1, -0.75, -0.5, -0.25, 0.25, 0.5, 0.71, and 1 rad/s.

1.3 *Dynamic window approach (DWA)*

Unlike VFH and other local navigation techniques that use distances to obstacles as their main input, the dynamic window approach (DWA) operates directly in the velocity space. It has two key steps:

1. Generate a valid search space by finding the (translational and rotational) velocities that lead to a safe trajectory in a short time interval without collision.
2. Maximize an objective function so that the chosen velocities moves the robot to the goal with maximum clearance from any obstacle.

DWA reduces the search space for possible velocities with the following criteria:

1. Dynamic window: the velocity pairs must be possible to achieve within a short time interval, given the acceleration limits of the robot[3].
2. Admissible velocities: only safe velocities are considered, i.e., velocities which allow the robot to stop before it reaches the closest obstacle on a trajectory.

Once the search space for the velocities has been generated, the most suitable velocity is selected that maximizes the objective function, a weighted sum of three components:

1. Heading to the target.
2. Clearance to the closest obstacle.
3. Forward velocity of the robot.

The Dynamic Window Approach (DWA) algorithm can be summarised as (see Listing 16.1):

1. Discretely sample in the robot's control space (v, ω)[4].
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed.
4. Discard illegal trajectories (those that collide with obstacles).
5. Pick the highest-scoring trajectory and send the associated velocity to the robot.

[3] If the time step is Δt, the current linear speed is v, and the maximum linear acceleration is a, then the dynamic window for the linear speed is set by the range $(v - a\Delta t, v + a\Delta t)$. Similarly, if the current angular speed is ω, and the maximum angular acceleration is α, then the dynamic window for the angular speed is set by the range $(\omega - \alpha\Delta t, \omega + \alpha\Delta t)$.

[4] For a holonomic robot on a plane, the control space is (v_x, v_y, ω) often expressed as $(\Delta x, \Delta y, \Delta\theta)$ assuming a fixed time interval Δt.

1.4 *Trajectory rollout*

Trajectory rollout is similar to DWA but samples the robot's control space differently. Trajectory Rollout samples from the set of achievable velocities over the entire forward simulation period given the acceleration limits of the robot, while DWA samples from the set of achievable velocities for just one simulation step given the acceleration limits of the robot. This means that DWA is a more efficient algorithm because it samples a smaller space, but it may be outperformed by Trajectory Rollout for robots with low acceleration limits.

2 Summary

1. Local navigation applies to navigation over short distances (in the range of the sensors).
2. The primary application of local navigation is for obstacle avoidance.
3. Local navigation runs at a faster rate, typ. 20 Hz, than global navigation, typ. 0.1 Hz.
4. The DWA algorithm is popular since it operates in the control-space of a robot and uses dynamics.

3 Exercises

1. What is the prime purpose of local navigation?
2. Why is the DWA algorithm popular?
3. Describe how the DWA algorithm operates?
4. Speculate on the speed difference of the DWA algorithm for holonomic and non-holonomic robots.
5. How does the DWA algorithm deal with robot dynamics?
6. Comment on the rates that local and global navigation need to be performed.

```
def DWA(robot, goal_pose, laserscan):
    """Implement DWA for robot trying to achieve goal_pose
    given laserscan data"""

    # Calculate desired speed; fast if far away, slow if close
    desired_v = robot.calculate_v(goal_pose)

    # Find allowable speeds in dynamic window around current speeds
    # based on achievable accelerations
    allowable_v = robot.allowable_v()
    allowable_w  = robot.allowable_w()

    optimal_cost = None

    # Iterate over all possible speeds in dynamic window
    for v in robot.allowable_v:
        for w in allowable_w:

            # Find closest object on trajectory
            obstacle_distance = robot.find_distance(v, w, laserscan)

            # Ignore speed if cannot stop in time
            braking_distance = robot.braking_distance(v)
            if obstacle_distance < braking_distance:
                continue

            heading_error = robot.heading_error(v, w, goal_pose)

            clearance = (obstacle_distance - braking_distance) /
                        (dmax - braking_distance)

            cost = cost_function(heading_error, clearance,
                                 abs(desired_v - v))
            if optimal_cost is None or cost < optimal_cost:
                optimal_v = v
                optimal_w = w
                optimal_cost = cost

    return optimal_v, optimal_w
```

Listing 16.1: Python pseudo-code for DWA algorithm.

17
Path planning

Path planning is also known as global navigation. Due to its combinatorial complexity there are many algorithms with different heuristics.

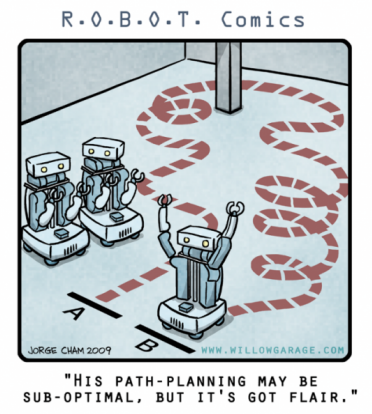


Figure 17.1: Tee hee.

1 *Configuration space*

Before path planning it is common is to grow the obstacles in the map[1] of the environment, a process called *inflation*. This uses the radius of a circle that encompasses the robot[2] (see Figure 17.2). The new map representation is called the *configuration space* (C-space); it represents the legal positions of the robot in the environment. This simplifies path planning since it reduces the robot to a single point.

[1] Usually an occupancy grid.

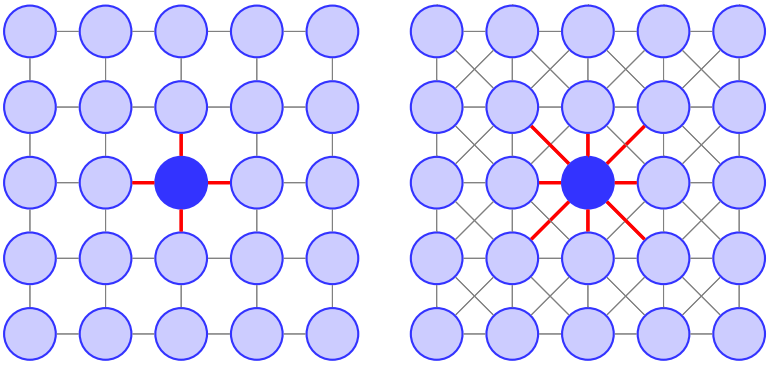
[2] Assuming it can turn in-place.

Figure 17.2: Red cells represent obstacles in the costmap, the blue cells represent obstacles inflated by the inscribed radius of the robot, and the red polygon represents the footprint of the robot. For the robot to avoid collision, the footprint of the robot should never intersect a red cell and the centre point of the robot should never cross a blue cell. From `http://wiki.ros.org/costmap_2d`.

2 Grids and graphs

Global navigation algorithms require a map. Usually this is stored as a grid and treated as a graph with eight-way connectivity.

2.1 Neighbourhoods



(a) (b)

Figure 17.3: A grid treated as a graph: (a) von Neumann neighbourhood, (b) Moore neighbourhood.

There are two ways to connect the cells in a grid:

von Neumann neighbourhood consists of the four neighbours on the faces of a square.

Moore neighbourhood consists of the eight[3] neighbours on the faces and corners of a square.

[3] The number of nearest neighbours for n-D space is $3^n - 1$ I think!

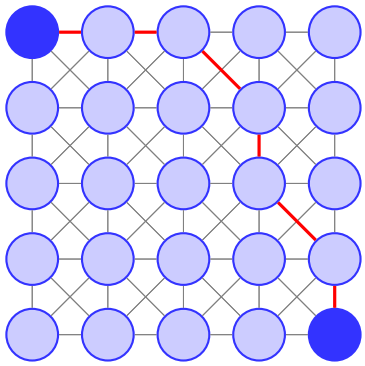


Figure 17.4: One of many possible paths (shown in red) between a pair of nodes using Moore neighbourhoods.

3 Global navigation algorithms

There are many path planning algorithms[4]. Most use graphs.

[4] There's even a book on them!

3.1 Dijkstra path planning

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

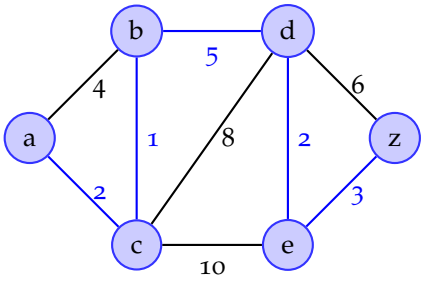


Figure 17.5: Dijkstra's shortest-path example.

3.2 A* path planning

A* is another algorithm for pathfinding and graph traversal. It is a generalization of Dijkstra's algorithm that reduces the size of the subgraph that must be explored, if additional information provides a lower bound on the "distance" to the target.

A* uses heuristics to only expand points it believes to be the most likely to provide the shortest route. Dijkstra's is more likely to produce an optimal path to the goal.

Like all informed search algorithms, A* first searches the routes that appear to be most likely to lead towards the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already travelled into account.

3.3 Wavefront path planning

Wavefront path planning is a specialisation of Dijkstra's algorithm. It considers the configuration space to be like a heat conducting material. Obstacles have zero conductivity, while open space has an infinite conductivity.

The wavefront algorithm expanding wavefronts that emanate from the goal, and thus is a breadth-first search. The cost assigned to a node is essentially the distance from the goal. The search stops when a wavefront has hit the current position, or there are no more nodes to explore (i.e., in this case there is no path). When a wavefront hits the robot, the path to the goal is then found by exploring the node with a lower cost until the goal is reached.

More difficult terrains can be modelled by reducing their conductivity.

3.4 Sampling methods

To simplify the search, sampling methods are employed. The algorithms either assume that the measurements are certain (RRT, PRM) or uncertain (POMDPs).

Rapidly exploring random tree (RRT) an algorithm designed to efficiently search non-convex, high-dimensional spaces by randomly building a space-filling tree[5]. The tree is constructed incrementally from samples drawn randomly from the search space and is inherently biased to grow towards large unsearched areas of the problem. They easily handle problems with obstacles and differential constraints (nonholonomic and kinodynamic).

[5] http://en.wikipedia.org/wiki/Rapidly_exploring_random_tree

Probabilistic roadmaps (PRM) have two phases: a construction and a query phase[6]. In the construction phase a roadmap (graph) is built, approximating the motions that can be made in the environment (using a local planner), using random sampling from the configuration space of the robot. In the query phase, the start and goal configurations are added and a path is found using Dijkstra's algorithm.

[6] http://en.wikipedia.org/wiki/Probabilistic_roadmap

Partially observable Markov decision processes (POMDPs) assume that the system dynamics are determined by a Markov decision process (MDP) but that the underlying state cannot be directly observed. This is a general framework for many path planning algorithms that operate with uncertainty.

Like Bayes filters, POMDPs are difficult to implement efficiently in practice due to the combinatorial nature of the problem.

4 Realtime path deformation

If something alters the planned path, say someone crosses the path, you need to recover the path again. Algorithms such as D* lite allow efficient re-computation. However, the elastic band method allows the existing path to be adjusted to handle deformations. It enables realtime modifications to a precomputed path that considers additional obstacles not considered during the global planning.

5 *Topological graph creation*

A topological map can be created from a grid-based one. This has a number of advantages:

- Faster path planning since the graph is smaller[7].
- The graph can be simplified using pruning.
- Alternative paths can be generated more quickly for dynamic environments.

[7] Large free areas, such as corridors, can be represented by a few nodes.

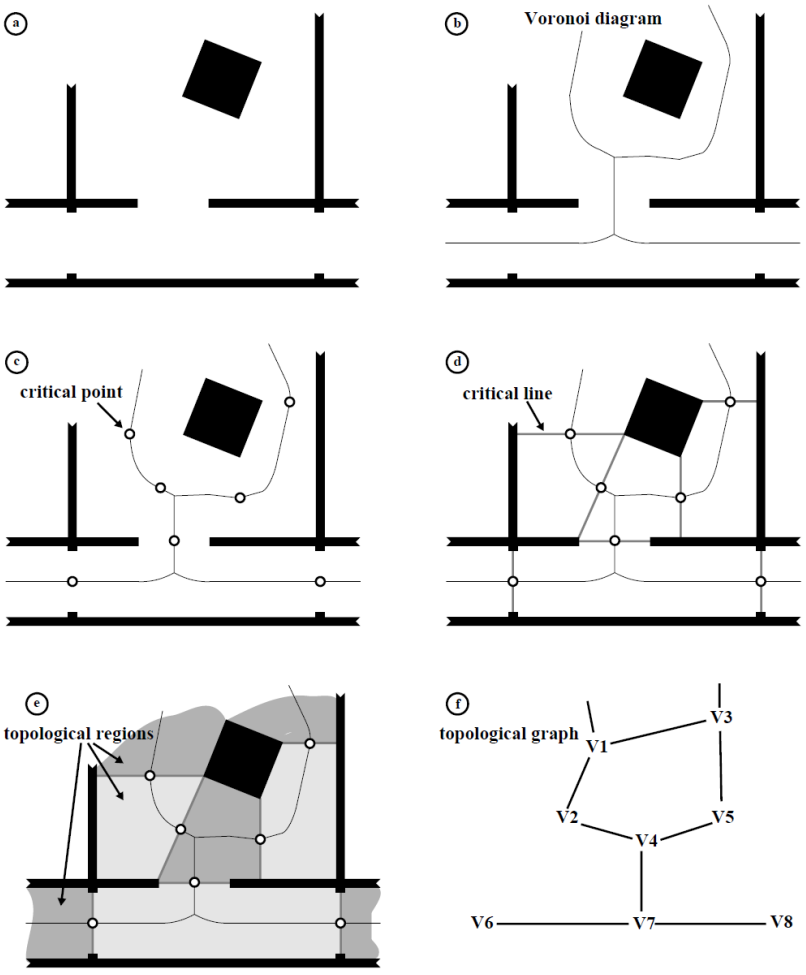


Figure 17.6: Constructing a topological map from an occupancy grid map (from S. Thrun. "Learning maps for indoor mobile robot navigation").

An algorithm to construct a topological map from a grid map is:

1. Threshold each cell in the occupancy grid map.
2. Construct a Voronoi diagram[8].
3. Locate the critical points: these are points that lie on the Voronoi diagram and are local minima with regards to clearances between obstacles (and that with the neighbouring points).

[8] In mathematics, a Voronoi diagram is a way of dividing space into a number of regions. A set of points (called seeds, sites, or generators) is specified beforehand and for each seed there will be a corresponding region consisting of all points closer to that seed than to any other. The regions are called Voronoi cells. It is dual to the Delaunay triangulation.

4. Construct critical lines: these are simply lines extended from each critical point to their nearest obstacle in the thresholded map.
5. Construct topological graph by assigning a node for each region separated by critical lines and obstacles, while the Voronoi diagram specifies how the nodes are connected.

6 ROS navigation

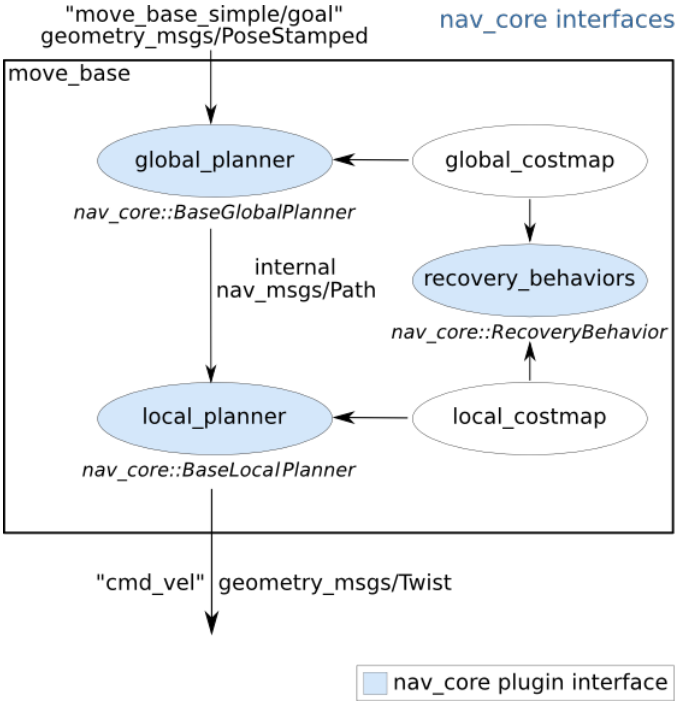


Figure 17.7: Interfaces for the ROS nav_core, from http://wiki.ros.org/nav_core.

ROS provides a navigation node (nav_core) with plugins for a global path planner and a local planner. In addition, it supports recovery behaviours when it gets confused by the sensor data conflicting with the map.

6.1 ROS global planners

There are two global planners used with ROS nav_core:

navfn uses Dijkstra's algorithm on a grid.

carrot_planner takes a goal point and attempts to move the robot as close to it as possible, even when that goal point is in an obstacle[9].

[9] It tries to place carrots in front of the robot for it to follow.

6.2 ROS local planners

ROS has two local planners for navigation on a plane:

DWA local planner implements the Dynamic Window Approach[10] (DWA). This supports holonomic or pseudo-holonomic robots since the control-space is (v_x, v_y, ω).

[10] http://wiki.ros.org/dwa_local_planner

Base local planner implements Trajectory Rollout and the DWA[11] for a control-space (v, ω).

[11] http://wiki.ros.org/base_local_planner

6.3 Recovery behaviours

Recovery behaviours used with ROS nav_core:

clear_costmap_recovery reverts the costmaps used by move_base to the static map outside of a user-specified range.

rotate_recovery performs a 360 degree rotation of the robot to attempt to clear out space.

6.4 Cost maps

The costmap_2d package[12] provides a configurable structure that maintains information about where the robot should navigate in the form of a occupancy grid.

[12] http://wiki.ros.org/costmap_2d

The costmap package automatically subscribes to ROS sensor topics and creates an occupancy grid. It inflates the occupancy grid to create a costmap. There are two costmaps:

local costmap This only uses the sensor data and is used by local navigation.

global costmap This uses the sensor data and the map and is used by by the path planner.

6.5 Complete system

1. Localise robot, say using AMCL, given map.
2. Use global planner, say Dijkstra's algorithm, to produce a path.
3. Use local planner, say DWA, to generate velocity commands to follow path.

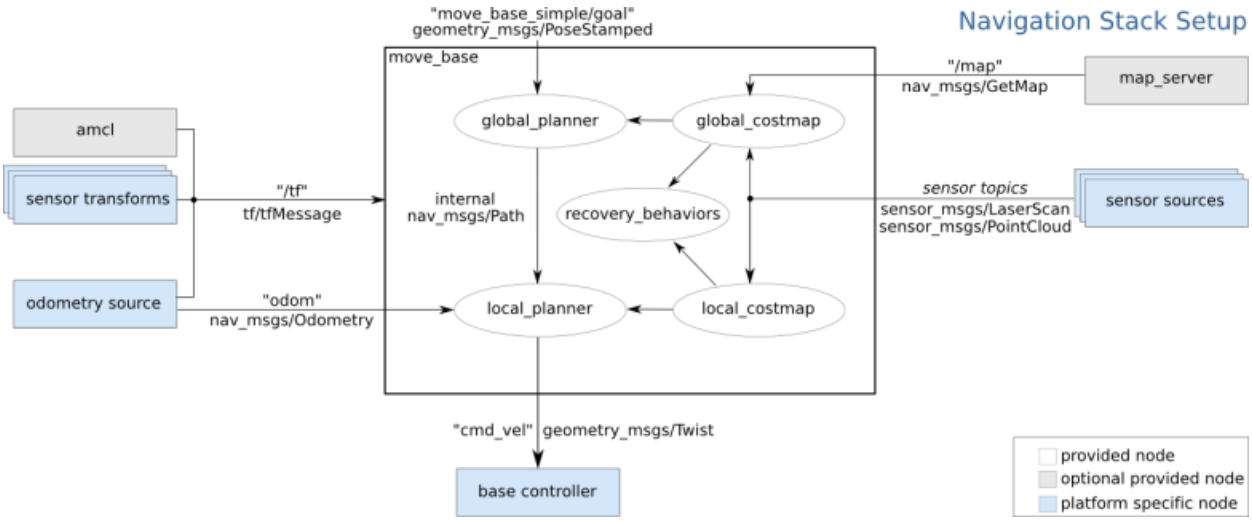


Figure 17.8: ROS move_base_node.

7 *Exercises*

1. How is an occupancy grid treated as a graph?
2. What is the advantage of a topological map for path planning?
3. How can topological maps be created from grid maps?
4. Is the A* algorithm always faster than the Dijkstra algorithm?
5. Why is the A* algorithm faster than the Dijkstra algorithm?
6. Does the A* algorithm always find an optimal solution?
7. Why is it not worthwhile searching for an optimal solution?
8. What is configuration space and what is it used for?
9. What is a recovery behaviour?
10. ROS navigation uses a local costmap and a global costmap. What are the differences?
11. How does a costmap differ from an occupancy grid?

18
Radar and sonar range estimation

Radar and sonar are active[1] ranging devices. They provide an estimate of the range and bearing of a target.

The theory behind radar and sonar is similar but there are many practical differences due to the difference in the waves:

radar use electromagnetic waves and these travel at the speed of light, $c = 3.0 \times 10^8$ m/s.

sonar use acoustic waves and these travel at approximately[2] 1500 m/s in seawater and 340 m/s in air.

With a radar the transmitting and receiving transducers are antennas[3] while with underwater sonar the transmitting transducer is called a projector and the receiving transducer is called a hydrophone.

There are two primary modes of operation:

pulse echo for target range estimation

Doppler for target speed estimation

[1] There are passive devices that listen to the radiations from objects; such as vibrations from water pumps in nuclear submarines.

[2] The speed of acoustic propagation varies with temperature and pressure as well as salinity in the case of water.

[3] Usually microwave antennas.

1 *Pulse echo operation*

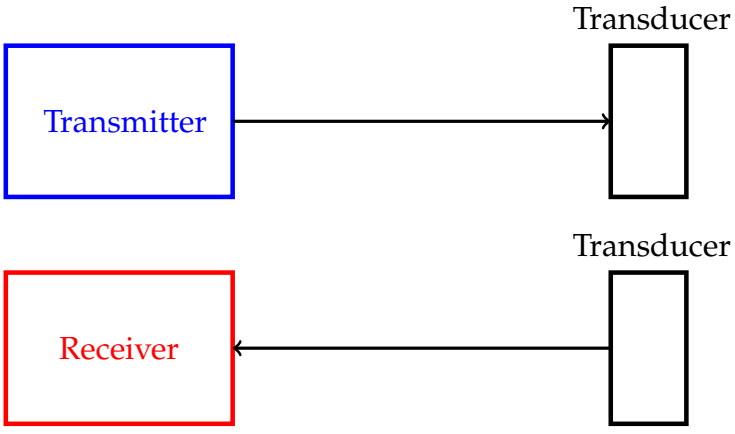


Figure 18.1: Radar with separate transmitting and receiving transducers (antennas).

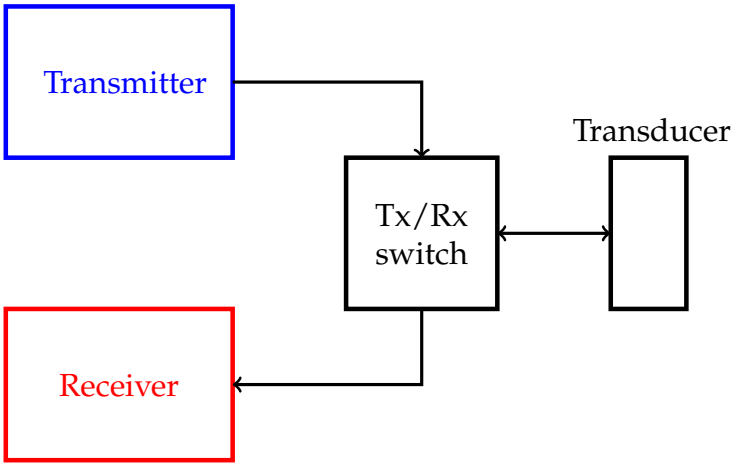


Figure 18.2: Radar with shared transmitting and receiving transducers (antennas) using transmit/receive switch.

Most radars and sonars are designed to estimate the range to a target. They transmit a pulse of energy and time how long it takes for the echo scattered by the target to return.

The echo from a target at a range, r, is delayed by a time, τ, where

$$\tau = \frac{2r}{c}, \tag{18.1}$$

and where c is the wave speed of propagation. Thus estimating range is equivalent to estimating the time delay, assuming that c is known.

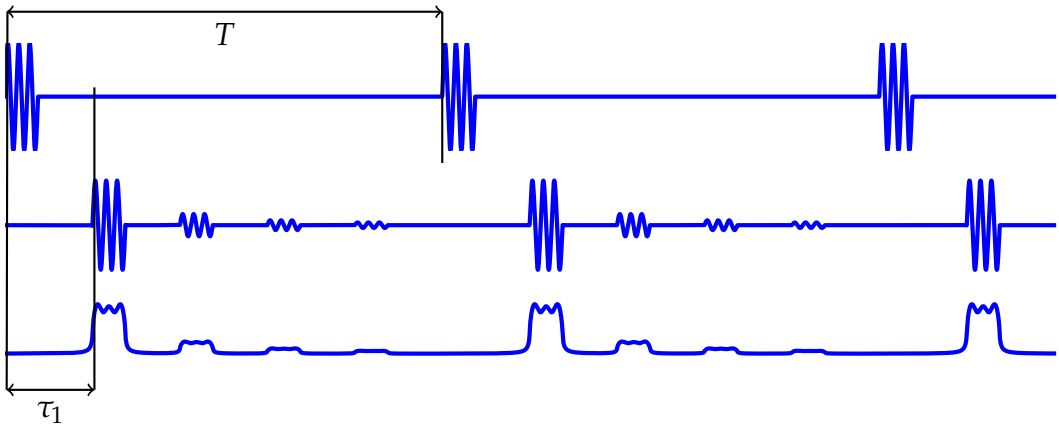


Figure 18.3: Gated CW pulse with a repetition period T, its echoes, and the envelope of the echoes.

If the transmit signal is $s(t)$, the echo signal for a single target, $e(t)$, has the form[4],

$$e(t) = \frac{a}{r^2} s\left(t - \frac{2r}{c}\right), \tag{18.2}$$

where a describes the fraction of the transmit signal that is scattered by the target. The r^2 factor describes the spreading losses[5].

[4] Assuming that nothing is moving that will induce a Doppler shift.

[5] In practice, there are also absorption losses that increase with range, r.

When there are multiple targets, the echo signal is a superposition of scaled and delayed transmit signals:

$$e(t) = \sum_i \frac{a_i}{r_i^2} s\left(t - \frac{2r_i}{c}\right). \tag{18.3}$$

2 Target scattering

A target is a scattering object of interest. There are two types:

specular targets are smooth compared to the wavelength[6]. They act like a mirror and reflect energy strongly in one direction.

diffuse targets are rough compared to the wavelength. They scatter energy weakly in all directions.

[6] Typically if the rms roughness is smaller than $\lambda/4$, the target is specular.

Specular targets can produced much stronger echoes but only if they are oriented normal to the radar or sonar boresight direction.

3 Maximum unambiguous range

Once a pulse has been transmitted it is necessary to wait for all the echoes to arrive before sending the next pulse. Otherwise, the range measurement is ambiguous. If the pulse repetition period is T, the maximum unambiguous range is given by

$$R_u = \frac{cT}{2}. \tag{18.4}$$

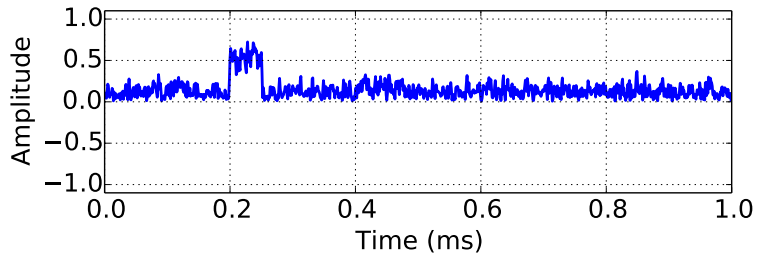
T is the reciprocal of the pulse repetition frequency (prf). The faster the prf, the greater the number of range estimates per second.

4 Time varying gain (TVG)

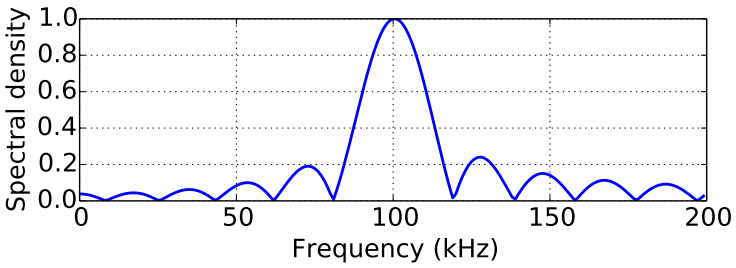
Since the amplitude of the echoes decreases as r^{-2} the receiver tries to compensate using a variable gain amplifier. This has a minimum gain when the pulse is transmitted and then increases quadratically with time. This helps compensate for the spreading losses but also amplifies the noise.

5 *Matched-filtering*

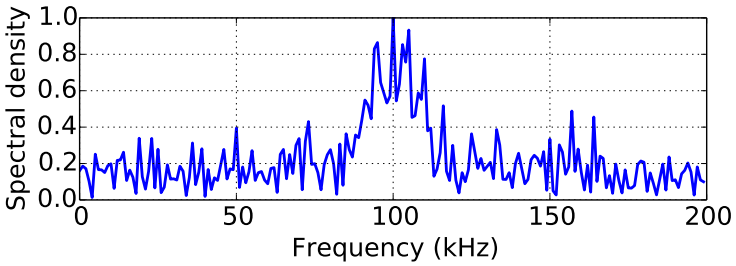
Due to the strong reduction of received power with range, even small noise levels limit the maximum range of operation. To increase the operating range it is necessary to filter noise that is outside the signal bandwidth. The optimum filter is a matched-filter; it is matched to the known transmitted signal.



(a)



(b)



(b)

Figure 18.4: Gated CW pulse: (a) envelope of received echoes with noise (b) transmitted spectrum, (c) echoes spectrum with noise.

Matched-filter is achieved with a correlation[7] of the echo signal, $e(t)$, with the transmitted signal, $s(t)$,

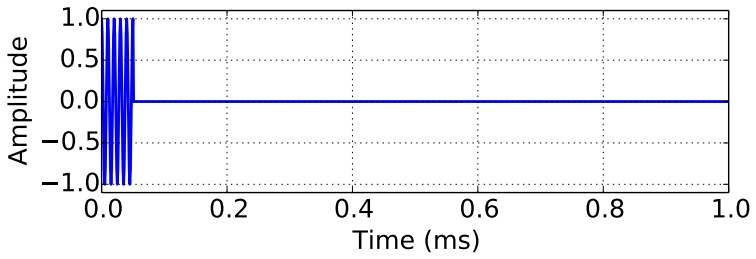
$$d(t) = e(t) \star s(t). \tag{18.5}$$

This can be performed efficiently in the Fourier domain as a multiplication,

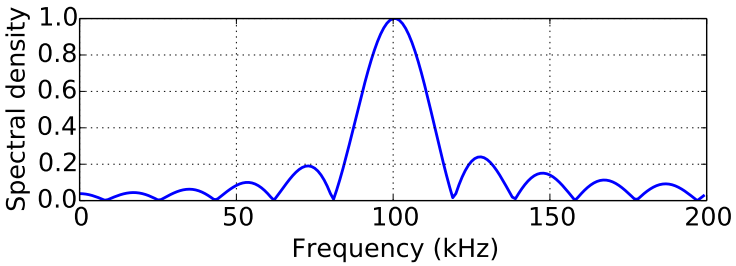
$$D(f) = E(s)S^*(f), \tag{18.6}$$

where the asterisk denotes complex conjugation.

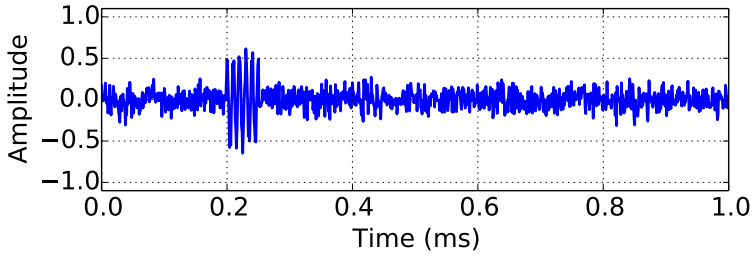
[7] Usually using a digital signal processor (DSP) or specialised hardware.



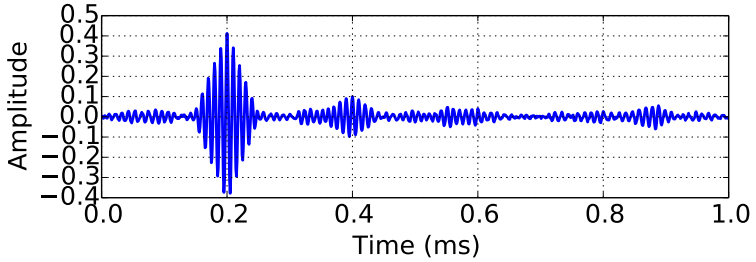
(a)



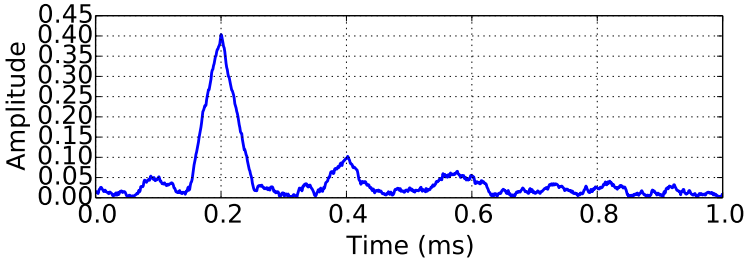
(b)



(c)



(d)



(e)

Figure 18.5: Gated CW pulse: (a) transmit signal, (b) transmit signal spectrum, (c) echoes with noise, (d) pulse-compressed echoes with noise, (e) envelope of pulse-compressed echoes with noise.

6 Range accuracy

The range measurement accuracy depends on the signal to noise ratio (SNR) of the echoes. Clearly, if the noise level is high then the measurements will be uncertain.

The echo signals get contaminated by noise. For a radar this noise is primarily thermal noise generated in the front-end electronics of the receiver[8]. Sonar noise is primarily due to sources in the environment[9].

[8] Cryogenic cooling helps.

[9] Breaking waves, marine creatures, etc.

The range accuracy is a function of the echo signal to noise ratio and the signal bandwidth, B,

$$\sigma_r \approx \frac{c}{B\sqrt{\text{SNR}}}. \tag{18.7}$$

7 Range resolution

The range resolution is a measure of how well a radar or sonar can separate targets at different ranges. It depends on the transmitted signal, how it is processed, and the speed of propagation. Assuming a matched-filter, it is given by

$$\Delta r = \frac{c}{2B}, \tag{18.8}$$

where B is the signal bandwidth.

8 Pulse compression

The amount of power that can be transmitted depends on the signal amplitude. Underwater, the maximum power level is limited by cavitation[10].

[10] This occurs when the acoustic pressure exceeds the static pressure and thus the water starts to boil.

One trick is to use pulse compression where instead of using a short pulse, a much longer pulse is used but with a varying frequency (or some other modulation). When the echo signals are correlated (match-filtered) with a replica of the transmitted signal, the amplitude of the resultant signal is proportional to the transmitted signal energy and its width is inversely proportional to the transmitted signal bandwidth. Thus pulse-compression can retain a high resolution but avoids high peak powers by employing long-duration signals.

Pulse compression is achieved by the matched-filter; correlating a signal with itself generates the autocorrelation of the signal. Thus ideal pulse compression signals have a short-duration autocorrelation. In practice, the

width of the autocorrelation cannot be smaller than the signal bandwith, B.

The common pulse-compression waveforms are linear frequency modulated (FM) chirps and pseudorandom binary sequences. A linear FM chirp can be described mathematically by

$$s(t) = \cos\left(2\pi f_0 t + \pi\mu t_1^2\right) \operatorname{rect}\left(\frac{t_1}{T_p}\right). \tag{18.9}$$

Here f_0 is the centre frequency, T_p is the chirp period, $t_1 = t - T_p/2$, and μ is the chirp rate, given by

$$\mu = \frac{B}{T_p}. \tag{18.10}$$

The instantaneous frequency is

$$f_i(t) = f_0 + \mu\left(t - \frac{T_p}{2}\right), \quad 0 \geq t \geq T_p. \tag{18.11}$$

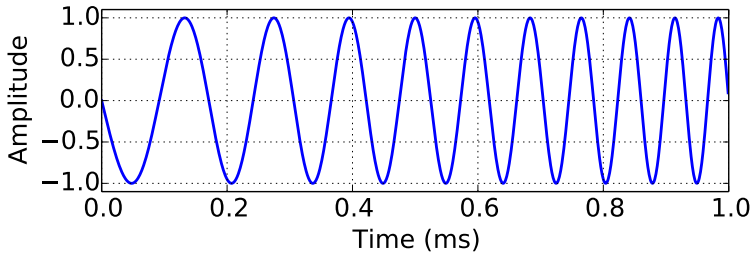


Figure 18.6: Example linear FM chirp signal.

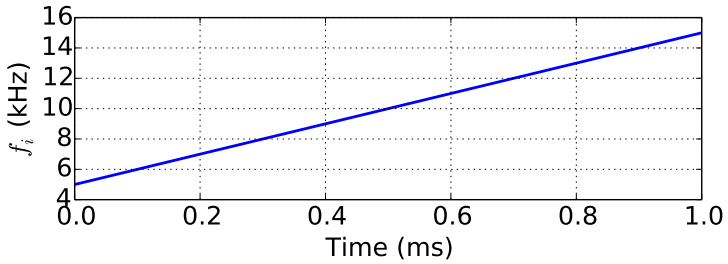
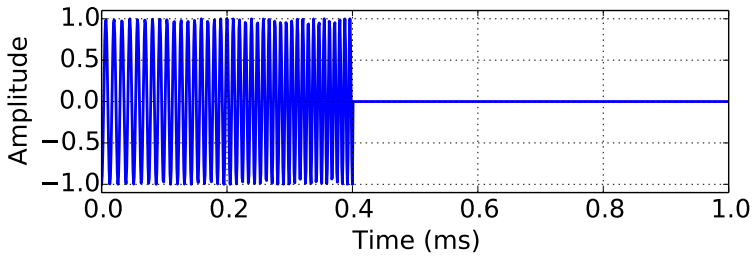
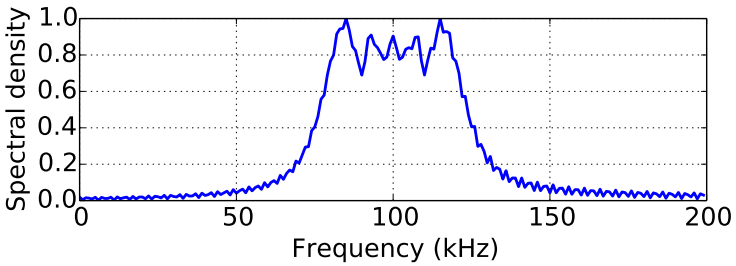


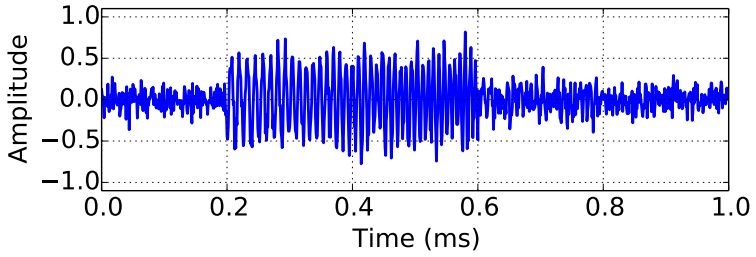
Figure 18.7: Instantaneous frequency of example linear FM chirp signal.



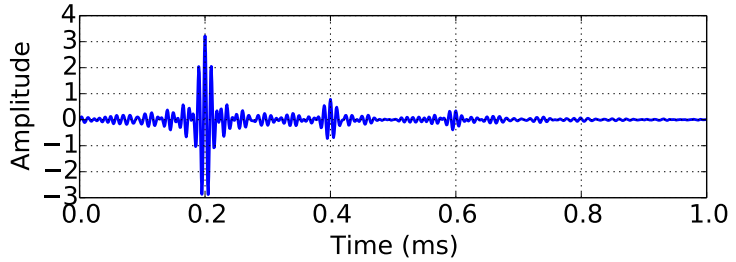
(a)



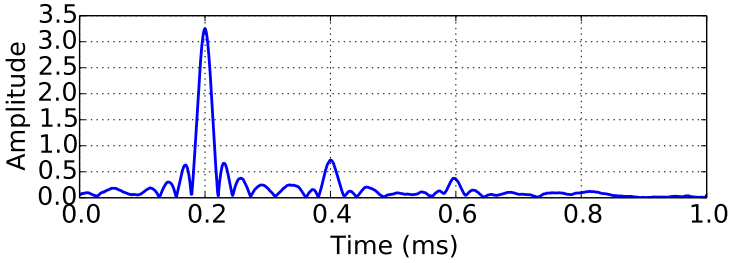
(b)



(c)

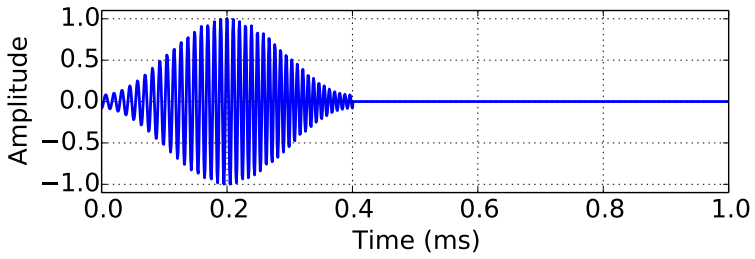


(d)

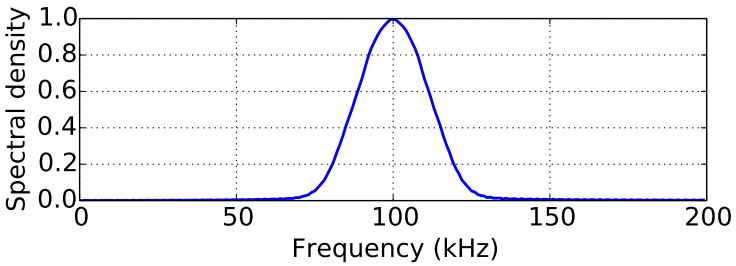


(e)

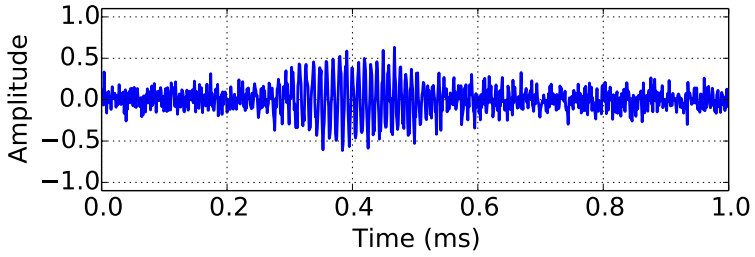
Figure 18.8: Linear FM chirp: (a) transmit signal, (b) transmit signal spectrum, (c) echoes with noise, (d) pulse-compressed echoes with noise, (e) envelope of pulse-compressed echoes with noise.



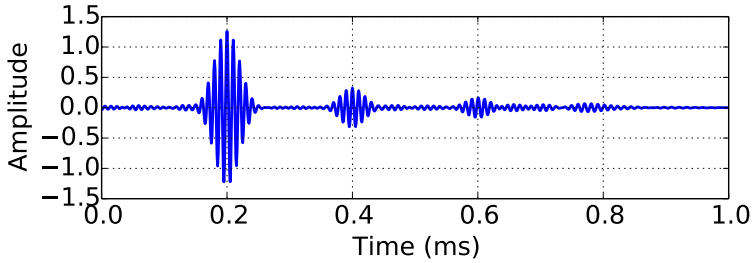
(a)



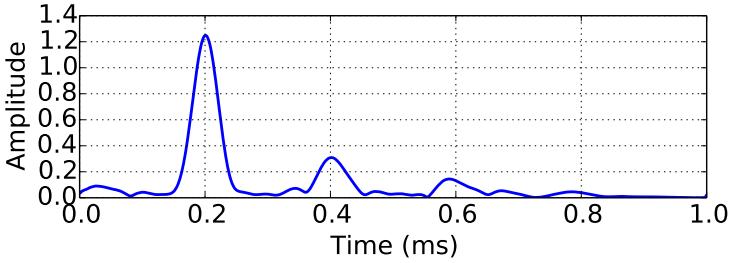
(b)



(c)



(d)



(e)

Figure 18.9: Hamming windowed linear FM chirp: (a) transmit signal, (b) transmit signal spectrum, (c) echoes with noise, (d) pulse-compressed echoes with noise, (e) envelope of pulse-compressed echoes with noise.

9 The radar equation

The echo amplitude drops with the square of range and thus the echo power drops with the fourth power of range; due to the inverse square law, the transmit power drops with the square of range and similarly the returned echo drops with the square of range. The received power level can be modelled by the radar equation[11]:

$$P_r(r,\theta,\phi) = \frac{\lambda^2 P_t G_t G_r B_t(\theta,\phi) B_r(\theta,\phi)}{4\pi^2 r^4}\sigma^2, \qquad (18.12)$$

[11] Assuming a monostatic configuration where the transmitting transducer is close to the receiving transducer.

where

- r is the range to the target,
- r^4 models the spreading losses,
- λ is the wavelength[12],
- P_t is the transmitted power,
- G_t is the gain of the transmitting transducer (proportional to its area),
- G_r is the gain of the receiving transducer (proportional to its area),
- B_t is the beam pattern of the transmitting transducer,
- B_r is the beam pattern of the receiving transducer,
- θ and ϕ are angles from boresight in the horizontal and vertical planes[13],
- σ^2 is the scattering cross-sectional area of the target.

[12] Usually this refers to the wavelength at the centre frequency of operation, $c = f_0\lambda$.

[13] Only in special cases are they equivalent to the spherical polar angles.

The $\lambda^2/4\pi^2$ comes from diffraction theory and helps to keep things dimensionally consistent. There are also absorption losses that are not modelled by this equation.

The received power is not a big deal for a sonar since it is does not need to extract power from the echo. Since the transducer is connected to a high impedance amplifier, the output voltage from the transducer is more important than the current it can deliver. Besides, most of the noise is acoustic and this gets amplified along with the echo. Doubling the size of a hydrophone can capture twice as much power but since most hydrophones respond to pressure (and not force) the output voltage is not increased (the electrical impedance of the transducer is halved and thus it can deliver twice as much current). Nevertheless, even though the voltage is not increased, the noise is halved (assuming it is omni-directional) since the transducer is more directional.

10 Exercises

1. An underwater sonar has a centre frequency of 100 kHz and a bandwidth of 50 kHz. Determine the range resolution. Assume $c = 1500$ m/s.
2. An underwater sonar has a centre frequency of 100 kHz, a bandwidth of 50 kHz, a pulse duration of 1 ms, and a pulse repetition frequency of 10 Hz. Determine the maximum unambiguous range.
3. If a diffuse target at a range of 10 m produces an echo signal amplitude of 1 mV, what is the echo signal amplitude if the target is at a range of 20 m.
4. Compare the time-bandwidth products of a gated CW pulse and a linear FM pulse of the same nominal bandwidths.
5. Compare the echo delay for a target at 10 m for a radar, underwater sonar, and air sonar.
6. A 79 GHz radar has a 600 MHz bandwidth. Determine its range resolution.
7. A 79 GHz radar has a 0.1 m range resolution. Determine its bandwidth.
8. A 79 GHz radar has maximum range of 250 m. Determine the fastest pulse repetition frequency.

19
Radar and sonar bearing estimation

Bearing estimation is the problem of determining from which angle the echo is arriving. If the beam widths of the transducer are narrow then this is essentially the boresight direction.

1 Beam patterns

The field of an aperture can be found from diffraction theory. It is complicated, especially when close to the aperture, but becomes simpler beyond the Rayleigh range,

$$R_r = \frac{2D^2}{\lambda}, \tag{19.1}$$

where D is the maximum extent of the aperture and $\lambda = c/f_0$ is the wavelength at the centre frequency. Beyond the Rayleigh range is called the Fraunhoffer region and the radiation appears to come from a point source. In the Fourier transform the beam pattern[1] is related to a 2-D Fourier transform of the aperture.

[1] At points closer than the Rayleigh range, the concept of beam pattern does not make sense.

Let's consider a transducer with a rectangular aperture of width D_x and height D_y, oriented so the x-axis is horizontal, the y-axis is vertical, and boresight is along the z-axis. The aperture function can be described by

$$a(x,y) = \begin{cases} 1, & -\frac{D_x}{2} < x < \frac{D_x}{2}, -\frac{D_y}{2} < y < \frac{D_y}{2} \\ 0, & \text{otherwise} \end{cases} \tag{19.2}$$

$$= \operatorname{rect}\left(\frac{x}{D_x}\right) \operatorname{rect}\left(\frac{y}{D_y}\right). \tag{19.3}$$

The 2-D Fourier transform is

$$A(f_x, f_y) = D_x D_y \operatorname{sinc}(f_x D_x) \operatorname{sinc}(f_y D_y). \tag{19.4}$$

The beam pattern is obtained from this Fourier transform, normalised, with

$$f_x = \frac{\sin\theta}{\lambda}, \tag{19.5}$$

$$f_y = \frac{\sin\phi}{\lambda}, \tag{19.6}$$

where θ is the horizontal angle and ϕ is the vertical angle, both measured from boresight. Thus, the beam pattern for a rectangular aperture is

$$B(\theta,\phi) = \text{sinc}\left(D_x\frac{\sin\theta}{\lambda}\right)\text{sinc}\left(D_y\frac{\sin\phi}{\lambda}\right). \tag{19.7}$$

Things to note:

1. The beam pattern for a rectangular aperture is separable[2] into horizontal and vertical components.
2. The beam pattern varies with wavelength, λ, and thus frequency f.
3. The beam pattern is narrower at shorter wavelengths (higher frequencies).
4. The beam pattern is narrower for larger apertures.
5. The beam pattern has a main lobe and many side lobes.
6. The approximate 3 dB (half-power) beamwidth for a rectangular aperture is

$$\Delta\theta \approx \frac{\lambda}{D}, \tag{19.8}$$

   where D is the dimension of the transducer in the plane of interest.
7. The larger the aperture of a receiving transducer, the greater the received power and thus the greater the gain[3].

[2] This is not the case for a circular aperture.

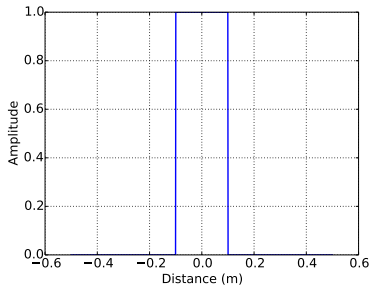
[3] Compared to an isotropic transducer.

The level of the side lobes can be reduced but at the expense of a wider lobe. This is achieved by tapering the response across the aperture so that outside regions of the aperture transmit less energy than the inside regions.

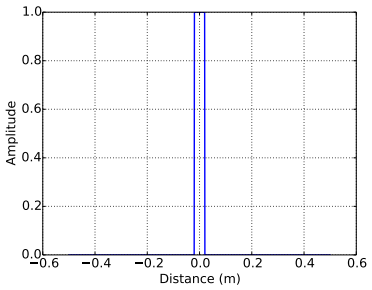
If the horizontal beamwidth is narrow, the horizontal resolution is

$$\Delta x \approx r\Delta\theta, \tag{19.9}$$

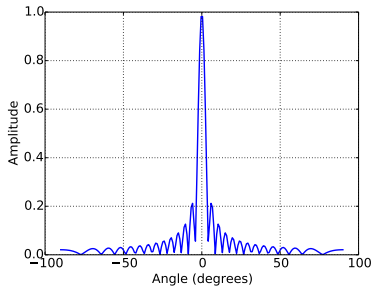
$$\approx r\frac{\lambda}{D_x}. \tag{19.10}$$



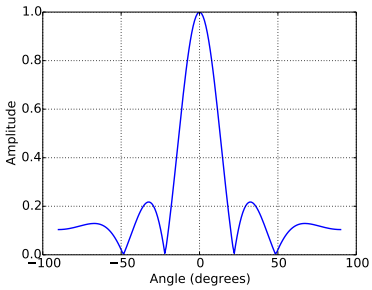
(a)



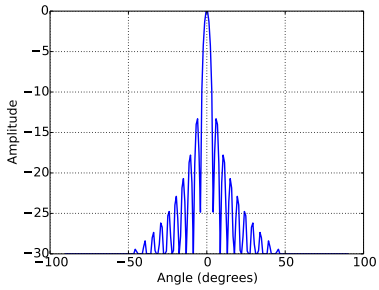
(b)



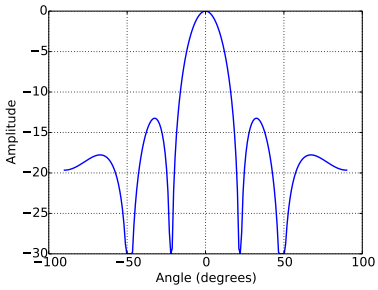
(c)



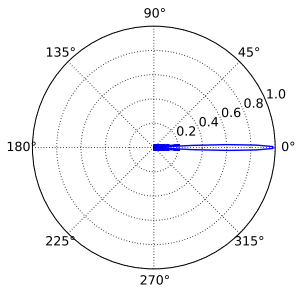
(d)



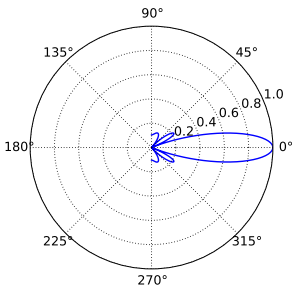
(e)



(f)



(g)



(h)

Figure 19.1: (a,b) Rectangular aperture (c,d) beam pattern (e,f) beam pattern in dB, (g,h) polar beam pattern. Wavelength, $\lambda = 15\,\text{mm}$, aperture width, $W = 0.2\,\text{m}$ (a,c,e,g), $W = 0.04\,\text{m}$ (b,d,f,h) .

2 Beamforming

One approach to narrow the beamwidth is to employ an array of transducers, all cunningly electrically connected together. By appropriately delaying the signals fed to each transmitting transducer, it is possible to steer the transmit beam[4]. It is also possible to focus the beam more tightly[5].

[4] The same process can be performed on reception.
[5] Up to some point before the beam diverges again.

3 Aperture synthesis

Aperture synthesis uses the concept of moving the transducer along a known path and using beamforming techniques to combine the echoes. A synthetic aperture can be many times larger than any physical aperture and thus can achieve higher angular resolution.

4 Monopulse

The bearing can be estimated more precisely if there is only a single scattering object at a given range. One approach is to use a pair of receiving apertures and to compare the amplitude difference. A more accurate approach is to compare the phase difference.

	Long range	Mid range	Short range
Frequency band	77 GHz	79 GHz	79 GHz
Bandwidth	600 MHz	600 MHz	4 GHz
Range	10-250 m	1-100 m	0.15-30 m
Range resolution	0.5 m	0.5 m	0.1 m
Speed resolution	0.1 m/s	0.1 m/s	0.1 m/s
Angular accuracy	0.1°	0.5°	1°
3 dB azimuth beamwidth	±15°	±40°	±80°
3 dB elevation beamwidth	±5°	±5°	±10°

Table 19.1: Comparison of different automotive radar sensor classifications. from "Millimeter-Wave Receiver Concepts for 77 GHz Automotive Radar in Silicon", Dietmar Kissinger.

5 Exercises

1. A radar has a centre frequency of 79 GHz. Determine its wavelength.
2. A radar with a centre frequency of 79 GHz has a rectangular aperture of 10 cm by 5 cm. Determine its angular resolution in each direction.

3. A radar with a centre frequency of 79 GHz has a rectangular aperture with a horizontal dimension of 10 cm. Determine the horizontal resolution at 10 m and 100 m.

4. A 79 GHz radar has a $\pm 40^\circ$ horizontal beamwidth. Determine the horizontal aperture size.

5. A 79 GHz radar has a $\pm 5^\circ$ vertical beamwidth. Determine the vertical aperture size.

20
Inertial measurement

Typically, inertial measurement units (IMU) have three complementary sensors. A nine degree unit has:

- three axis accelerometer, $\mathbf{a} = (a_x, a_y, a_z)$
- three axis gyroscope, $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)$
- three axis magnetometer, $\boldsymbol{\theta} = (\theta_x, \theta_y, \theta_z)$

1 *Accelerometer*

The accelerometer measures the acceleration vector, i.e, the acceleration along each Cartesian axis,

$$\mathbf{a} = a_x\hat{\mathbf{x}} + a_y\hat{\mathbf{y}} + a_z\hat{\mathbf{z}}. \tag{20.1}$$

Typically, this is dominated by gravitational acceleration. If the IMU is moving with constant speed, then

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2} = g, \tag{20.2}$$

and thus the accelerometer can provide the pitch and roll, using

$$\phi_x = \tan^{-1}\frac{a_y}{a_z}, \tag{20.3}$$

$$\phi_y = \tan^{-1}\frac{a_x}{a_z}. \tag{20.4}$$

In theory, the acceleration measurements can be integrated to give velocity and integrated again to give position. However, in practice, slight offset errors when integrated produce drift. This is exacerbated for the position estimation since it will drift quadratically with time.

2 Gyroscope

The gyroscope measures the angular velocity, i.e., the angular speed around each axis,

$$\boldsymbol{\omega} = \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{\theta}. \tag{20.5}$$

Again, in theory, the angular velocity measurements can be integrated to give orientation by slight offset errors when integrated produce drift.

3 Magnetometer

The magnetometer measures the Earth's magnetic field to estimate orientation. However, the Earth's magnetic field vector varies around the planet and is affected by the presence of ferrous materials.

4 Barometer

An air pressure sensor can give a good estimate of altitude if calibrated for the ambient air pressure.

5 Sensor fusion

5.1 Complementary filter

The pitch can be estimated using

$$\hat{\theta}_x(n+1) = \alpha\left(\hat{\theta}_x(n) + \omega_x(n)\Delta t\right) + (1-\alpha)\theta_x(n), \tag{20.6}$$

where $\omega_x(n)$ is the angular speed around the x-axis measured by the gyroscope, $\theta_x(n)$ is pitch around the x-axis inferred from the accelerometer, Δt is the sampling interval, and α is a weighting parameter.

The fused angle is found from the angle estimate from the accelerometer, passed through a low-pass filter, added to the integrated gyroscope angular speed, passed through a complementary high-pass filter. In the Laplace domain, these operations can be represented using

$$\hat{\theta}(s) = L(s)\theta(s) + \frac{\omega(s)}{s}H(s), \tag{20.7}$$

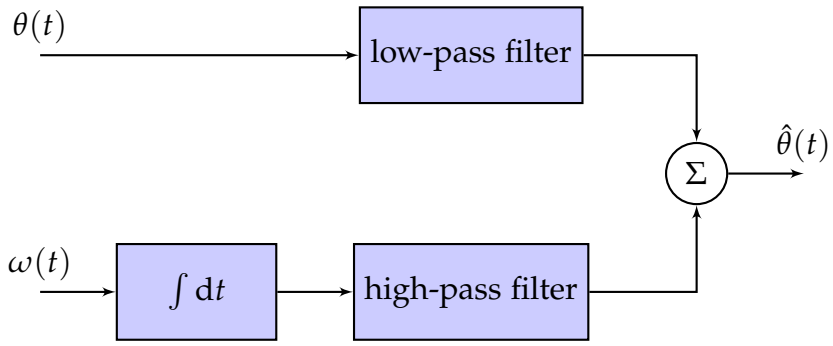


Figure 20.1: Complementary filter.

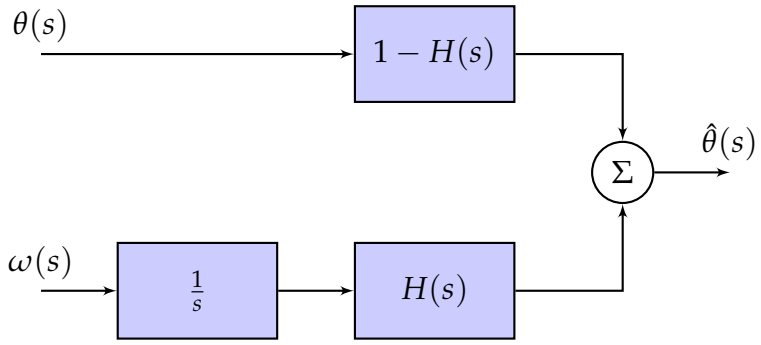


Figure 20.2: Complementary filter in the Laplace domain. $H(s)$ is the transfer function for a high-pass filter, $1 - H(s)$ is the transfer function of a complementary low-pass filter.

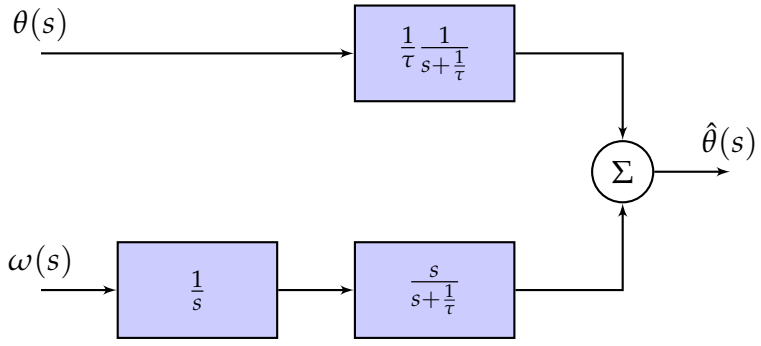


Figure 20.3: Complementary first-order filter in the Laplace domain.

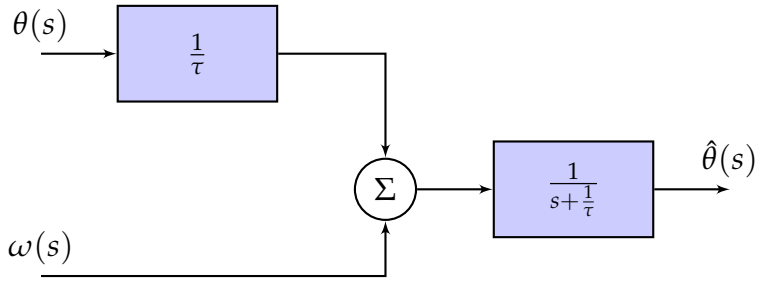


Figure 20.4: Simplified complementary first-order filter in the Laplace domain.

where $L(s)$ is the transfer function of the low-pass filter and $H(s)$ is the transfer function of the high-pass filter,

$$L(s) = 1 - H(s) = \frac{1}{\tau}\frac{1}{s+\frac{1}{\tau}}, \tag{20.8}$$

$$H(s) = 1 - L(s) = \frac{s}{s+\frac{1}{\tau}}. \tag{20.9}$$

Using these complementary filters, (20.7) becomes

$$\hat{\theta}(s) = \frac{1}{\tau}\frac{1}{s+\frac{1}{\tau}}\theta(s) + \frac{\omega(s)}{s}\frac{s}{s+\frac{1}{\tau}}. \tag{20.10}$$

This can be expressed as (see Figure 20.4)

$$\hat{\theta}(s)\left(s+\frac{1}{\tau}\right) = \frac{1}{\tau}\theta(s) + \omega(s), \tag{20.11}$$

and taking inverse Laplace transforms yields

$$\frac{\mathrm{d}}{\mathrm{d}t}\hat{\theta}(t) + \frac{1}{\tau}\hat{\theta}(t) = \frac{1}{\tau}\theta(t) + \omega(t). \tag{20.12}$$

Using discrete-time signals where $t = n\Delta t$, (20.12) becomes

$$\frac{\hat{\theta}(n) - \hat{\theta}(n-1)}{\Delta t} + \frac{1}{\tau}\hat{\theta}(n) = \frac{1}{\tau}\theta(n) + \omega(n) \tag{20.13}$$

Solving for $\hat{\theta}(n)$ leads to a recursive (IIR[1]) digital filter[2],

$$\hat{\theta}(n) = \alpha\left(\hat{\theta}(n-1) + \omega(n)\Delta t\right) + (1-\alpha)\theta(n), \tag{20.14}$$

where

$$\alpha = \frac{1}{1+\frac{\Delta t}{\tau}}, \tag{20.15}$$

or

$$\tau = \frac{\alpha}{1-\alpha}\Delta t. \tag{20.16}$$

[1] Infinite impulse response.
[2] The complementary filter is equivalent to a steady-state Kalman filter for a certain class of problems.

Table 20.1: Relationship between complementary filter parameter α and its time constant τ.

α	$\tau/\Delta t$
0.9	9
0.95	19
0.98	49
0.99	99
0.999	999

5.2 *Kalman filter*

There are many different Kalman filter formulations[3] to fuse the gyroscope and accelerometer measurements to estimate angle.

[3] Some cheat by pretending that the gyroscope measurements are control inputs.

The measurement vector at time step t is

$$\mathbf{z}_t = \begin{bmatrix} \tilde{\theta}_t \\ \tilde{\omega}_t \end{bmatrix}, \tag{20.17}$$

where a tilde denotes a noisy measurement.

One possible system state is the angle, θ, angular rate, ω, and gyroscope angular rate bias, β,

$$\mathbf{x}_t = \begin{bmatrix} \theta_t \\ \omega_t \\ \beta_t \end{bmatrix}. \tag{20.18}$$

The measurements are related to this state by

$$\tilde{\theta}_t = \theta_t + v_{\theta t}, \tag{20.19}$$
$$\tilde{\omega}_t = \omega_t + \alpha_t + v_{\omega t}, \tag{20.20}$$

where $v_{\theta t}$ models the angular measurement noise and $v_{\omega t}$ models the angular rate measurement noise. In matrix form,

$$\mathbf{z}_t = \mathbf{C}\mathbf{x}_t + \mathbf{v}_t, \tag{20.21}$$

where

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}. \tag{20.22}$$

If the angular rate is slowly changing, then the state update equations are,

$$\theta_t = \theta_{t-1} + \omega_{t-1}\Delta t + w_{\theta t}, \tag{20.23}$$
$$\omega_t = \omega_{t-1} + w_{\omega t}, \tag{20.24}$$
$$\alpha_t = \alpha_{t-1} + w_{\alpha t}. \tag{20.25}$$

These equations can be expressed in matrix form as

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{w}_t, \tag{20.26}$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta t & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{20.27}$$

Note, with this formulation, there is no control input.

6 AUV state space

The AUV state space can be represented by the AUV pose and its time derivative (the linear and angular velocities).