Example System - A Radar Autotracking System

Pete Wilson • Kiva Design • pete@kivadesigngroupe.com

Introduction

Warships need to see what's going on around them. To do this, they use a number of sensors, one of which is a surveillance radar. This is the sort of radar that one can see at airports - a curved antenna, rotating on a sturdy mast, with one rotation every few seconds. As it rotates, the radar emits pulses at a regular intervals. After each pulse is emitted, it remains silent for a while listening for radar echoes.

These days, the radar is likely to be attached to a computer system, which will convert the analog signals to digital, and process them so that echoes seen can be classified and tracked. If the echo looks threatening, its position and velocity can be brought to the attention of other systems (whether human or automated) and mayhap eventually handed over to some weapon system to take appropriate action. Very similar activities go on in an air traffic control system, but there the system will be looking out for planes diverging from the expected flightpath, and for possible collisions.

This system has a number of aspects which make it interesting for our multicore work. It has some gratuitous concurrency - on the face of it, there's no reason that each individual piece of sky cannot be processed concurrently, for example; and presumably the computations for each target can be processed independently. And it has a shared database (the structure holding the track information), which will need some concurrency control. It has some fairly heavy computational work to be done to process each track; it has a DSP-like workload to accomplish when processing the raw radar data; it has a real-time precision issue (in telling the hardware exactly when to look at the raw radar data); and more.

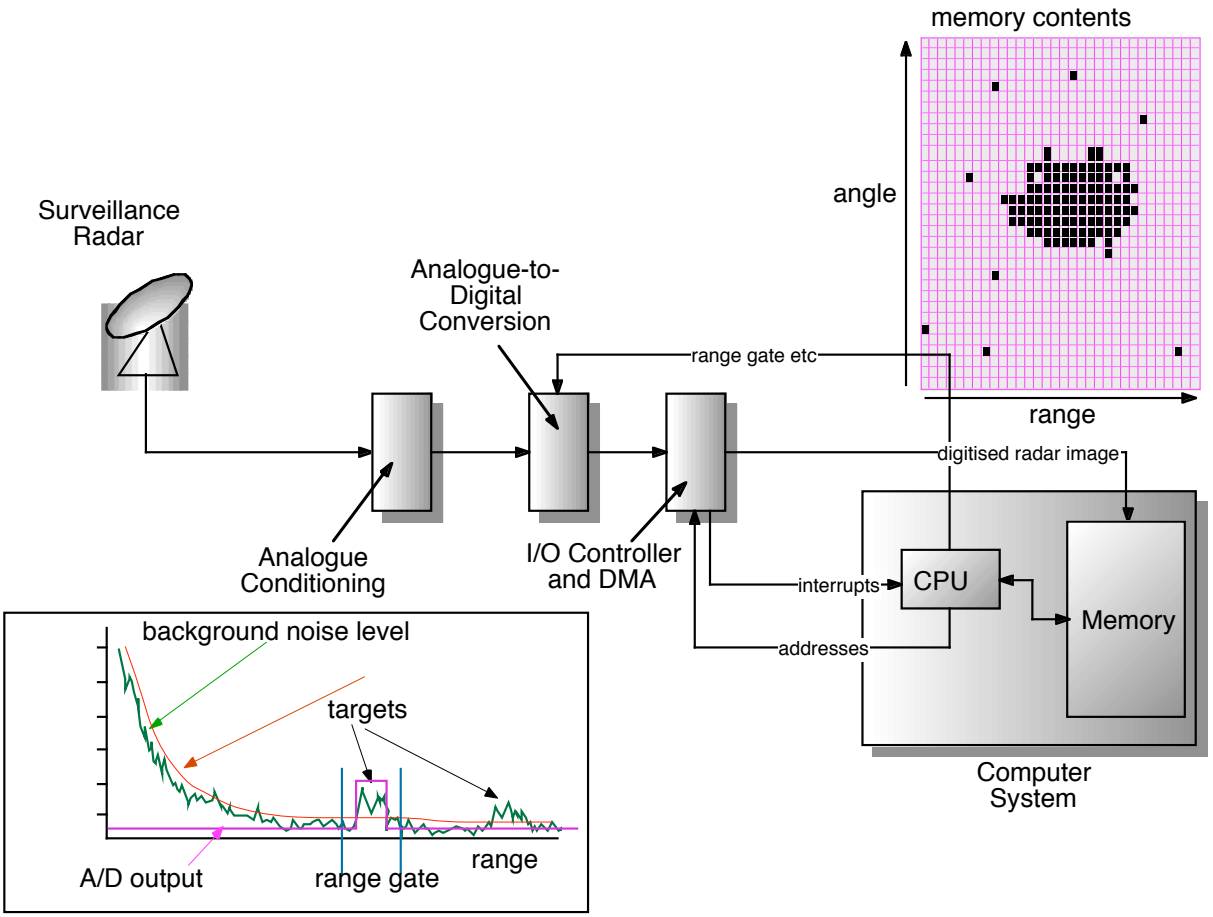
This essay concentrates on explaining the system; discussing its implementation on first a uniprocessor and later on a multicore platform are subjects for later essays, when we have covered RTOS. The description of the autotracking system provided here is a simplified version of that used in the Ferranti CAAIS (Computer-Aided Action-Information System) system implemented using Ferranti FM1600B computers for the Royal Navy, initially in the Type 12 frigate *HMS Torquay*[1] pictured above.

[1] http://en.wikipedia.org/wiki/HMS_Torquay

What the System Does

The diagram below shows a stylised representation of a radar autotracking system. The surveillance radar rotates at a constant rate (once every ten seconds in our example), and emits a radar pulse at a constant rate. Since we want to be able to see up to a bit less than 186 miles away, and the speed of light is 186,000 miles per second, the pulses occur once every two milliseconds. As the pulse travels away from the radar antenna, it encounters whatever is in its path and when it does so, a portion of the pulse is reflected back to the radar. Reflections fro things further away result in later reflections. The strength of the reflected signal from nearby targets is much stronger than that from distant targets, and so electronics shown as 'Analogue Conditioning' in the diagram changes the sensitivity of the receiver over time for each pulse, so that reflections from targets of equal radar size at all ranges provide essentially the same signal strength for the Analogue-to-Digital converter, which turns the reflected analogue pulse into digital form.

The A/D in this system is very simple - it samples the signal compares it with a fixed threshold value, providing a 1 when the signal is above the threshold and 0 when below. The A/D operates for a short period of time so that its output reflects just a small slice of the return pulse - in the diagram, it is shown as being controlled by a *range gate*, which tells it when to start and end conversion. The digitised information is held in a single 32 bit word and written to memory by a DMA machine. To get a picture of the area surrounding a target, the A/D is told to sample a specific range gate for a specific range of pulses; the result is a digitised image of that area in memory as a set of 1's and 0's; an example is shown in the diagram. Such an area is called a *window*.



Obviously, the A/D must be given the window specifications before the radar gets round to the desired angle; to do this, the equipment generates an interrupt to the processor every 45 degrees. At that moment, the processor must provide a pointer to a list of window descriptors, arranged in increasing angle order, for the DMA engine to provide to the A/D converter. Each entry in the list specifies a start angle, an end angle, a start range, a granularity indicator (which specifies one of four conversion bit rates - higher bit rates provide higher resolution images but cover a smaller range window) and a memory address into which to place the digitised image.

At each 45 degree interrupt, the system does two things - it takes the list of windows just processed, and gives that to software; and provides a pointer to a new list.

Finding the targets in the window is art, not science. One approach is to assume that a target is represented by a bunch of contiguous 'ones' marred by noise. To find the targets, then we filter out the high-frequency bits and keep track of the extents of areas of contiguous filtered ones. One way to do this is to select a small matrix size - say, 3 x 3 - and move it over the digitised window. The filter simply counts all the ones set under neath itself, and if more than a threshold (say, 6) bits of the 9 are 1, it sets a 1 in its centre position. If we move the window from near range to far along each pulse, we can then note when we get a run of contiguous ones; we'll note start and end of each such run provided that it's longer than (say) 3 ones. Then we repeat the exercise on the next pulse, and again we note the start and end of each set of contiguous ones. At the end of that pulse, we see if this pulse's set of runs 'overlaps' the runs in the previous pulse. If any do, we take note. By the time we've processed the whole window, we will have information about the extent of areas of contiguous ones, and any that extend for (say) more than 3 pulses we will take as indicating real targets. Given that we know the coordinates of the windows themselves, and the scaling factors, we can then convert these chunks of extent information into (range, angle) polar coordinates and hand the measurements off to the next stage of processing.

The tracking software uses the measurement to update its estimate of the position and velocity of the target; it does this by computing a weighted average of the measurement and its own prediction, weighting the measurement much more strongly for a new track, or one that is manoeuvering; and weighting the prediction much more heavily when the target is well-established and 'behaving'.

When there is just one target seen in the window, it is simple to associate the correct measurement with the correct track; when there are several tracks in a window then some sorting out must be done (this can happen as the targets' tracks intercept at the resolution of the radar). When there are multiple targets seen in the window, software must choose which target to associate with each track of interest; when things are going well, it associates the tracks by doing a simple least-squares fit - the allocation of target to track which results in the smallest sum of squares of distances between predicted positions for the tracks and targets seen is done. When there are fewer targets seen than tracks, the software will allocate the same measurements to several tracks.

Finally, we can look a little more closely at how the tracking software handles the measurements. One way to approach estimating the target's position, course and speed would be to take the measurements (which are effectively in (r, theta) form - range and angle - and convert them to cartesian coordinates giving the measured x and y values x_m and y_m. These could then be averaged with the predicted position of the target at the time of measurement, the predicted position be computed by keeping x, y, xdot and ydot (position and velocity in cartesian coordinates) for each target, providing a new estimate of position and velocity still in cartesian coordinates.

While this is computationally simple, the selection of reasonable weights is hard to do. Real things have position, velocity and momentum: this means that they can vary their speed at a different rate from their course -

think of a car, which can take several seconds to get from 10 to 60 mph, but can change course 30 degrees very quickly. To track a target better, it is helpful to have a movement model which matches what the real target can do - and so rather than tracking in fixed (x,y) cartesian coordinates, our tracker uses along- and across-track tracking. This is illustrated in the following diagrams. Figure 1 shows the situation after we have received a measurement for a target; we know where it was last time (at (x0, y0)), we know our best estimate of its course and speed, and we know how long it has been between the previous measurement and this one. We can therefore straightforwardly compute where we think the target should have been at the time of measurement, using simple trigonometry.

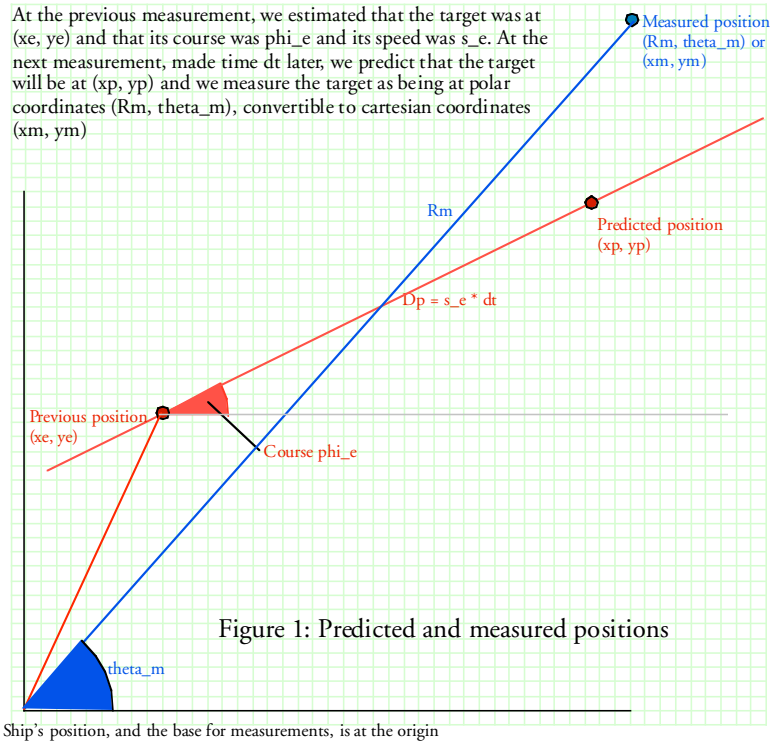


Figure 1: Predicted and measured positions

We get the measurements in polar coordinates (it's a radar). In Figure 1, we show the previous estimated position, the target's course, and the new measurement in a catesian space whose origin is the ship's position and whose y direction points north. The new measurement is provided in polar coordinates (Rm, theta_m).

We need to convert the measurement into along- and across-track coordinates. This is a cartesian coordinate system with the x-axis aligned along the estimated course, and lets us estimate velocity along the course and across it independently, matching the physics of real objects reasonably well. To transform to this coordinate system, we do the computations shown in Figure 2.

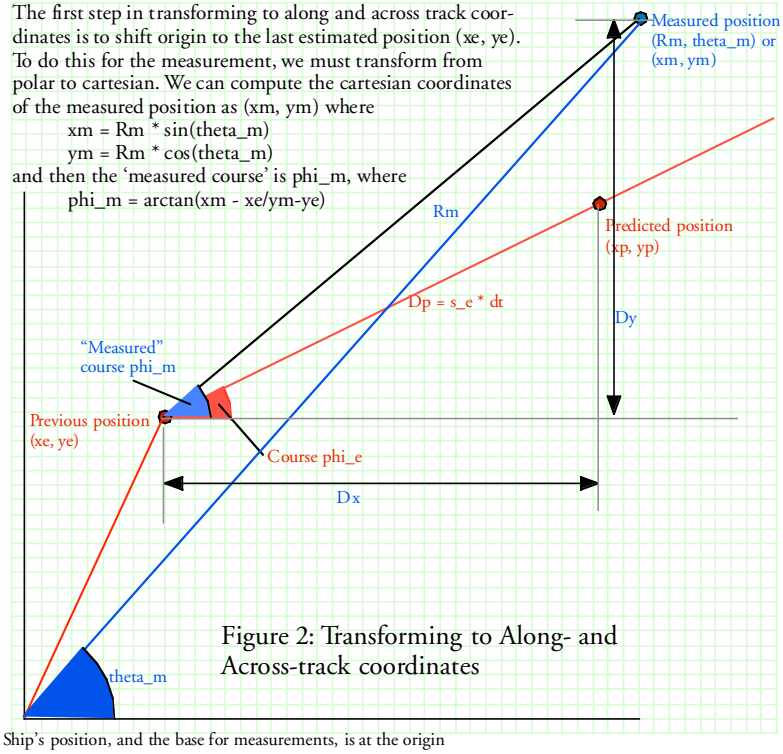


Figure 2: Transforming to Along- and Across-track coordinates

We first compute the cartesian coordinates of the measurement, using simple trigonometry. We then move the origin to the last estimated position of the target, and compute the apparent course *phi_m*. This is simply the angle given by arctan(y/x).

Now we need to form a new estimate of the target's position and velocity by

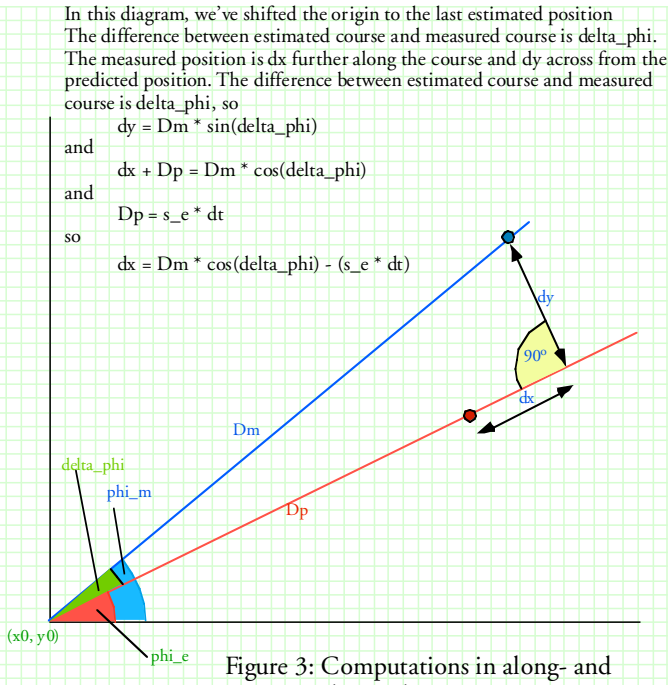


Figure 3: Computations in along- and across-track coordinates

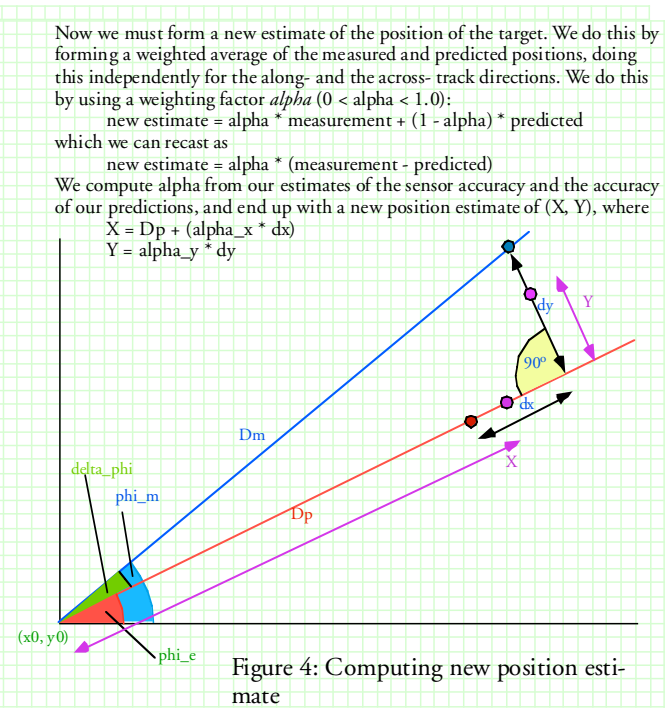


Figure 4: Computing new position estimate

combining our historical information (the estimated course and speed and estimated position) with the new measurement. To do this, we'll need the differences (dx, dy) in this coordinate system of the differences between predicted and measured positions. The computations are shown in Figure 3.

Now we need to form an appropriate weighted average of the measured and predicted position to for a new best estimate of its position. We choose a weight *alpha*, whose value will depend on how much confidence we have in our estimate of position; the alphas for along and across track can be different (if we have high confidence in speed but not course, for example). The computation is depicted in Figure 4.

We then do a similar exercise to compute the new estimates of speed and course, as shown in Figure 5, using a weighting factor *beta*.

Finally, we'll compute estimated position in global cartesian coordinates for use with the next measurement.

There's some magic involved in these computations, since there's no compelling model for how to compute the weighting values *alpha* and *beta*; however, with reasonably-trackable targets, things will settle down fairly well after some small number of measurements.

There are also other issues not touched upon in the figures: one example is that this system has a measurement behaviour analogous to the 'jaggies' seen on computer screens - the radar/analogue-to-digital converter system has a quantisation effect which needs to be handled in a real system. As with any digital system, the machinery presents its measurements to some granularity - in the case of a surveillance radar, the quantisation is some fraction of a degree in angle, and some linear distance in range. The effect is most marked for distant, slow-moving targets, which might occupy one 'bit' of range and angle for many, many scans, only to appear to move into and out of the next cell for some number of scans, and then to settle

Now we must form a new estimate of the velocity of the target. We do this by forming a weighted average of the measured and predicted velocities, doing this independently for the along- and the across- track directions. We do this by using a weighting factor *beta* (0 < beta < 1.0):

new estimate = beta * measurement + (1 - alpha) * predicted

We compute beta from its corresponding alpha, and end up with a new velocity estimate of (speed, course), where

speed = Dp + (beta_x * dx)

course = phi_e + (beta_y * delta_phi)

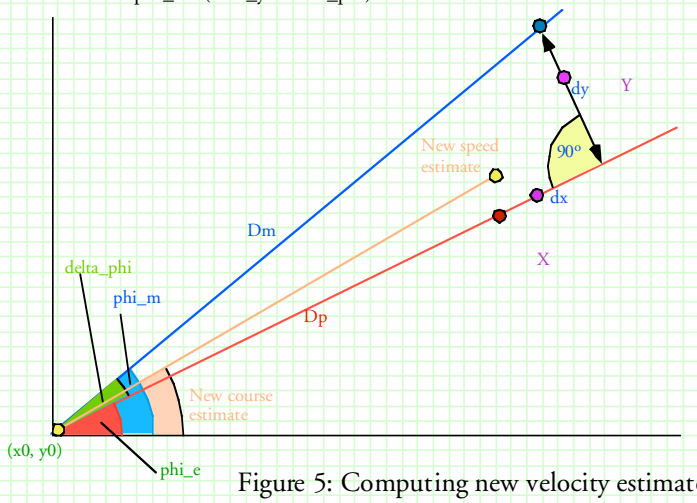


Figure 5: Computing new velocity estimate

into the next cell. If not handled appropriately, apparent accelerations of several *g*'s can be unnervingly 'observed' for surface vessels...

Multicore Issues

The autotracking system as described has limited capabilities, and minimises its need for computational performance in a number of ways. The first, most obvious tradeoff is to place windows around each target's expected position, rather than just digitising the whole sky. The next is the choice of using a simple binary A/D converter, rather than (say) a 10 bit video rate A/D. And the system only starts tracking stuff when told to by a human operator, rather than automatically starting tracks on new targets. And the tracking filter is both simple and needs very little state information; more powerful filters - such as the Kalman filter - were known at the time but were computationally unsupportable.

These decisions were made originally because of the available hardware (the whole CAAIS system, of which the radar autotracking was simply one subsystem) was based on a Ferranti FM1600B computer, a 24 bit machine with 32 kiloword (96KB) of core memory and a clock rate around one megahertz.

Of course, a more modern machine would provide a lot more capability and the algorithmic restrictions could be relaxed substantially. Nonetheless, it is interesting to consider how the system as described could be partitioned amongst multiple cores. We shall discuss this in greater depth in a later essay, but some fundamental observations may be made here:

- Firstly, we can *distribute* much of the computation - each target is independent, and so we can distribute much of the computation across multiple compute engines.
- Secondly, the computational work for a single target is a *pipeline*, and so we can distribute that work across a number of compute engines.
- And thirdly, there are some *special-purpose computations* which can be speeded up through dedicated hardware - the target extraction from the digitised radar window is one example, and the use of sines, cosines and square roots are others.

Related Code

C code which implements a version of the autotracking filter, including the computations necessary to compute alpha and beta, and a test harness is available.