Department of Computer Science, University College London

COMP0222/COMP0249 Lab 01: Introduction to the Software

13th January, 2026

1 Scope and Purpose

This lab introduces you to the basic libraries and software used throughout much of this module. You will also gain some experience with Kalman filters. The exercises here are designed to make you engage with the software to understand some aspects of how it works and to make clear that it isn't simply an opaque black box which you have absolutely no control over.

As explained in the overview lecture, all of the coding for initial algorithm development will be carried out in MATLAB. The main reasons for this relates to ease of use: unlike other alternatives, MATLAB offers a highly stable and consistent platform and does not require the use of docker images, virtual environments or other similar techniques. In addition, it provides a sufficient level of performance. It is also integerated with a large number of robotics-related packages, some of which will be used to support later exercises.

2 Scenario

The goal is to develop a localization system for a point-like robot called pointbot. pointbot is a particle which operates on a 2D plane. Its state only consists of its position and velocity in 2D,

$$\mathbf{x}_k = \begin{bmatrix} x_k & \dot{x}_k & y_k & \dot{y}_k \end{bmatrix}^\top. \tag{1}$$

Pointbot is observed by two types of sensors: a "GPS" sensor (which directly measures (x_k, y_k)) and a "bearing" sensor which measures the angle of the platform from a fixed position or sensor. Each type of sensor has a finite detection region, in which the platform can be detected.

3 Activities

Activity 0: Install the Software

First, you will need to install MATLAB from the Mathworks website, `http://www.mathworks.com`**. You can use your UCL login to do this. We recommend using the latest version (2025b at time of writing). This is particularly important if you have an Apple Silicon machine because MATLAB has recently been extensively overhauled for the platform. The code is unlikely to run on anything earlier than 2022b.**

Second, you will need to install the software for the module. As explained in the introductory lecture, this is hosted at `http://www.github.com/UCL/comp0222-comp0249_25-26`**. This repository is completely self-contained. All third party libraries are shipped directly with the code and submodules are not used.**

Suppose you clone your code to the directory `comp0222-comp0249_25-26`**. You should change to this top-level directory in MATLAB and type:**

```
>> setup
ebe added to the search path
>>
```

To test if everything is installed, type:

```
>> l1.activity0
```

A window should open and you should see something like the image in Figure 1. The filled blue circle is the ground truth location of the particle position. The red plus is the measurement from the noisy position sensor, and the red circle is the depiction of the 2 standard deviation covariance ellipse of this measurement.

The `l1.` **prefix arises because we use namespaces. We use these extensively to ensure that different activities across the module do not interfere with one another.**

Activity 1: Investigating the Simulator Output

Run:

```
>> l1.activity1
```

Pointbot now moves more agressively and the scale of the axes is incorrect — in many runs you should see pointbot disappear off of the screen. Work out what values the axes should be set to so that the platform is visible all the time.

Although you could do it trial and error, you should attempt to do this systematically. There is a method `history()` **which you can call from the simulator which will return the history of the simulated platform trajectory. Use this to investigate the quantitative values of the behaviour of the target, and modify the axes of the figures appropriately. What are the largest and smallest magnitudes of velocities that platform achieves?**

With reference to the process model in Subsection B.1, you could also investigate the effects of different values of `alpha`**,** `beta` **and** `sigmaQ`**. These values are stored in** `config/activity1.json`**.**

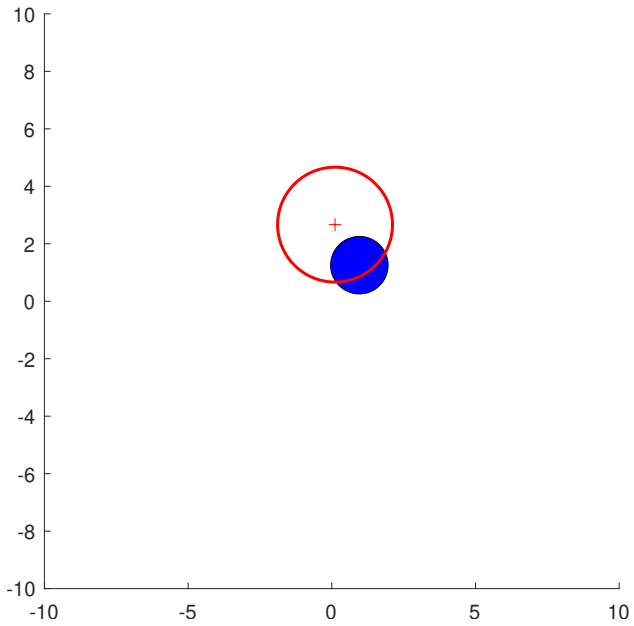

Figure 1: Simulator output from activity 0. Since the noise is randomly generated, you will probably see a slightly different view than this one.

Activity 2: Inspecting the Simulator Loop

In this activity, you are going to look a bit more about how the simulator loop works. Here, only the simulator is running and generating events. The environment is also a bit more interesting, because it now contains an occluder.

Run the script:

```
>> l1.activity2
```

You should now see a scenario similar to Figure 1. The scenario now includes a GPS "occluder" — this is the grey shaded box. If the platform (blue circle) strays inside the occluder box, no valid GPS measurement is available. The most recent GPS measurement is also shown. If the platform is within an occluder, the GPS shows the last measurement received.

In the code you will see the main loop:

```
while (simulator.keepRunning()  == true)
    simulator.step();
    events = simulator.events();
    simulatorViewer.visualize(events);
    drawnow
end
```

Investigate the value of `events` over time. What is the first event sent, and can you guess why? What subsequent events are generated? What do they represent? What data is stored in them? How regularly are they created?

What happens if you comment out the line `simulatorViewer.visualize(events);`? What happens if you comment out the line `drawnow`?

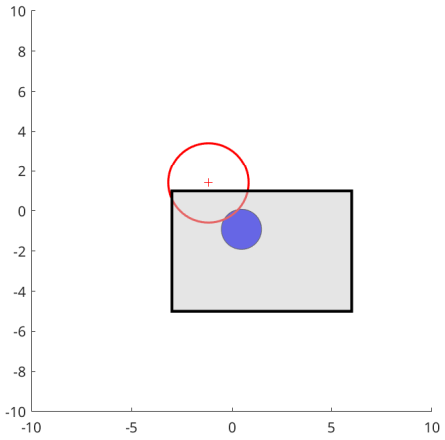

Figure 2: Simulator output from activity 2. An occluder (grey box) causes GPS dropout whenever the platform is inside it.

Activity 3: Prediction Performance of the Kalman Filter

The script `activity3.m` uses the Kalman filter object which is the estimation algorithm. (If you look in the `while` loop you can see how it's used).

The Kalman filter, as it stands, only runs the predictor and visualizes the output as a mean and covariance ellipse. The GPS sensor is disabled.

What do you notice about the behaviour of the predictor? How do you think this can be explained by the nature of the process model? Which kind of event causes the Kalman filter to predict ahead?

The Kalman Filter has a method `estimateHistory()` to get the time history of the state and covariance values. Plot these. What do you think causes the behaviour you see?

What happens if you set `alpha` and `beta` to 0 in `config/activity3.json`? Does it behave the way you expect it to?

Activity 4: Complete Implementation of the Kalman Filter for the GPS Measurement

In this activity, you will need to finish the implementation of the Kalman filter. In particular, you will need to implement the update step using the observation model in Subsection B.2 and the Kalman filter update equations in Subsection B.4.

The code should be implemented in the method `handleGPSUpdate` in the `KalmanFilter` class.

Test on the script:

```
>> l1.activity4
```

A useful way to debug algorithms is to allow for the case where there is no noise and the true state and the prediction should be identical. Noise can be disabled by changing the value associated with `perturbWithNoise` to `false`.

Note that this code includes the command `pause(0.2)` in the loop. This is to slow things down to make it easier to see what's going on.

Activity 5: MainLoop Class

This activity doesn't require you to code anything new, just check the form of the script `l1.activity5` **which you run from:**

```
>> l1.activity5
```

The previous activities have exposed the event loop directly. Although you can see exactly what is going on, it introduces a lot of redundant code which has to be executed in the right order.

This script shows how to use a helper class called `MainLoop` **is used.** `MainLoop` **now wraps up all the calls which were manually done in activity 1 and hides the details of the events which were generated.**

This script also shows a slightly more challenging environment where are there are more occluders which can hide the GPS. You should find your solution from Activity 4 will run in this activity without any changes.

Activity 6: Complete Implementation of the Kalman Filter for the Bearing Measurement

So far, we have only looked at the linear GPS sensors. In this final task, the goal is to extend the `KalmanFilter` **class to complete the implementation of this type of sensor.**

You will need to finish implementing the methods `handleBearingObservation`**. The bearing observation event contains the bearing measurements from all sensors that the platform is in range of. The** `info` **field specifies the ID of the sensor which detected the platform, which you will need to use to compute your solution.**

You should validate your implementation by computing the estimation error against the ground truth over time. Check the method `estimateHistory()` **of** `KalmanFilter`**.**

You can run:

```
>> l1.activity6
```

You might find that in some situations this filter will fail. This is a "feature" of the EKF which we will see will will have issues later.

A The Event Based Estimator Library

A.1 Rationale

Almost all modern autonomous systems operate in discrete time. Many systems, such as those using ROS[1]. Almost all estimation algorithms, including all the ones covered in this module and many more, are posed in terms of discrete time as well. They have a predict-update structures shown in Figure 3.

The Event-Based Estimator Library (`ebe`**) library was created to support these types of systems and algorithms in a very simple, self-contained manner. The library was designed to support the teaching of the algorithms and techniques in this module.**

The main components are:

1. **Events. All operations are carried out with these. Events have a type, a timestamp, and a payload which consists of data needed to describe the event.**

[1]The Robot Operating System, `https://www.ros.org/` is widely used in research and industry. Rather than discrete events, ROS uses a set of discrete messages.

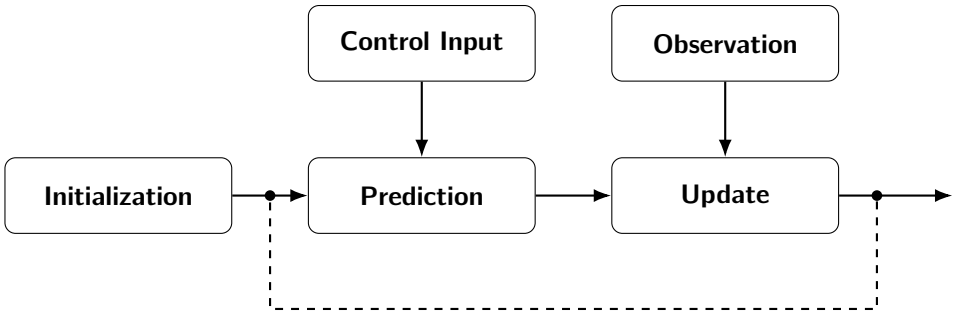

Figure 3: General sequence of how all recursive estimation algorithms operate.

2. **An event generator. This creates a sequence of events over time. These events are either created by a simulator or from a log file or other archive of data.**

3. **An estimator. This processes the events and uses them to construct an estimate of the environment.**

The basic sequence of operation is shown in the figures below. At each timestep k, which corresponds to a time t_k, an event is generated. The estimator initially starts in an uninitialised state but receives events. As the events are provided, it will eventually initialize. When it does that, it then switches into the predict-update sequence.

B System Description

This appendix describes the process model and the observation models.

B.1 Process Model

The state space is

$$\mathbf{x}_k = \begin{bmatrix} x_k & \dot{x}_k & y_k & \dot{y}_k \end{bmatrix}^\top. \tag{2}$$

The process model is a damped order system in which process noises are injected into the acceleration. The continuous time process model is

$$\dot{\mathbf{x}}_k = \begin{bmatrix} \dot{x}_k \\ \ddot{x}_k \\ \dot{y}_k \\ \ddot{y}_k \end{bmatrix} = \mathbf{F}_k^c \mathbf{x}_k + \mathbf{G}_k^c \mathbf{v}_k^c, \tag{3}$$

where

$$\mathbf{F}_k^c = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\alpha & -\beta & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\alpha & -\beta \end{bmatrix}, \quad \mathbf{G}_k^c = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

and $\mathbf{v}_k^c$ is zero mean, Gaussian distributed with covariance $\mathbf{Q}_k^c$. Note that the Kalman filter we are using here will be implemented in discrete time. Therefore, we use van Loan's method[2] to compute the discrete-time

[2]For details, see the section "Discretization of the Process Noise" of the Wikipedia page on Discretization.

equations. An implementation of this approach is provided in the file `ebe.utils.continuousToDiscrete`. The result is the discrete time system of the form

$$\mathbf{x}_{k+1} = \mathbf{F}_k^d \mathbf{x}_k + \mathbf{v}_k^d, \tag{5}$$

where $\mathbf{F}_k^d$ is the discrete time process model, and $\mathbf{v}_k^d$ is the process noise. For this linear system, it can be shown that the process noise is zero-mean, Gaussian distributed and has the covariance $\mathbf{Q}_k^d$.

B.2 GPS Observation Model

This directly measures just position

$$\mathbf{z}_k^G = \mathbf{H}_k^G \mathbf{x}_k + \mathbf{w}_k^G, \tag{6}$$

where

$$\mathbf{H}_k^G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{7}$$

and $\mathbf{w}_k^G$ is a zero-mean Gaussian distributed random variable with covariance $\mathbf{R}_k^G$.

Note that measurements are not received when the robot is under cover.

B.3 Bearing Sensor

At strategic points in the environment have been placed bearing sensors which can measure the angle of the platform relative to the sensor location. Suppose a sensor s^i is at a position (x^i, y^i) with a relative orientation θ^i. The observed angle is given by

$$\theta_k^i = \tan^{-1}\left(\frac{y_k - y^i}{x_k - x^i}\right) - \theta^i \tag{8}$$

Note that each sensor has a finite detection region and angle. The platform is detected for s^i if

$$\left(x_k - x^i\right)^2 + \left(y_k - y^i\right)^2 \leq \left(d^i\right)^2 \tag{9}$$

and if the angle lies in the detection cone

$$|\theta_k^i| \leq \theta^i. \tag{10}$$

The Jacobian of the observation model has to be taken with respect to the platform state, not the sensor state. Therefore,

$$\nabla_x \mathbf{h} = \frac{1}{\left(x_k - x^i\right)^2 + \left(y_k - y^i\right)^2} \begin{bmatrix} -\left(y_k - y^i\right) & 0 & \left(x_k - x^i\right) & 0 \end{bmatrix}. \tag{11}$$

B.4 Kalman Filter Equations

The state estimate is $\hat{\mathbf{x}}_{k|k}$ and the covariance $\mathbf{P}_{k|k}$.

The prediction equations are

$$\begin{aligned} \hat{\mathbf{x}}_{k+1|k} &= \mathbf{F}_k^d \hat{\mathbf{x}}_{k|k} \\ \mathbf{P}_{k+1|k} &= \mathbf{F}_k^d \mathbf{P}_{k|k} \left(\mathbf{F}_k^d\right)^\top + \mathbf{Q}_k \end{aligned}$$

When the observation is linear, the update equations are

$$\begin{aligned}\hat{\mathbf{x}}_{k+1|k+1} &= \hat{\mathbf{x}}_{k+1|k} + \mathbf{W}_{k+1|k}\boldsymbol{\nu}_{k+1|k} \\ \mathbf{P}_{k+1|k+1} &= \mathbf{P}_{k+1|k} - \mathbf{W}_{k+1|k}\mathbf{S}_{k+1|k}\mathbf{W}_{k+1|k}^{\top},\end{aligned}$$

where

$$\begin{aligned}\boldsymbol{\nu}_{k+1|k} &= \mathbf{z}_k - \hat{\mathbf{z}}_{k+1|k} \\ \hat{\mathbf{z}}_{k+1|k} &= \mathbf{H}_{k+1}\hat{\mathbf{x}}_{k+1|k} \\ \mathbf{W}_{k+1|k+1} &= \mathbf{C}_{k+1|k}\mathbf{S}_{k+1|k}^{-1} \\ \mathbf{C}_{k+1|k} &= \mathbf{P}_{k+1|k}\mathbf{H}_{k+1}^{\top} \\ \mathbf{S}_{k+1|k} &= \mathbf{H}_{k+1}\mathbf{C}_{k+1|k} + \mathbf{R}_{k+1}\end{aligned}$$

When the system is nonlinear, the EKF can sometimes be used. In this case,

$$\hat{\mathbf{z}}_{k+1|k} = \mathbf{h}\left[\hat{\mathbf{x}}_{k+1|k}, \mathbf{0}\right].$$

The KF equations are the same, except the matrix $\mathbf{H}_{k+1}$ is replaced by the Jacobian $\nabla_x \mathbf{h}$