# A 2.5 mA Wearable Positioning System[*]

Cliff Randell  Henk Muller  Andrew Moss
University of Bristol
http://www.cs.bris.ac.uk/

## Abstract

*In this paper we present a wearable positioning system that requires 2.5 mA to operate. The system consists of an infrastructure of ultrasonic transmitting devices, and a receiver device on the wearable. The receiver comprises an ultrasonic pick-up, an op-amp, and a PIC. The PIC implements a particle filter for estimating X and Y positions. The transmitter layout has been chosen to simplify the maths in the PIC so that the particle filter reduces to just two multiplication operations per particle. We have modified the traditional particle filter to operate in an environment with limited memory capabilities. The advantage of using a particle filter over other filtering techniques is that a particle filter is robust in environments with limited precision; all our data is stored as 8-bit. The number of particles is only limited by the amount of memory on the PIC; we only use 10% of the PIC's computational power.*

## 1 Introduction

One of the sensors used on many wearable systems is position. Various types of sensors are used in order to compute position; for example radio (GPS), radio and ultrasonics [7, 6, 5], just ultrasonics [3], or infrared [**?**]. Depending on the amount of noise in the data and on the required precision of position, some form of filtering algorithm needs to be applied. Popular choices for filtering algorithms are Kalman Filter [2] and Particle Filters [1]. The implementation of such a filter can be a computationally intensive task.

In this paper we look at ways of redesigning the filtering algorithm so that it suits small low-power devices. We have taken the particle filter approach as our starting point, and redesigned it so that it is suitable to run on a device with low memory foot print, small program memory, and low frequency. Our implementation provides an update rate of 12.5 Hz and can comfortably run on a 1 MHz 8-bit micro-controller, consuming 125 $\mu$A at 2 V. For experimental purposes we have implemented our system on a 20 MHz micro-controller, requiring 2.5 mA at 4.5 V.

We have used various techniques in order to fit the positioning system on the micro-controller. Our starting point was the Bristol Ultrasonic Positioning System [6]. In this system, a number of ultrasonic transducers are deployed in a grid-style configuration. This configuration has the advantage that only few arithmetic operations are required in order to retrieve X and Y locations from perfect measurements. In the case of imperfect measurements (noise, multi-path) some filtering is required.

We have extended the above system so that we can attain positions at a higher rate , as described in Section 2. In Section 3 we discuss the modifications we have made to particle filters in order to make them fit on a micro controller. We have then applied program transformation techniques in order to speed-up the code and make the timings predictable at the expense of program memory; since we have 2K of program memory available this is not an issue, and we specialise our program until it occupies all the available memory, as discussed in Section 5.

## 2 The system

The only way that we can implement a filter on an 8-bit micro-controller is by simplifying the mathematics. The system that we have designed comprises a number of transmitters that are configured along the two major axes of the system. Each transmitter in our system will emit a signal (a chirp) in turn. In order for our system to work, each transmitter should share either an $X$ or a $Y$ coordinate with the previous transmitter. In its simplest form, four transmitters can be configured in a rectangle, as is shown in Figure 1a. But a more complex arrangement, such as shown in Figure 1b. will also work.

Each transmitter transmits in turn, and the receiver measures the distance to the transmitter. Hence, in the example of the rectangle, chirps will be received from positions $(0, 0, C)$, $(0, A, C)$, $(B, A, C)$, and $(B, 0, C)$; where we assume that the origin of our coordinate system is under trans-

[*]Funding for this work is received from the U.K. Engineering and Physical Sciences Research Council, Equator Interdisciplinary Research Collaborartion, Grant No. GR/N15986/01.

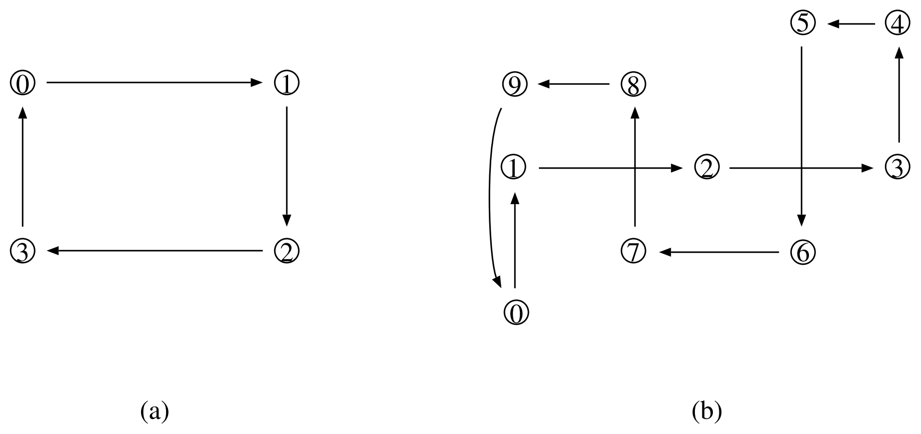


**Figure 1. Two sample configurations for transmitters: (a) is the minimum configuration; (b) shows a more complex arrangement.**

mitter $T_0$; $A$ and $B$ are the length and width of the rectangle; and $C$ is the height of the ceiling. If we assume that we are at a position $(x, y, z)$ in the room, and that we have measured a distance $d_2$ to Transmitter 2 and $d_3$ to Transmitter 3, then we can derive that:

$$\begin{aligned}
d_2^2 &= (x-B)^2 + (y-A)^2 + (z-C)^2 \\
d_3^2 &= (x-B)^2 + (y-0)^2 + (z-C)^2 \\
d_2^2 - d_3^2 &= (x-B)^2 - (x-B)^2 + \\
&\quad (y-A)^2 - (y-0)^2 + \\
&\quad (z-C)^2 - (z-C)^2 \\
d_2^2 - d_3^2 &= -2Ay + A^2 \\
y &= \frac{A^2 - d_2^2 + d_3^2}{2A} \qquad (1)
\end{aligned}$$

Which gives us a relation between $y$ and the measurements $d_2$ and $d_3$. Similarly, we can derive relations between $x$ and the measurements $d_3$ and $d_0$, etc. In fact, if we have a rectangular grid, and we chirp in turn in such a way that alternately either the $X$ or $Y$ coordinate of the chirp changes, then we get a set of equations that alternately predict $X$ and $Y$. In the case of the complex arrangement of transmitters, we get subsequent predictions for X, Y, Y, X, Y, X, Y, X, Y, X, X, ...

Since the distance between transmitters is a compile time constant, the only run-time operations we have to perform are squaring of each measurement, two subtractions, and a multiplication with $1/(2A)$ (which is also a compile time constant). This property is the basis for the design of the "Bristol Ultrasonic Location System" [6], where $A$ and $B$ are chosen carefully so that all operations except for the square operation are trivial. The problem with implementing the maths directly is that we lack resilience against noise and multi-path, and ad-hoc heuristics are required to get rid of those. In this paper, we show how Equation 1 can be used to control a particle filter that will filter out noise and echos. As already observed.

## 3 Filtering data

In order to cope with multi-path and noise, a form of *filtering* is required. Filtering helps by modelling the data, and modelling the expected noise in the system and the measurements. There are two popular filters, *Kalman Filters* [2], and *Particle Filters* [1]. A Kalman Filer requires a model of the position and how the measurements relate to the position, and given (noisy) measurements, the filter will estimate the position using the. A Particle Filter also uses a model for the state, and a probability density functions for measurements in this state. It uses a Monte Carlo approach by building a probability distribution of the state, given (noisy) measurements.

### 3.1 Particle filters

A simple way to view a particle filter, is that it estimates a multidimensional state vector $\vec{x}$, using a model of state progression, $\vec{x'} = f(\vec{x})$, and a likelihood function $p(\vec{z}|\vec{x})$. The state progression function $f$ models how the state will have progressed since the last time, and the likelihood function gives us the probability of this state given a measurement $\vec{z}$.

The particle filter maintains an array of $N$ states $\vec{x_i}, 0 \leq i < N$, (typically a few 100), and progresses each particle on every step. The likelihood is then computed for each particle for a particular measurement $z_j$, $p(\vec{z_j}|\vec{x_i})$, which is used to build a PDF of the state space. The state space

**Process measurement z[j]:**
```
   For i from 0 to N-1
      progress particle[i]
      weight[i] *= likelihood particle[i] given z[j]
```
**Resample:**
```
   cdf[0] = weight[0];
   For i from 1 to N-1
      cdf[i] = cdf[i-1] + weight[i]
   For i from 0 to N-1
      newParticle[i] = particle[random based on cdf]
   particle = newParticle
```

**Figure 2. Pseudo code of a particle filter**

**Process measurement z[j] & resample:**
```
   previous = likelihood particle[0] given z[j]
   For i from 1 to N-1
      progress particle[i]
      likely = likelihood particle[i] given z[j]
      if random * (likely + previous) $>$ likely then
         particle[i] = particle[i-1]
      else
         previous = likely
```

**Figure 3. In-situ pseudo code**

can be *resampled*, so that parts of the state space with a high probability are assigned more particles than unlikely parts of the state space. Pseudo code for the particle filter algorithm is shown in Figure 4.

The strengths of a particle filter in comparison with a Kalman Filter are:

- It can model systems where the distribution of errors is not Normal,

- It does not require a Jacobian to be computed. A Kalman Filter needs the derivative of the measurement equation to the measurement state. This may be hard to compute for certain measurement functions.

- A Particle Filter is robust in that it will cope better when the model is not completely correct. For example, if one were modelling a positioning system, and the model assumes constant velocity with a limited acceleration, then a particle filter will recover quickly when the real acceleration is beyond the bounds of the model, whereas a Kalman filter may go off on a tangent.

- A particle filter is robust in the presence of rounding errors and limited precision. Kalman Filters often require high precision computations, and may be numerically instable in the presence of rounding errors.

The strength of a Kalman Filter is that it converges quickly to an optimal solution if one can compute the Jacobian, errors are normal, and the model is correct. In addition, a Kalman filter typically requires less memory resources.

## 3.2 Resampling the Particle Filter In-situ

There are two problems that we must address in order to make a particle filter useful on devices with limited resources. In a particle filter one normally first computes the *cumulative distribution function*, CDF, for the particles. Given the CDF the particles can be resampled, computing a new set of particles for the next iteration. Calculating the CDF requires us to store a probability with each particle, and resampling requires us to duplicate the storage required for particles, as we are sampling one distribution to create a new distribution.

Because the process of computing the CDF and resampling is too expensive, we have devised an *in-situ resampling method* which avoids computing the CDF, and stochastically resamples. Our in-situ resampling method is executed simultaneously while updating the particles themselves. In short, we update a particle, compute its likelihood, and then take a nondeterministic decision: we either keep this particle or we remove this particle and replace it with the previous particle. This process is shown in pseudocode in Figure 3.

We use the standard state progression for a the particle filter, and the standard method for computing the likelihood of a particle. Given the likelihood of the current and previous particle, we will take a decision as to whether the current particle is allowed to progress. Suppose the likelihood of the previous particle is $p_n$ and the likelihood of the current particle is $p_m$, then we will allow the current particle with a probability of $p_m/(p_m + p_n)$, and the previous particle with a probability of $p_n/(p_m + p_n)$.

If the current particle is allowed to progress, the algorithm proceeds with the next iteration. If the current particle is not allowed to progress, we replace the current particle with the previous particle, and we continue to the next iteration. Using this method, we have to remember only the likelihood of the previous particle. A particle with a very high likelihood is likely to be replacing a large number of subsequent particles. A particle with a very low likelihood is likely to be replaced. Note that the algorithm above never replaces particle 0; this is rectified by means of *pseudo random stepping*.

```
step = 1; k = 1
For j from 0
   Obtain z[j]
   step = step + 4; if step > N then step -= N
   For i from 0 to N-1
      progress particle[k]
      likely = compute likelihood particle[k]
      if random * (likely + previous) > likely then
         particle[k] = particle[old]
      else
         previous = likely
      old = k
      k = k + step; if k > N then k -= N
```

**Figure 4. Full in-situ algorithm**

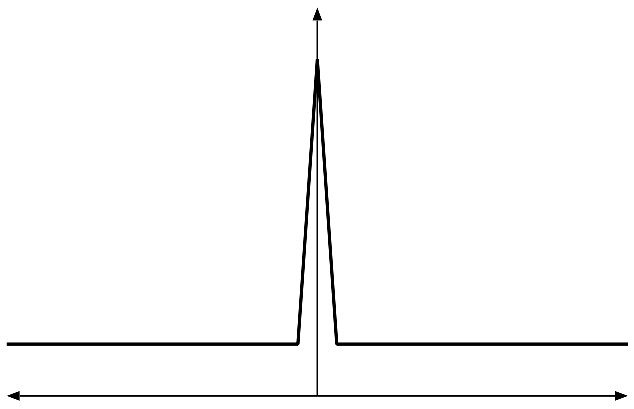


**Figure 5. Likelihood function**

### 3.3 Pseudo Random Stepping

The method above does not work very well since the algorithm iterates through the particles in the same order on every step of the filter. For example, if two good particles happen to be located next to each other in the array, then either the latter one will be replaced with the first one, or the former one will not be allowed to replace any particles.

We solve this issue by stepping through the array of particles in a pseudo-random order. We maintain a list of numbers $P_0, P_1, ...P_{M-1}$ that are relative prime to the number of particles N, and get our particle filter to operate on particle $j + iP_{j \bmod M} \bmod N$ on the $i$th iteration of step $j$. Both modulo operations can be implemented using a conditional subtraction. And if we pick $N$ to be a power of 2, then any odd number will do for $P_j$. The full algorithm is shown in Figure 4.

## 4 Modelling position using the modified particle filter

There are two candidates for the state of a particle: the position and the position and a speed. I.e., we can define the state to be either of

$$\vec{x} = \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$

or

$$\vec{x} = \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix}$$

The state transition functions can be either of:

$$\vec{x} = \vec{x} + \vec{\epsilon}\Delta t$$

or

$$\vec{x} = \begin{pmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \vec{x} + \vec{\delta}\Delta t$$

Where $\vec{\epsilon}$ is a random deviation of the position, or $\vec{\delta}$ is a random deviation of position and velocity. Note that the first state progression function involves two additions, and the second state progression function involves 6 additions. In addition, random numbers need to be generated.

The likelihood function comes from Equation refequation:square. This relates the $y$ coordinate to a measurement as:

$$y = \frac{A^2 - d_2^2 + d_3^2}{2A}$$

The difference between the left hand side and the right hand side gives us a measure for the likelihood that two subsequent observations $(d_2, d_3)$ from transmitters that are a distance $A$ apart along the $Y$-axis, support a $Y$-coordinate $y$. This difference must then be mapped on a likelihood. We use a clamped sawtooth function shown in Figure 5. The slopes of the sawtooth have a slope of 1 in 16, so that we can implement them by means of shift right.

Given a collection of $N$ particles we have to perform the following computations for each set of measurements.

- Square the measurement, and store it for future use.
- Perform two subtractions to compute the denominator of the above equation.
- Multiply by $\frac{1}{2A}$
- Then perform a subtraction for each particle.
- Compute $F$ for each particle, involving 4 shifts and two subtractions.

- Add the probability of the previous particle and multiply with a random number.

Each particles is stored in a single byte (position model) or two bytes (position and velocity). Because we perform in-situ resampling, we do not have to store a weight for the particle, and we do not have to store copies of the particles. So in the position model, given $M$ bytes of memory, we can use at most $M/2$ particles for $X$ and $M/2$ particles for $Y$.

## 5 Implementation

We have chosen the position-only model for implementation on the PIC. The advantage of this model is that all states are one single byte. The byte for the coordinate is measured in units of 3.43 cm (100 $\mu$s at the speed of sound). This means that the largest area we can cover measures $256 \times 3.43$ cm $= 8.7$ by $8.7$ metres.

The infrastructure that we used for testing comprises a 2x2m grid of transmitters, transmitting with a periodicity of 25 Hz; each transmitter transmits 40 $\mu$s after the previous transmitter. In addition, once every 160 $\mu$s an RF ping is transmitted in order to synchronise the clocks on transmitter and receiver. No data is carried on the ping.

The PIC outputs position as a textual stream over RS-232 at present. No attempt has been made to parallelise the code for RS-232 operation and the code for ultra-sonic reception. In between each iteration of the position system, the RS-232 code operates sequentially. This latency could be removed by threading the two codes together.

### 5.1 Micro controller

We have implemented the programs on a PIC 16F628A micro-controller. This micro-controller has 2048 words of instruction memory, and 224 bytes of RAM. It has an eight-bit instruction set with operations for ADD, SUB, and single bit rotate. In particular, there is no multiplication hardware. The PIC takes 2.5 mA when running at 20 MHz and 125 $\mu$A when running at 1 MHz.

The address space of the 224 bytes of memory is non contiguous, with two banks of 64 bytes that can be accessed using an array index. This limitation has led us to use the position-only model to allow 64 particles of one byte. The X and Y coordinates each use a separate filter; 64 bytes in the first bank store the state of the X particle filter, and 64 bytes in the second bank store the state of the Y particle filter.

A choice of position and velocity for the state of the filters would only allow 32 particles to model each 2D state space. The reduction in particles whilst increasing the degree of freedom in the system would reduce the robustness and accuracy of the filter.

### 5.2 Random numbers

Our particle filter relies on a steady stream of random numbers. We need to generate 128 random bytes 25 times a second. A good source of random numbers on our PIC is the RF receiver. When no data is received, the RF receiver is generating a random sequence of 0s and 1s. When sampled at rates above 10 KHz, the randomness is impaired and streams of 1s and 0s appear. For this reason, we use a linear feedback shift register (a Tausworthe generator), in addition to the RF hardware. Every time a random number is needed, the shift register is rotated right, and the left most bit is reseeded with $b_1 \oplus b_4 \oplus b_{RF}$. As with many Tausworthe generators, there is some correlation between subsequent random numbers, but since we step through our particles in a pseudo random fashion, this does not turn out to be an issue.

### 5.3 Multiplications

The micro-controller does not contain hardware to perform multiplications. These operations must be implemented in software. There are many possible multiplication routines depending on the size and format of the numbers. Multiplication of signed numbers requires a few extra steps, and fixed point fractional numbers require different normalisation steps. One example routine for an unsigned 8-bit by 8-bit into 16-bit multiplication is shown in Figure 6.

Bæverfjord has used many techniques to reduce the size of the routine. In particular the shifting operations that must be computed in each cycle (as the PIC can only shift numbers one place) are used to control the loop iteration, saving a few cycles on each pass. Despite the optimisation employed in the routine the loop still takes eight cycles for each iteration, with one less cycle in the final iteration taking 67 cycles total and two extra for the return. Of the eight cycles in each iteration three are used to perform the multiplicative steps, with the other five being the overhead of loop control.

In the case where we multiply with a constant (which is half the number of multiplication operations), we partially evaluate the multiplication routine [4], and generate a piece of straight line code that multiplies by just that constant. All of the loop control overhead is removed from the output routine. For example, multiplication by `0.101` (0.625) would normally involve setting `mulplr` to 5, calling `mul8x8s` and then shifting the result right by three places. But partial evaluation will create a program that is shown in Figure 7.

Partial evaluation creates a tradeoff between the speed of execution, and the amount of memory required for the program. The generation of code that is performed during the evaluation will perform several types of optimisation. Dead

```
; Small - 67 cycles constant,
; 11 instructions - Bjorn Baeverfjord

mul8x8s clrf    prodH
        movlw   .128
        movwf   prodL
        movfw   mulcnd
mulsl   rrf     mulplr
        skpnc
        addwf   prodH
        rrf     prodH
        rrf     prodL
        skpc
        goto    mulsl
        return
```

**Figure 6. An example multiply routine; verbatim from** `http://sciencezero.4hv.org/computing/picmul8x8.htm`**.**

code, which cannot be executed, is eliminated. Constants are propagated through operations, which reduces the time taken for some operations. The most important optimisation is that conditions in the program that can be evaluated at compile-time are removed. This optimisation automatically unfolds loops and removes if-then-else clauses.

Each of these operations reduces the time taken to execute a program, but some of them increase the number of instructions that are generated, increasing the memory required to store the program. In our application time is at a premium — it is not possible to perform all of the operations required for a particle filter in the number of clock cycles available. There is excess space available in the $2048$ instruction program memory. We utilise this tradeoff by attempting to optimise the execution time of the program, while fitting within the limit on the program memory.

Another advantage of partial evaluation is that timing analysis becomes easier. The multiplication shown in Figure 6 has a varying length of execution depending on the value of `mulplr`. When the program is partially evaluted for a fixed value, the result is straight line code which has a fixed timing. As the constants that fix the timing are known then we can analyse the precise time necessary. The partial evaluator produces a program that requires $17+h$ clock cycles, where $h$ is the number of bits set to one in `mulplr`. If, for example, we use a system with a grid that is 0.8 by 0.8 metres, then we need to multiply by $1/(2\times 0.8) = 0.625$ or $10100000$, which requires 19 cycles; approximately $28$% of the cycles taken by a non specialised version.

Indeed, the full program does more than just perform the computations, such as a loop over all the particles. The

```
mul0101 clrf   prodH
        movf   in,0
        rrf    prodH
        rrf    prodL
        rrf    prodH
        rrf    prodL
        rrf    prodH
        rrf    prodL
        rrf    prodH
        rrf    prodL
        addwf  prodH
        rrf    prodH
        rrf    prodL
        rrf    prodH
        rrf    prodL
        addwf  prodH
        rrf    prodH
        rrf    prodL
        rrf    prodH
        rrf    prodL
```

**Figure 7. Specialised multiplication by 0.625**

overhead of this is quite small though. Most importantly, this approach allows us to *size* the micro controller required to its absolute minimum. The position system above can run of two AA batteries for more than a month continuously.

## 6 Results

We have performed four experiments. Three experiments on real hardware, and one controlled experiment on our simulator. The three experiments on the real hardware have been used to establish the static and dynamic errors of the position system, and to measure the power consumption of the hardware. The experiment on the simulator has been used to establish its performance in the light of systematic echos.

### 6.1 Static performance

For the static performance we left the receiver near the origin of our coordinate system, and measured the X and Y position for a couple of seconds. A histogram of the measured points in shown in Figure 8. The mean X coordinate is 9.7 cm offset to the origin, and the mean Y coordinate is 6.4 cm offset. The standard deviation is 2.5 cm on both; which is acceptable given that we measure our coordinates on a grid with a resolution of 3.4 cm.

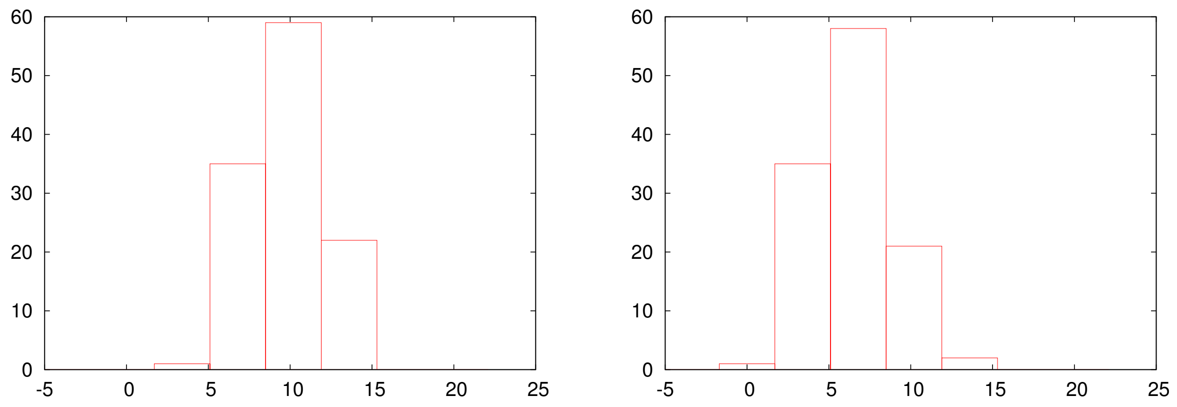


**Figure 8. Static performance of the receiver**

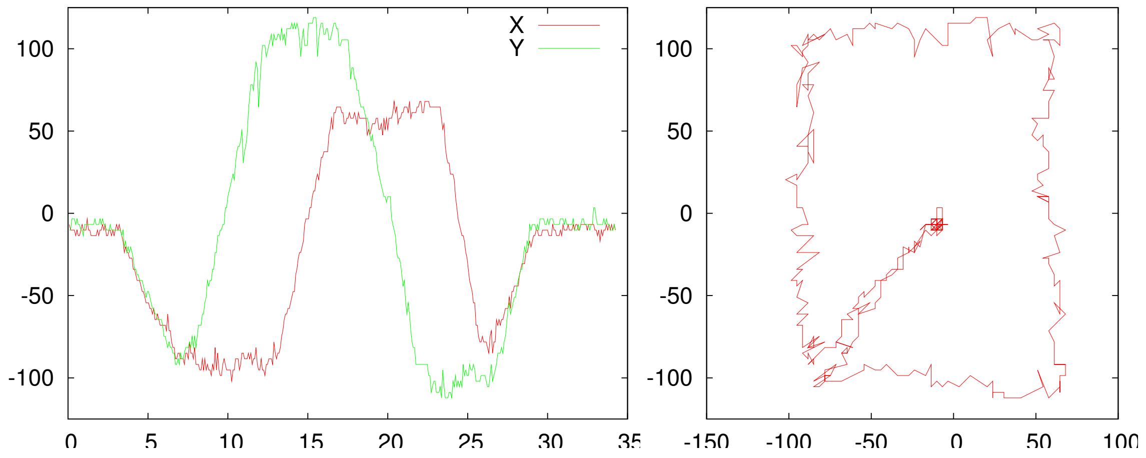


**Figure 9. Dynamic performance of the receiver**

### 6.2 Dynamic performance

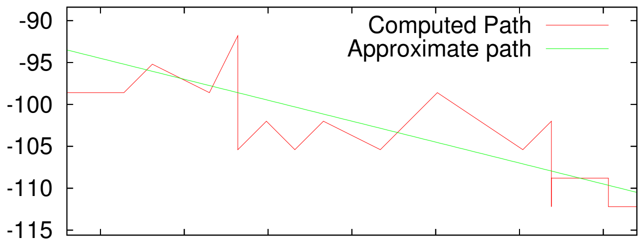
We did not formally measure the dynamic performance of the receiver, but did some preliminary experiments in order to establish its dynamic behaviour. We moved the receiver along a rectangular track, and measured the X and Y coordinates. The filtered output positions were recorder, and are shown in Figure 9. The left hand figure plots X and Y (in centimetres) against time (in seconds), the right hand plot plots X against Y in centimetres.

We can see that the edges of the rectangle are rather jagged. One reason for this is that in the current implementation of our particle filter we have a known rounding error, which causes the particles to be moved up and down on every other iteration of the filter. This is shown in Figure 10a which provides a close-up of part of the track. Figure 10b shows the same path, but just with the "even" points of the path, and Figure 10b shows just the "odd" points of the same path We can see that those two paths are smoother. Our static measurements don't have this problem because they are measured in the origin, where this rounding error has only a negligible impact.

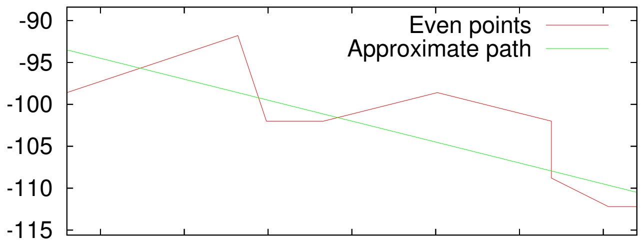
### 6.3 Resource usage

We have measured the resources required for our implementation. Our implementation requires 75% of the PIC's program memory, and 70% of the PIC's data memory. We can extend the system with another two transmitters without running out of program memory. The computations on the particle filter take 2 ms on a 20 MHz PIC. Given that we need 40 ms between subsequent chirps (in order to wait for echos to disappear), we can execute our program on a 1 MHz PIC.

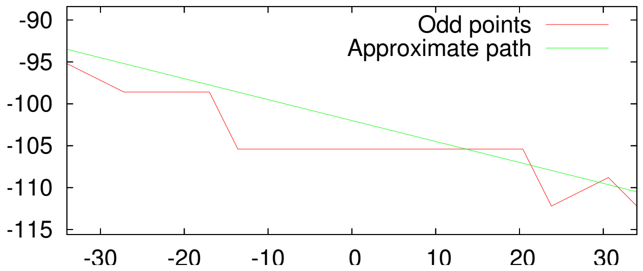
On the 20 MHz PIC the power consumption of the unit is .... mA, on which 2.5 mA is consumed by the PIC. With a 1 MHz PIC the latter can be reduced to 125 $\mu$A, reducing the power consumption of the whole unit to .... $\mu$A.



**(a) Close-up of path.**



**(b) Close-up of path, even points only**



**(a) Close-up of path, odd points only.**

**Figure 10. Dynamic performance of the receiver**

### 6.4 Multi-path

## 7 Discussion and Future Work

In this paper we have shown how we can modify a relatively complex filtering algorithm to run on an low power embedded micro controller, making it suitable to run off a battery.

The system successfully deals with noisy input; it filters noisy readings out without a problem, and it can also deal with multi path measurements. We have also observed that the system can successfully create a bi-modal distribution of the position, for example in the case where one reading is consistently a multi-path reading. However, in that case our filter presently returns the mean of the distribution rather than the mode; which makes the returned position meaningless. We are working on a system that will produce the mode, which is non-trivial given the lack of memory.

Lack of memory has also meant that we had to abandon speed. Modelling speed is feasible computation wise, but we do not have enough memory to store a reasonable number of particles sampling speed and position. Experiments on a simulator with more memory shows that we need at least 100 particles, which would require a micro-controller with 420 bytes of ram; preferably stored in a contiguous address space. Incorporating speed would make the advantages of filtering more obvious. If we had sufficient memory, we would still only require 10% of the compuational power of a 20 MHz PIC micro-controller, which would allow us to run on a power budget of close to 250 $\mu A$.

## References

[1] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for on-line non-linear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, pages 174–188, Feb. 2002.

[2] R. E. Kalman. A New Approach to Linear Filtering and Prediction. In *Journal of Basic Engineering (ASME)*, pages 82(D):35–45, March 1960.

[3] M. McCarthy and H. L. Muller. RF Free Ultrasonic Positioning. In *Seventh International Symposium on Wearable Computers*. IEEE Computer Society, October 2003.

[4] A. Moss and H. Muller. Efficient code generation for a domain specific language. In *Generative Programming and Component Engineering*, pages 47–62. Springer Verlag, September 2005.

[5] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *Mobile Computing and Networking*, pages 32–43, Aug. 2000.

[6] C. Randell and H. Muller. Low Cost Indoor Positioning System. In G. D. Abowd, editor, *Ubicomp 2001: Ubiquitous Computing*, pages 42–48. Springer-Verlag, Sept 2001.

[7] A. Ward, A. Jones, and A. Hopper. A New Location Technique for the Active Office. In *IEEE Personnel Communications, volume 4 no.5*, pages 42–47, Oct. 1997.