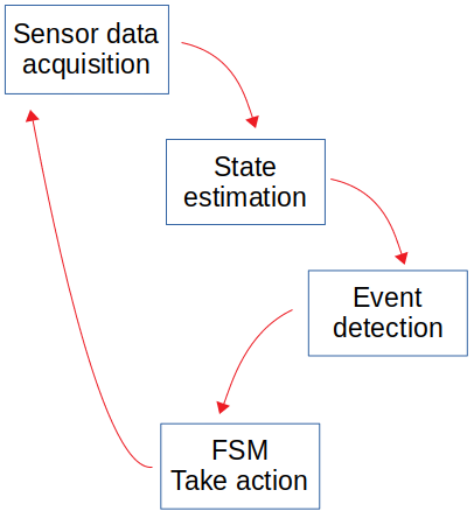
Controlling a Robot's Actions
Finite-State Machines and Robot Behaviour

The last component of our course ties together everything we have learned up to this point, and provides a framework for getting our robot to perform complex tasks using a fairly simple, yet flexible and powerful form of Artificial Intelligence – Finite State Machines.

FSMs are often used in gaming, to provide interesting and complex behaviour for non-playable characters. In the context of robotics, FSMs allow for the implementation of a rich set of behaviours that are:

- Dependent on the state of the robot
- Reactive to changes in state either of the robot or the environment
- Easy to implement and extend as needed
- Easy to simulate, which is an advantage when designing and testing a new system
- Predictable, which for many robotics applications is a ***requirement***

In order to understand how an FSM can be used to control a robotic system, we first need to have a look at the different components of a robot's control/decision/action loop. Such a look is shown in the figure below.



Sensor data acquisition

As we have discussed at length, this step requires the robot to use any available sensors to obtain estimates of the relevant data required to make decisions and complete its task. In this step we have to be mindful of

- Noise
- Sampling rate issues

The sensing step should take care to apply denoising if appropriate given the type of data being measured, and the sensors available. Denoising must be time-efficient, since the robot is now acting within a loop of sequential operations, any delays in the sensor measurement stage will impact the succeeding steps of processing.

Keep in mind the sampling rate. As noted during our discussion of *control systems*, if the sampling rate of the system is too low, it will not be possible to implement a controller that successfully brings the system into agreement with the reference.

The issue here is that because of the sequential nature of the loop, sensor measurements can only take place once the rest of loop has finished its work. It is ***essential*** that all steps in the process are implemented as efficiently as possible, minding ***code optimization*** and ***profiling***, and ensuring there are no ***unnecessary computation steps*** that could delay the completion of the loop.

The maximum sampling rate achievable is a function of the time it takes to complete the loop. Effort must be made to identify and remove any bottlenecks.

State Estimation

This step takes care of the fundamental problem of determining reliable values for the ***state variables***, and can make use of any appropriate methods for improving on the data provided by sensors, or for estimating values from noisy, incomplete, or indirectly measured data. This includes

- Using estimation methods such as the Kalman Filter
- Using a sampling based estimator, such as a particle filter
- Using physics, as in the case of inertial navigation (possibly together with Kalman Filters)
- Using Machine Learning to implement a regression method that estimates state variables

These are only a handful of examples of what the state estimation step may need to do. What is actually required will depend on the particular application, the sensors available, whether or not the sensors' measurements directly relate to state variables, and speed/power requirements (e.g. we can't run a powerful machine learning algorithm on a low-powered, portable robotic platform).

Regardless of what is implemented here, the same warning about ***computation speed*** that applies to the sensing step also applies here. Any delays caused by the state estimation process will impact the performance of the robot control loop.

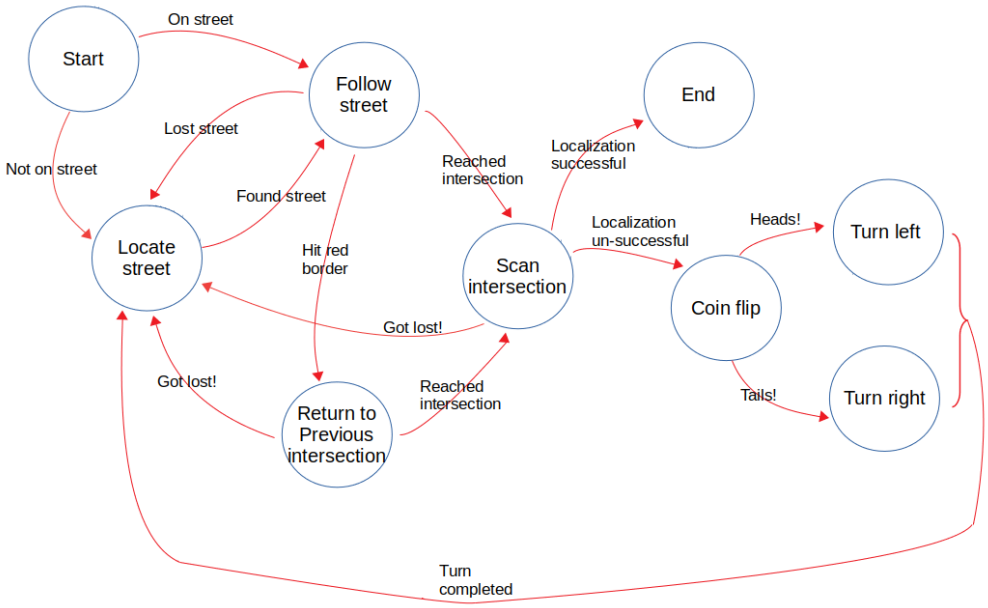
Event Detection and FSM-based decision making

In order to understand how the last two steps in the loop work, first we should review what the components of an FSM are, and how they can be used to implement robot behaviours.

An FSM consists of a set of ***states,*** and a set of ***state-transitions***. The ***states*** correspond to actions a robot may need to perform to achieve its goal. The ***state-transitions*** are triggered by ***events***, and they

correspond to ***decisions*** the robot takes regarding what action much be performed at any given point in time.

Let us consider the case of the localization problem you solved using ***histogram localization*** on a colour-coded map of streets and intersections. An FSM to control a robot performing this task may look like the model shown below (note this is only ***one possible implementation***, multiple ways to do this are possible).



A few things are worth noting from the diagram. First, there is always a ***start state*** which takes care of any initialization required for the robot (in this case, setting up initial probabilities for each intersection, and figuring out if the robot is or is not on a street at the beginning of the process). There is also an ***end state*** which is reached once the robot has achieved its (or one of its) goal(s).

Secondly, the diagram illustrates several types of ***events*** (the conditions indicated with each of the state transition arrows). There are events that are directly derived from sensor input (e.g. ***reached intersection*** which can easily be decided from looking at the input from the colour sensor); there are events that require ***processing*** and are the result of applying a particular ***algorithm*** to the current and possibly also historical state variables (e.g. ***localization successful*** which requires us to carry out histogram localization and determine if the robot knows where it is); there are also ***stochastic events*** that depend on some random outcome (e.g. ***coin flip*** which effectively takes a random number between 0 and 1 and chooses one of two outcomes based on this).

Many events will be combinations of the three types described above. The ***event detection*** step in the main robot control loop is in charge of evaluating the truth value for every possible event in our FSM. Once these events have been processed, the ***FSM take action step*** determines what action must be performed. This is a very simple process:

- Look at the ***current state*** for the robot
- Based on the ***events*** that have been determined to be true choose an appropriate transition to the ***next state*** (if no transition is indicated, ***next state=current state***)
- Send the appropriate commands to the robot that correspond to what it needs to do when in ***next state***.
- Update the state so ***current-state = next state***
- Return to the top of the robot control loop (Sensor data acquisition)

As you can see, the process is straightforward. There are however a few fine points that need to be taken into consideration:

- States (which amount to actions) can themselves correspond to complicated tasks. For instance, ***Return to previous intersection*** involves a sequence of actions the robot must perform. To implement such a state, we can create a ***smaller FSM*** that correspond to the sequence of actions the robot performs to complete '***return to previous intersection'***. We can thus have multiple levels of detail represented by FSMs that get triggered as needed. This is called ***hierarchical FSM*** and allows us to model our system making use of abstraction at the highest level, while still being able to model and simulate each simple action the robot must eventually perform.

- Fall-backs are critical – notice the ***'got lost'*** events in the diagram above. They are there to prevent a situation in which the robot is expected to complete a particular task (e.g. scan an intersection) but for reasons due to the normal uncertainties involved in robot operations, the robot is now in a situation where it is ***no longer possible for it to complete the task*** specified by its current state. Without a ***fallback***, that is, a condition that allows the robot to ***exit the current state without having completed the task***, the robot would be stuck in a loop. If you have problems while implementing an FSM and your robot gets stuck in a specific state, the first thing to consider is whether you have missed any conditions that would make the robot unable to complete its task, and that should be covered by an appropriate fallback.

- The level of detail in the FSM is not specified anywhere. As a designer you are free to build your FSM using any level of abstraction that is reasonable for your task. However, for the FSM to be actually helpful, your design should be ***detailed enough*** that it can easily be mapped into ***an implementation*** of the software that is needed to provide the functionality specified by a state. As an example, an FSM with a single state called 'Perform Histogram Localization Process' is correct, but does not help anyone actually implement the required software.

- For the above reasons, it is usually a good idea to have your FSM reviewed by a colleague that is not directly involved with its development. The process can help spot imprecisions, missing transitions, missing fallbacks, or places where the state description is too general to be of use.

Implementing FSMs

The implementation of an FSM is straightforward. The simplest implementation (though not elegant by any means) consists of a set of ***if-then-else*** statements that check for ***each possible state*** and ***each event detected to be true*** what the corresponding transition should be. The state is then updated as needed, and a corresponding ***function implementing the actions required by the state*** is called.

A more elegant implementation uses ***a state transition table***. In this table there is ***one row per state***, and ***one column per event***. Given the transition table ***T,*** the value of ***T(i,j)*** gives the state that results if event ***j*** is ***TRUE*** and the ***current state*** is ***i***. For example, suppose that ***i=5*** corresponds to ***'scan intersection***' in the FSM above, and that the ***'end state'*** has index ***6.*** Suppose that the event ***'localization successful'*** is represented by column ***2*** in the transition matrix ***T***. Then we would have

T(5,2)=6

From state #5 (***scan intersection***), if we find that event #2 (***localization successful***) is true, the next state is state #6 (***end state***). The state transition table provides a very compact and elegant way to represent and handle the FSM without very long and involved sequences of ***if statements***. It is also much better in terms of performance – as it avoids having to evaluate long sequences of conditional statements (with the associated performance penalties due to branch mispredictions).

However, notice that if we are using a ***state transition table***, each distinct event must be represented by a column. Composite events which consist of multiple simpler events joined together by logical operators (e.g. in the FSM above we may have an event ***'turn completed && on street'*** which directly transitions to ***'follow street'***) would need to be stored as distinct events, with their own column in the transition matrix.

Important note: However the FSM is implemented, the ***functions that control the robot's actions can not contain wait loops***. For example, if we have a robot facing a particular direction, and want the robot to align itself with a different, reference direction, we may be tempted to write something like this:

```
while (current_dir != reference_dir)
{
   rotate_left();
}
```

This simple, harmless loop will fail to do what we expect. The reason is very simple – ***while the program is stuck in this loop, no sensor measurements are being taken, so 'current_dir' is not being updated and the condition will never fail***.

Recall that the sensor measurement step takes place ***after*** the FSM has decided on, and taken an action. But the action itself can not contain a wait loop that relies on a condition that depends on sensor measurements because no sensing happens until the FSM action step is complete.

So instead, the correct implementation will look more like

```
if (|current_dir-reference_dir|>threshold)
   rotate_left();

if (|current_dir-reference_dir|<=threshold)
   stop_rotation();
```

Notice that now the FSM decides that the robot needs to turn, it ***sends the command to start turning and continues*** without any wait loop.

This allows the sensor update to take new measurements, the robot will continue turning as long as the reference direction is different enough from the current direction, but once once the current direction is close enough to the reference direction, the FSM instructs the robot to stop turning.

Event-driven programming

Implied in the design of the robot control loop is the premise that the system is ***event driven***. What the robot will do depends on its current state, and whatever ***events*** are taking place at any given time. This is the same kind of programming framework that is the foundation for graphical used interfaces and other kinds of interactive systems.

The key property of these systems is that they ***must react quickly*** to events. That is not a precise statement, and if we want to make it precise, we have to delve into the field of ***real-time systems*** and ***real-time operating systems***. That is beyond the scope of this unit, however, we can certainly introduce the fundamental ideas that form the basis for the design and implementation of ***real-time systems***.

Real-time constraints

In real-time systems, the central idea is that ***events*** and ***tasks*** have ***deadlines***. That means that in terms of actual real-world clock time, the system has only a ***limited amount of time*** during which appropriate action must be taken to ***handle an event*** or ***complete a task***. If the system misses the deadline, we consider it a ***failure***.

There are several flavours of real-time operations:

- ***Hard real-time***, in which a single failure to meet a deadline causes the entire system to fail. E.g. consider a car's ABS braking system, either it kicks in on time when needed, or the result can be catastrophic.

- ***Firm real-time***, occasional failures to meet a deadline are tolerable but if too many occur or they happen too often the system fails. E.g. consider an automated red-light camera, if it misses a driver running a light every now and then, this is bad but not catastrophic. However too many misses makes the system useless.

- ***Soft real-time***, missed deadlines cause a degradation of service. For example, on-line video conferencing. Occasional video/audio hiccups are tolerable, but too many make the system

difficult and annoying to use.

Real time systems usually employ one or several of the following techniques to achieve their tasks while meeting the specified deadlines:

- ***Partitioning:*** Different critical tasks are assigned a fixed pool of system resources (e.g. memory, CPU time, access to devices). The partitions are fixed and ensure each critical task has enough resources to complete within the specified deadline. The downside of such a system is that it often requires more resources than an equivalent non-partitioned system, and some of the resources available may be heavily under-utilized.

- ***Priority-based multi-tasking***: Makes use of a ***job-scheduling policy*** that gives ***higher priority*** to ***real-time tasks*** whose deadlines are near. Any task in the system can make use of the resources available, but any running task can be ***pre-empted*** if a ***higher priority task with a near deadline*** needs to be completed. The ***scheduler*** is the central component of such a system, and many algorithms exist to implement it, providing various guarantees about the ability of the system to respond to events within a ***deterministic*** time frame.

The constraints of real-time operation imply that our regular desktop operating systems are not well suited for use in real-time applications. Instead, specialized ***real-time Operating Systems (RTOS)*** are preferred in robotics applications. Once again, the design and implementation of an ***RTOS*** is beyond the scope of our unit, but we can note a couple of critical differences between regular desktop/server O/S and an RTOS.

Deterministic latency

Because robotics systems need to respond to events that happen ***asynchronously***, and because such events are usually signaled and handled via ***interrupts and system calls***, a real-time O/S needs to have ***guaranteed latency*** for:

- Interrupt handling
- Job context switching
- Job scheduling
- System calls

The important idea here is that if we have a guarantee for how long it will take the O/S to perform the basic function of *swapping a current running job out, handling an interrupt or system call, and swapping the next job to be run in,* we can design our application with deadlines that can be met given the system's guaranteed latencies. Desktop O/S do not have guaranteed latency bounds for these critical components. Notice the additional requirement that ***job scheduling*** has to take place within a guaranteed amount of time.

Microkernels

Most desktop O/S consist of what is commonly called a ***monolithic kernel***. The ***kernel*** is he core of the O/S, and in most desktop systems it includes all kinds of functionality related to peripherals and devices that are attached to the system – for instance, there will be components running that are part of the kernel and whose job it is to listen for incoming network connections, there are device drivers running for any hardware attached to the system, there may be processes related to printing, or disk management, etc.

Because these services are all part of the ***kernel***, they have the highest system priority and ***can not be pre-empted by a user application*** regardless of its priority. You may see where this could create problems: A high-priority real-time task may have to wait for a kernel process that is updating a print queue, for instance.

To resolve this, many ***RTOS*** are built on a ***micro-kernel*** which contains only the ***most fundamental parts of the O/S***, namely: ***memory and CPU management, scheduling,*** and ***inter-process communication***. All other system services are ***run in user-space, as applications that can be pre-empted*** if a high priority, real-time task requires attention.

Examples of RTOS commonly used in robotics include ***VxWorks*** by ***Wind River*** systems, used in aviation and in NASA's planetary exploration rovers and probes, and ***QNX (Neutrino)*** by ***Blackberry,*** which is extensively used in transportation systems, and ***LynxOS*** often used in the aviation industry.

This is just a very general overview of what is a complex and wide area of research and application, if you are interested in this, you should pick up a text or two on real-time Operating Systems and learn more.