Particle Filter and NXTCAM driver for LEGO NXT

This page captures my recent experiences in testing the LEGO Mindstorms NXT and some third party sensors to determine its suitability for teaching both university and high school students. The challenge I set myself was to use the NXT to perform probabilistic localisation, which is about the most complex robot algorithm that I teach to undergraduates. Along the way I tried out a few of the Mindsensors products to see what they could contribute. In summary, I have to say that I found the NXT to be an excellent tool with plenty of potential for teaching.

On this page:

Localisation with a Particle Filter
NXTCam Camera from Mindsensors
The Mindsensors Infrared Range Sensor
The Mindsensors Compass

Localisation with a Particle Filter

Localisation is the process of working out where you are, so localisation for a robot is having the robot work out where it is using the readings from its own sensors. Sounds easy, but it can be surprisingly challenging. The big problem stems from the fact that the robot’s wheels have to slip to make the robot move (a fact of physics), so careful measurement of the motors’ rotation sensors is not going to help you. Fitting streams of range readings from ultrasound or infrared sensors has problems with noise and error, too. So, over the last decade or so, roboticists have pretty well agreed that the best way to localise is to combine the all of robot’s sensor readings over time using probability theory. Sounds challenging, but it can be surprisingly easy.

I’ve been using a probabilistic filter called a particle filter to localise a robot built around the NXT. The other name for this style of probabilistic localisation is Monte-Carlo Localisation, so called because it relies on finding the best odds from a series of seemingly random outcomes. All of the code is written in Robot-C and runs on-board the NXT in real time. This surprised me. Particle filters are supposed to be horribly computationally intensive with lots of floating point math, so surely you can’t expect it to run on a LEGO brick ... but run it did.

Particle Filters - the General Idea

A particle filter is made up of lots of particles (who would have thought?). Each particle represents a guess about the process that is being filtered; in our case, it is a guess about the location and orientation of the robot. Now, because the process that we are filtering is noisy (or why would we bother filtering it?) our guesses about the state of the process will tend to diverge. So every time we move the robot, we update the particles based on the noisy odometry readings from each wheel. Because we know that the odometry readings aren’t right, we take some reasonably informed guesses about what could be the right readings, and update each particle based on those guesses. If we make enough guesses, surely one of them will be right - or at least much better than if we blindly believe the odometry readings. This guessing of new position of particles based on the noisy odometry readings is called the predict cycle.

So even if we start with all the particles in the same place, the particles will quickly spread out as the robot moves around while we make various stray guesses about what the right readings might be. We need a way to make sure to converge the particles back again using our range sensors. The range sensors can sense the surrounding environment, and we can compare the range readings from the environment with the readings that we should be getting from our knowledge of the map. We do this by calculating what the range readings should be for each particle - that is, for each guess about robot position. Any particle that has calculated range readings that match the current sensor range readings from the sensors is obviously one of our good guesses, and a particle with very different calculated range readings is rubbish. The process of sorting the good particles from the bad is called the update cycle, because each particle has its goodness score - its weight - updated by the quality of its match to the sensor data.

So now we have our particles scored on how good they are, and from this you can work out where your robot is, either by simply taking the particle with best score, or some kind of weighted average. Sounds pretty good, but there’s a catch. The particles will continue to drift further and further away from reality. While the update cycle provides a method of scoring the particles, it doesn’t go putting them back in the right position. So what happens is that the particles on the fringe of the pack give you no useful information at all, and soon you are left with only a handful of particles that could match the new sensor readings. A couple of bad odometry guesses later, you have no particles left that could match the sensor readings and the filter is broken. Well, technically, the filter is impoverished but that seems so politically correct. It’s broken.

To stop the filter from breaking, you need to get a set of particles that represent where your robot is likely to be. This process is called resampling and it is typically the most time consuming part of the whole deal. For that reason , it is a good idea to spend a little time testing how much information is left in your particles before running off and resampling the lot. The mechanics of resampling are a bit tricky, but suffice to say you end up with a set of useful particles so your filter won’t break.

So in essence a particle filter for localising a robot looks something like this:
task main()
{
    set all particles at known start point;
    while (true)
    {
        read the encoders;
        for each particle
        {
            predict the position in the particle with a guess based on the
            odometers;
        }
        read the range sensors;
        for each particle
        {
            calculate the expected range readings based on the guessed
            position in the particle;
            update the goodness measure of the particle based on difference
            between the calculated range and the real range;
        }
        measure the particles' health level;
        if the particles are getting sick
        {
            resample the particles to make them healthy;
        }
        else if the particles are dead
        {
            ABORT ... ABORT ... ABORT ... (or reset particles to an uniform
            spread across environment)
        }
        localise by taking a measure of the robot's position based on
        weighted average of the particles;
    }
}

The Test Platform

The test platform is a wheelchair robot built with the NXT mounted over the two NXT motors. Sonar sensors are arranged on three sides of the robot.
Localisation in Action

In order to test the ability of the particle filter to maintain localisation, I set up a test where the robot had to visit various absolute way-points in order. The waypoints were set up to proscribe a rectangular path that ran the robot parallel to the walls. The videos below show two cases: the first without the particle filter running, and then with the particle filter running.

Download [no_filter.wmv](Windows) [no_filter.mov] (Mac)
Download [with_filter.wmv](Windows) [with_filter.mov] (Mac)

The first video shows the robot gradually losing localisation as the error in the integration of the odometry increases. After about 45 seconds, it runs into a wall.

The second video shows the robot maintaining localisation and correcting the errors in odometry. The robot is still perfectly localised after 90 seconds, and can happily do this task until the batteries run flat.

Code in Robot-C

The particle filter is implemented in Robot-C and available for download here - [nxt_particles.c](Windows) [nxt_particles.mov] (Mac)

NXTCam Camera from Mindsensors

The [NXTCam](Windows) does pretty well what it says on the box, but with a few caveats. According to Mindsensors: “This camera has realtime image processing capability to detect and tracking colorful and feature rich objects. This camera connects directly to NXT and supports tracking of upto 8 objects. It connects to NXT on a sensor port, connects to computer using USB interface, tracks upto 8 objects (with 8 different user defined colors) at 30fps, provides realtime tracking statistics to NXT, maximum power consumption: 5V at 57 mA, PC not needed for autonomous operation on NXT, supported environments: NXT-G, RobotC.”

All true enough, it can do these things although sometimes not very well. Nevertheless, my kids were very impressed to watch the robot chase a ball across the carpet! Speed of response is certainly not an issue and all in all, good value at the price, but they do need to straighten out the bugs and documentation soon.

My main gripe with the NXTCam is that the color discrimination performance is poor. The camera is very sensitive to red, and barely sensitive to blue. I've tried it in a few different lighting conditions. It is better under fluorescent light, but still not good. I've been using a black ball against a pale background which works well. Also the blobs can report erroneous boundaries near the edges of the image. The documentation is poor (and wrong in places) and you need to be prepared to work things out for yourself.

The erroneous reporting of blob boundaries made the camera useless for localisation, as there is no obvious way to filter these bad measurements. Also the limitations with colour segmentation mean that I would have had to set up an artificial environment with contrasting colours and uniform lighting. And the lack of documentation would also mean that I would have to determine the camera parameters for myself so that I could transform image coordinates to world coordinates.

I've written my own library which sets up the camera and grabs the blobs into an array. You will need to use [nxtcamview](Windows) to set up the color spaces (although you could probably write a Robot C program to do it as well). I also have some demos that display the blobs as text, display the blobs graphically and chase a blob around. These files are available for download below. Limited documentation is included in the source files.

4/12/07: New version of code provided to work around bug in Robot-C compiler where arrays of structs sometimes go horribly wrong.

- [nxtcamlib.c](Windows) - Provides an easy to use Robot C interface to the NXTCam. Include this file in your application as shown in the demos below.
- [camtest.c](Windows) - Text based debugging tool for NXTCam using Robot C. Uses nxtcamlib.c.
- [cam_display.c](Windows) - Graphical debugging tool for NXTCam using Robot C. Uses nxtcamlib.c.
- [blob_chase.c](Windows) - Fun demo of the NXTCam. Uses nxtcamlib.c.
- [nxtcamlib.zip](Windows) - All of above files zipped together.

The Mindsensors

Infrared Range Sensor

The Mindsensors range sensor gives mm accurate readings to solid surfaces. This sensor (as I understand it) uses a triangulation principle by
measuring the offset of the reflected light from the infrared transmitter. This is great, as the method is somewhat immune to the changes in reflectivity of the surface being measured and to changes in ambient light conditions. The sensor itself is of good quality, and easy enough to hook up using LEGO pieces to your robot, as are all the Mindsensors devices I've tried. The compass uses the usual Mindsensors I²C interface which is easy enough to use with plenty of examples.

The infrared transmitter in the sensor isn't a laser, so the beam isn't collimated - meaning that its not the narrow spear of light we associate with lasers, but more the spreading beam you get from a torchlight. This means that the triangulation calculation is based on the centroid of that beam, rather than a crisp point. The upshot of this method of using uncollimated light is that you get very accurate measurements when the full spot of light is reflected back to the sensor. As the beam falls of the edge of an object, the centroid becomes distorted and the measurement at the edge makes the object look further away than it really is. So if you scan across a cube, for instance, rather than picking up crisp edges, the cube will look like a trapezoid.

This is not a problem when the application is obstacle avoidance for example. In fact, it is even a desirable side effect that will help the robot be a little be more cautious in uncertain areas. If all you want is a good quality sensor to stop your robot from running into things, then this is the one for you.

For localisation, though, this otherwise very nice sensor is completely useless. The measurements in the update process of a particle filter need a mean distance with known noise. The gradual worsening of measurements around the edges of objects means that you can't model the noise, so you don't know if these are errors in readings or errors in localisation. The standard sonar sensor, for all its lack of accuracy and occasional weirdness has more easily modellable noise, which is why I ended up using it as my sensor of choice for NXT localisation.

The Mindsensors Compass

The Mindsensors compass reads the magnetic field apparently quite well, and provides a useful resource in areas where the magnetic field is well known in advance. The factory settings are good, and the calibration is fairly straightforward as well if you should feel the need. The compass uses the usual Mindsensors I²C interface which is easy enough to use with plenty of examples.

BUT - when I tried to add the sensor into my particle filter, I found the magnetic field in our concrete office building to be heavily distorted - out by as much as 120 degrees in places - which made the compass useless for localisation. If I mapped magnetic field, I could probably add it in, but there's no straightforward way to use it for localisation.

So, there is nothing wrong with the compass per se, but there are well known limitations with using the compass for navigation indoors. I advise a check with a hand compass in the area you intend to work with first before deciding whether this is the sensor for you.