Faculty of Engineering and Material Science
German University in Cairo

Conductor's baton prediction using radar sensor

By
Ahmad Hamdy Sayed Hassanien

Supervised by
Prof. Dr.-Ing. Peter Ott

Timeline
03.01.2020 – 31.08.2020

This is to certify that:
(i) the thesis comprises only my original work toward the Bachelor of Science (B.Sc.) at the German University in Cairo (GUC),
(ii) due acknowledgment has been made in the text to all other material used

Ahmad Hamdy Sayed Hassanien
30 August, 2020

Acknowledgements

First, I would like to thank my supervisor, Prof. Dr.-Ing. Peter Ott , for his generous support during my work on this thesis and for his excellent guidance, patience and for encouraging me and pushing me forward. I would like to thank the German University in Cairo and Hochschule Heilbronn for giving me this opportunity, to do my bachelor thesis abroad in Germany. I would like to thank member of the Labor Technische Optik, Dimitrii Kozlov, for his generous tips and support. Finally, I would like to express my gratitude to my parents and friends for their continuous encouragement.

ABSTRACT

In this project we be using a radar for human gesture detection, in order to allow a user to conduct a virtual orchestra. The radar system used, Mikrosense, provides RD maps, from which both range and velocity can be determined.

The live data from the radar is fed to a particle filter, which has a mathematical model for hand movement. The particle filter continuously tries to adjust the parameters of the model, in order to match the actual movement as closely as possible. This allows us to extract certain features of the hand movement, such as amplitude and frequency, without being directly measured. Using our model, we can when the next down beat will occur.

The particle filter communicates with a video and audio controller. Based on the current location in music, the controller adjusts it's speed, so that the music ends up aligning with the conductor not only in speed, but also in phase.

Contents

List of Symbols

The meanings for various symbols used in this document are explained here

Symbol	Unit	Meaning
y	mm	Radar Range
v	**mm/s**	Radar Velocity
u	-	Control data
z	-	Measurement data
h	-	Measurement function
bel	-	Belief
w	-	Particle weight
Q	-	Process Noise
R	-	Measurement Noise
η	-	Normalisation factor
μ	-	Mean

1 INTRODUCTION

1.1 Motivation

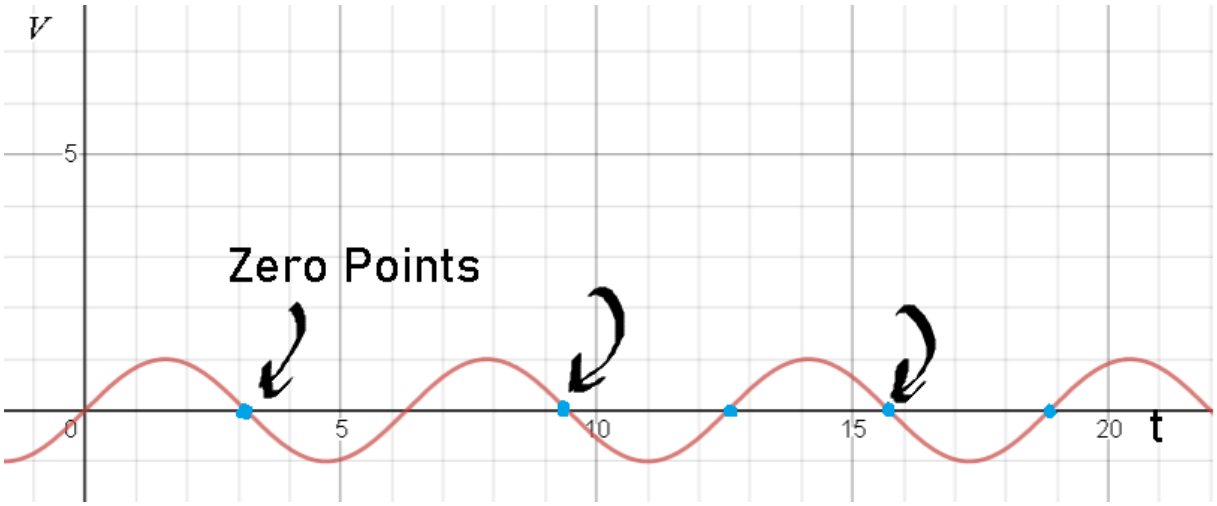
The modern FMCW radar, also known as Frequency Modulated Continuous Wave technology is used extensively in the field of automotive applications. This project is part of project Mikrosens. Project Mikrosens is a collaboration between multiple universities under The University of Ulm. The goal of the project is to create uses and applications for radar technology on the micro scale[10]. While this radar technology has been recently mostly used in driver-less cars, the goal of this project is to explore other applications, such as the speed measurement of river flow in order to predict natural catastrophes before they happen. Through the examination of the reflected wave, the radar sensor was also used to identify the state of earth's surface (dry/snow covered/ raining, etc) .

This project will mainly focus on developing a system, that can understand the motion of a waving hand, and thus be able to predict it's movement in the future based on a mathematical model.

Our use case will be a virtual orchestra, where a user can conduct with their hand infront of a screen. The goal of the project is to correctly predict when the next down beat will be, and based on that, adjust the orchestra's speed accordingly.

1.2 Overview

1.2.1 Previous System



This project is an upgrade based on a previous project, also used radar technology. However a limitation in the previous project was the lack of beat prediction capability, so the system could only capture the tempo of conduction, however without locking the downbeat of the conductor with the orchestra. A zero crossing algorithm was implemented, and with that information the frequency could be determined by measuring the distance between the points of zero velocity. In this project, we aim to add on this, by predicting when the next point of zero velocity will happen (but only on the down beat).

1.2.2 Virtual Orchestra

The idea for the project is to use radar technology[34] to allow a person to conduct a virtual orchestra. A prerecorded video of an orchestra is played on the screen, and the speed of the playing is adjusted so that the orchestra follows the conductor's hand in speed and phase. Similar experiments[18][20][22][41][43][48] have been previously done with different setups, such as using motion sensors [6].

Fig 1.2: Haus Der Musik

A similar idea can also be seen at the House of Music Vienna[42].

The goal of the project is to showcase the power of radar technology in game form[46][51].

Fig 1.3: Pre-recorded orchestra used in project

In order to sync the orchestra with the conductor, we need to be able to predict when the next downbeat will occur and adjust our playing tempo accordingly. Beat prediction has also been used to reduce the risk of heart surgery[25], where the fast motion of the beating hear can pose a serious challenge.

The aim of this project is to use the radar sensor to control a given video of played music. By actively learning the movement pattern of the hand, while simultaneously adjusting the video feed's speed and phase, the conducting of the video will feel as natural as conducting with a real orchestra.

Other ideas for data collection [20] used a 3d sensor attached to the waiving hand. We chose not to use this approach, as the wireless radar system would be less prone to damage and is therefore more robust.

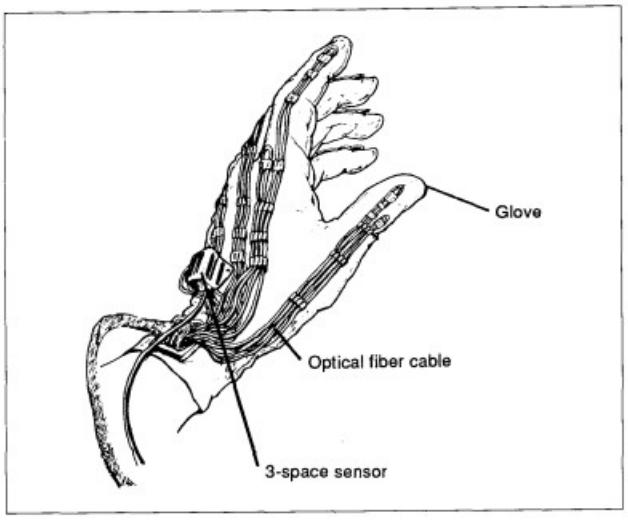


Fig 1.4: Data glove idea [20]

1.2.3 FMCW radar Mikrosens

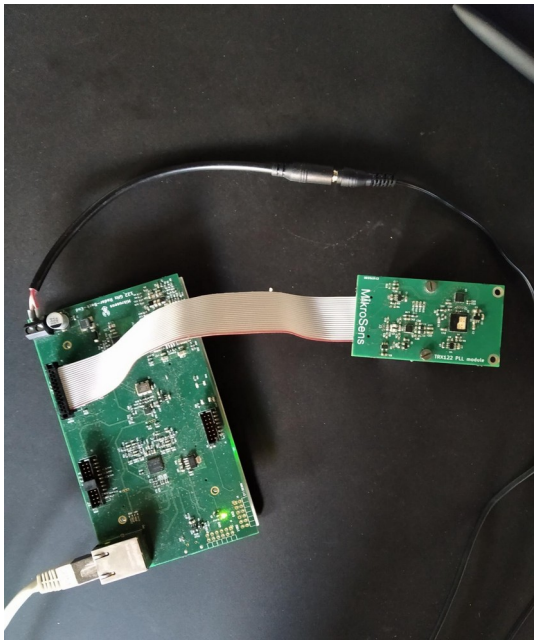


Fig 1.5: Mikrosens FPGA

The Frequency Modulated Continuous Wave Radar basically builds on the continuous wave radar. One drawback of a continuous wave radar is that it cannot determine the range. The idea of the FMCW radar is to send a signal of increasing (or decreasing) frequency (or phase). This is called modulation. When the signal is received as an echo from an object, there will be a delay in the change of frequency.

Some advantages of the FMCW radar are:

-It can measure very small ranges to target
-We can get the target range and velocity at the same time (this offers a serious advantage for improving the accuracy of our prediction, which we will se later on)
-High range resolution

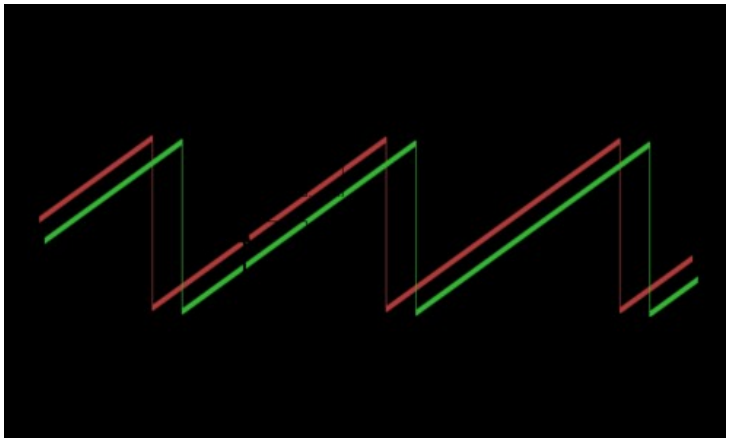


Fig 1.6: Example of transmitted (red) signal and returned (green signal)

2 STATE OF THE ART

2.1 Probability Theory

2.1.1 Notation clarification

To match other papers/books that deal with probabilistic robotics, I will be using the following notion:
z_t : A measurement data point at time t
x_t: A state at time t
u_t: A control data point at time t

For example, our posterior distribution, given a control input u_t, will be written as $\mathrm{p}(x_t \mid x_{t-1}, u_t)$.

Conditional Probability:
The probability of event A, given that we know event B has happened, is written as:

$$P(A \mid B) = \frac{P(A, B)}{P(B)} \tag{2.1}$$

Bayes rule:

$$p(x \mid y, z) = \frac{p(y \mid x, z)p(x \mid z)}{p(y \mid z)} \tag{2.2}$$

2.1.2 Bayes Rule

In probability theory, we say that the chance of random variable X being the exact value x,, given that it is known that the random variable Y is currently a known value, y, is:

$$p(x \mid y) \tag{2.3}$$

This is called our posterior probability distribution.
Where x is the quantity we want to know (a state of our machine or robot), given that we have the quantity y (some measurement from a sensor). How do we find this? With Bayes rule[7]:

$$p(x \mid y) = \frac{p(y \mid x)p(x)}{p(y)} \tag{2.4}$$

The probability

$$p(x) \tag{2.5}$$

is the prior probability distribution, and

$$p(z \mid x) \tag{2.6}$$

is our likelihood function. The factor

$$p(y)^{-1} \tag{2.7}$$

does not depend on x, it is the normalizer variable.

Example, in a population:
P(cancer)=0.25%
P(symptoms of cancer)=0.3%

given that we know that, if some one has cancer, they have a 90% of showing symptoms. And some random person walks into a hospital, showing symptoms. Do they have cancer? The chance of cancer for this individual would be:

(0.9*0.25)/0.3

*In simple terms, a **prior** probability distribution describes the state or robot or machine, in the previous step. When we move forward in time, our states would change, the distribution describing the state would change. This new distribution is the **posterior.**

2.1.3 Total Probability Theorem

Is the total probability of an event happening, considering that it may arise from different sources. For example, in a warehouse there are 100 bulbs. 60% come from company Y_1, and 40% come from Y_2.

90% of bulbs made by Y1 are white, and 10% are red.
95% of bulbs made by Y2 are white, and 5% are red.

The probability of a red bulb in the warehouse, regardless where it comes from, is the probability of a red bulb for a certain company, times the chance of a bulb from that company. Then we repeat for all the other companies and sum. This idea is mathematically expressed as:

$$\begin{aligned} p(x) &= \int p(x, y) dy \\ &= \int p(x \mid y) p(y) \end{aligned} \tag{2.8}$$

The Gaussian Function can be described by the following probability density function[3]:

$$p(x) = \left(2\pi\sigma^2\right)^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right\} \tag{2.9}$$

Where
σ: The variance of the distribution
μ: The mean of our distribution

2.1.4 Markov Assumption

The Markov assumption essentially states, that in a Bayesian network (as the one shown below), a node can only be affected by nodes it descends from, and can only affect nodes that descend from it.

So for instance, in the diagram below, the measurements Z descend from the corresponding state. That means they only depend on said states. We will use this assumption to simplify our math later down the line.

When a state follows the Markov assumption, meaning that knowledge of past states or controls carry no additional information that would help us to predict the future more accurately, we call this a complete state.

There are two important definitions that describe a system:

Deterministic: This means that given a certain control signal, the future state that we will move into can be known with absolute certainty. For example, if you unplug a bulb (control) it's current state (ON), when moving to the next state (OFF), will be deterministic. There are no in-between, the output result is fully determined by the input. The probability of the next state being OFF is 100%.

Stochastic: This means that the process inherits some randomness. For example, throwing a ball in non-ideal, real world conditions. Mathematically, if you know the weight of the ball, speed at which it is thrown, angle of departure and so on, there is an exact point at which it will land (say, at 10 meters). But because this is a stochastic system, the landing location becomes a probability density function, rather than just a point. IE: there is a 70% it will land exactly at 10 meters, a 20% chance at 8 meters, and so on.

It it important to notice that our definition of completeness does not require the future to be a deterministic function of the state, and Temporal processes that meet these conditions are commonly known as Markov chains.

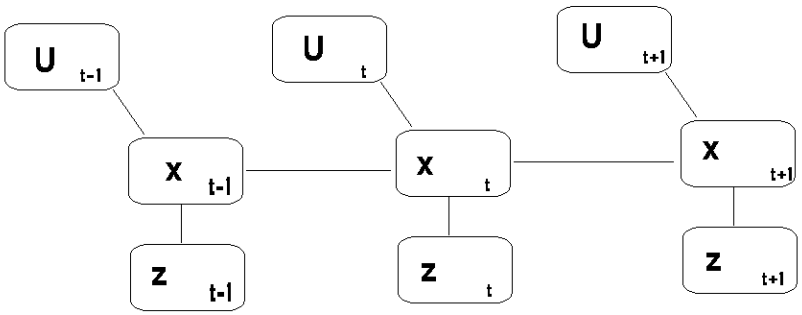


Fig 2.1:Markov Chain

Based on this, we can make some simplifications:

$$p(z_t \mid x_{0:t}, z_{0:t-1}, u_{1:t}) = p(z_t \mid x_t) \tag{2.10}$$

$$p(x_t \mid x_{0:t-1}, z_{0:t-1}, u_{1,t}) = p(x_t \mid x_{t-1}, u_t) \tag{2.11}$$

2.2 Probabilistic filters

Probabilistic filters such as Kalman Filters and Particle Filters are an essential development in Modern Engineering. They are an essential development in autonomous applications such as self driving cars, and are also used in sensor fusion applications. An example of sensor fusion is GPS for navigation, where the data from multiple sources is combined to get a better estimation than just one.

In our project, we will be using them to determine states that cannot be directly known. (The states describing the conductor's motion).

Why did we choose a particle filter? Let us explore our options. For a typical object-tracking problem, we have the following possible candidates [35]:

Filter	Properties
Linear Kalman Filter	For Linear Problems Gaussian noise Uni-Modal
Discrete Bayes Filter	Uni-variate Discrete Multi Modal Non Linear
Unscented Kalman Filter[36]	Non Linear Continuous Multivariate
Extended Kalman Filter[37]	Non Linear
Particle Filter	Non Linear Multi Modal

2.2.1 Bayes Filter

We will need to understand the discrete Bayes filter before we proceed. Let's derive it mathematically. Our goal is, given the previous posterior, to get the posterior for the next time step. Given of course that we have access to all previous control signals, u, and measurements, z.

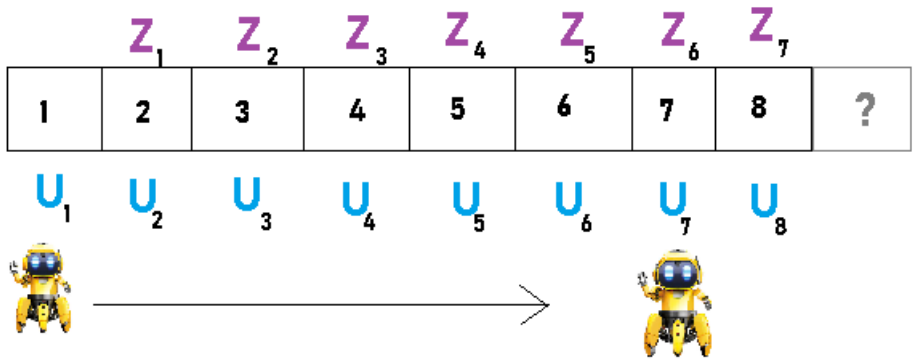


Fig 2.2: Ilustration of the Bayes filter for a hallway robot

Let's first consider our next posterior $p(x_t \mid z_{1:t}, u_{1:t})$

Using Bayes rule, we can:

$$p(x_t \mid z_{1:t}, u_{1:t}) = \frac{p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) p(x_t \mid z_{1:t-1}, u_{1:t})}{p(z_t \mid z_{1:t-1}, u_{1:t})} \tag{2.12}$$

Let simplify a bit.

We know that $p(z_t \mid z_{1:t-1}, u_{1:t})$ is a normalizer, so:

$$p(x_t \mid z_{1:t}, u_{1:t}) = \eta p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) p(x_t \mid z_{1:t-1}, u_{1:t}) \tag{2.13}$$

We can still simplify further. We assume that our states are complete, therefore, the marokvian assumption applies. That means we don't need all the previous states, but instead only the previous state, combined with the previous control signal, to predict the next state. Another rule from our assumption of state completeness, is that a measurement only depends on the state of the corresponding time.

$$p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) = p(z_t \mid x_t) \tag{2.14}$$

So now our posterior looks like:

$$p(x_t \mid z_{1:t}, u_{1:t}) = \eta p(z_t \mid x_t) p(x_t \mid z_{1:t-1}, u_{1:t}) \tag{2.15}$$

$p(x_t \mid z_{1:t-1}, u_{1:t})$ can be expanded using the rule of total probability:

$$
\begin{aligned}
&= p(x_t \mid z_{1:t-1}, u_{1:t}) \\
&= \int p(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}) p(x_{t-1} \mid z_{1:t-1}, u_{1:t}) dx_{t-1}
\end{aligned} \tag{2.16}
$$

Again we can simplify using the Markovian assumption.

$$
p(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}) = p(x_t \mid x_{t-1}, u_t) \tag{2.17}
$$

Hence

$$
= \int p(x_t \mid x_{t-1}, u_t) p(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}) dx_{t-1} \tag{2.18}
$$

remember that we defined:

$$
\overline{\text{bel}}(x_t) = \int p(x_t \mid x_{t-1}, u_t) p(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}) dx_{t-1} \tag{2.19}
$$

and

$$
\begin{aligned}
\text{bel}(x_l) &= \eta p(z_t \mid x_t) p(x_t \mid z_{1:t-1}, u_{1:t}) \\
\text{bel}(x_t) &= \eta p(z_t \mid x_t) \overline{bel}(x_t)
\end{aligned} \tag{2.20}
$$

combining this together, we get to the bayes filter algorithm:

1: **Algorithm Bayes_filter**($bel(x_{t-1}), u_t, z_t$):
2: for all x_t do
3: $\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1})\ bel(x_{t-1})\ dx$
4: $bel(x_t) = \eta\ p(z_t \mid x_t)\ \overline{bel}(x_t)$
5: endfor
6: return $bel(x_t)$

Fig 2.3: Bayes filter algorithm [3]

2.2.2 Kalman Filter

The Kalman filter[23] is an algorithm for data processing, which recursively calculates an estimation of a certain state (for example, where a robot is in a hallway). We can use the Kalman Filter, as long as our system behaviour is linear, and the noise from the sensor we are using can be described as a zero-mean Gaussian. The Kalman filter is used for it's computational efficiency and simplicity of implementation. It should be noted however, that it is only appropriate for linear problems. Therefore, it cannot be used to, for example, track a pendulum in it's non linear region of movement. The Kalman filter can be modified, however, to suit such non linear problems. Using the first order and second order Taylor series expansion, we can linearize the system (Extended and Unscented Kalman filters[11][12]). It should be also noted that the Kalman filter cannot be used, when dealing with applications where the error is not parameterised, and it can be quite sensitive to data outliers.

2.2.3 Particle filter

The particle filter[30][39] is the Kalman filter's non parametric equivalent. Also called Sequential Monte Carlo [14][15][31], it is one of the most popular methods for approximating probability densities, mainly because it doesn't make any assumptions regarding the shape of the said density, which is why it is so flexible.

Using the particle filter [16], we can handle non Gaussian noise and systems that behave non linearly[19]. The idea is to describe the posterior density function using particles. This density can be arbitrary. The approximate a state, we take the weighted sum of random samples. With each measurement coming from the sensor, we update our weights of each particle using the likelihood function. However, with time our filter may face what's called particle filter divergence. This happens when many our posterior approximation gets corrupted due to non contributing low weight particles. We resolve this issue in the resampling step, where we eliminate such particles [17].

While the particle filter certainly offers a lot of flexibility, it should be noted that it can be computationally expensive, as a large number of samples is usually needed to provide a good approximation. High dimensional state spaces lead to an exponential increase in the required number of particles. Therefore it is might not be the best option for low power situations.
The particle filter is a non parametric filter. That means that, unlike a classical Kalman filter, the posterior does not have to be a fixed function like a Gaussian. Here we assume that the posterior is not Gaussian, and instead try to approximate it with a bunch of particles that are distributed across specific parts of the state space. Now when we have this approximation of the posterior distribution, we can randomly pick samples from it. These randomly drawn samples describe our approximation of the state space. Also, because we are not limited to a specific posterior form, our non parametric filter can deal with multi modal situations so we can track multiple objects if we need to. The more parameters we have (in our case, particles) the better our approximation becomes. But the computational resources need also increases, a disadvantage of non parametric filters. To strike the optimal solution with provides both a good approximation while avoiding excess use of computational resources, we need to study the requirements of our application.

2.3 Particle Filter

2.3.1 Monte Carlo Method

Monte Carlo simulation is, in essence, the generation of random objects or processes by means of a computer.[1]. These random objects can be introduced artificially to solve problems originally deterministic in nature. A very simple example can be seen here[2]:

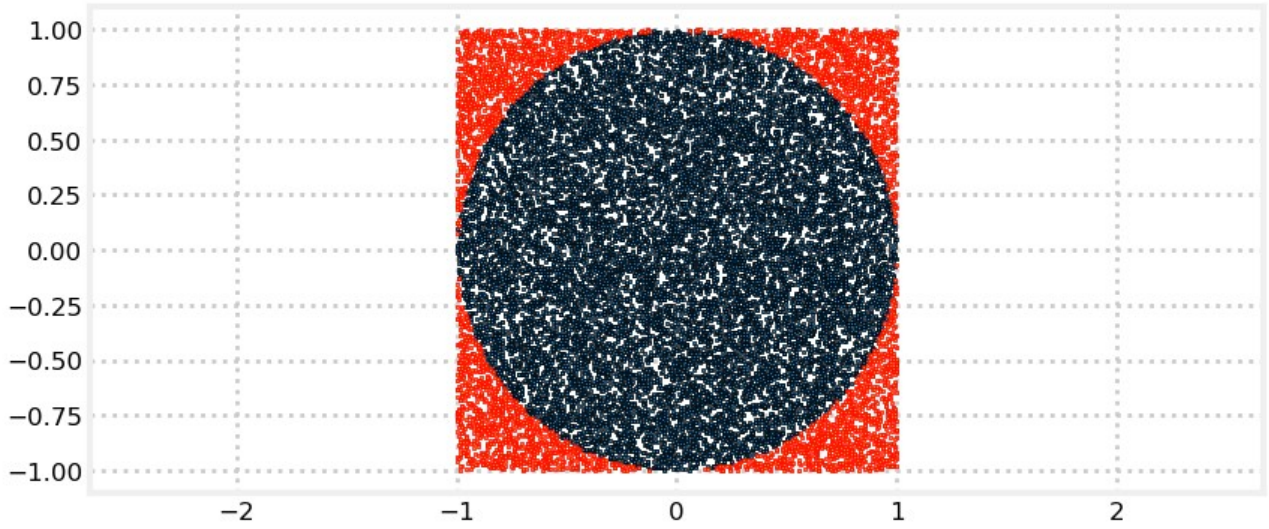


Fig 2.4: Monte Carlo Method for finding Pi [2]

Suppose that our goal is to calculate pi. We know that

$$\pi = A/r^2 \tag{2.21}$$

Now let's plot 2000 random points in the red square region. A point is inside a circle if its distance from the center of the circle is less than or equal to the radius. Some of these random particles will fall inside the circle, others will be outside. If we think of our particles as a unit of area, then we can say that in our example:

$$\pi = 4 \cdot (N_{in}/N_{tot}) \tag{2.22}$$

Where N_in would be the number of particles inside the circle, and N_tot is the total number of particles used. As we increase the number of particles used, we get a more accurate estimation of pi.

This same method can also be used to for numeric estimation of finite integrals[17].

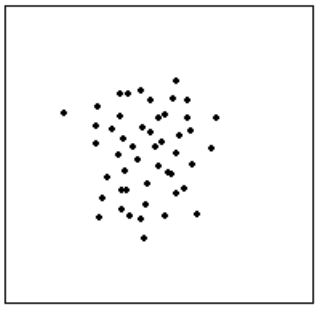
2.3.2 Particle Filter algorithm

Consider tracking a robot or a car in an urban environment. Let's imagine a robot moving in a hallway, and we need to track it. There are sensors on each door, and they can only tell if a robot is standing in-front of it or not (keep in mind, sensors are stochastic. Meaning, that a robot may stand there, and there is a small chance a sensor will still detect no robot). We start by creating several thousand *particles*. Each particle has a position that represents a possible belief of where the robot is in the scene. Suppose that we have no knowledge of the location of the robot. We would want to scatter the particles uniformly over the entire scene. If you think of all of the particles representing a probability distribution, locations where there are more particles represent a higher belief, and locations with fewer particles represents a lower belief. If there was a large clump of particles near a

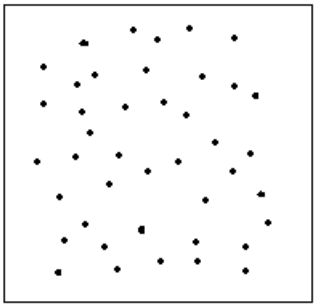
specific location that would imply that we were more certain that the robot is there. Each particle needs a weight , ideally the probability that it represents the true position of the robot. This probability is rarely computable, so we only require it be *proportional* to that probability, which is computable. At initialisation we have no reason to favour one particle over another, so we assign a weight of $1 / N$, for N particles. We use $1 / N$ so that the sum of all probabilities equals one[2].

There are 5 main steps in the operational loop of a particle filter.

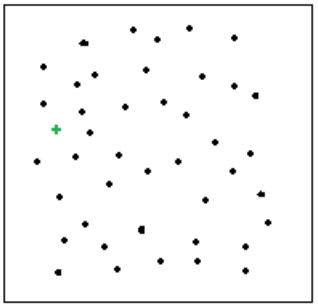
1- Randomly distribute a set of equally weighted particles across the state space



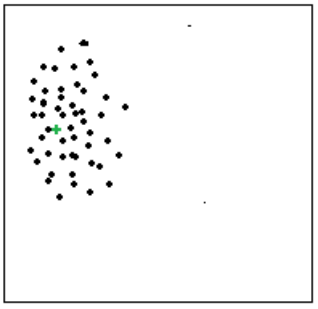
2- Predict the next iteration of values of the state space of each particle, based on a given mathematical model. For example, if a particle describes the motion of a robot on a single axis with 2 states, and , then , where is a noise model.



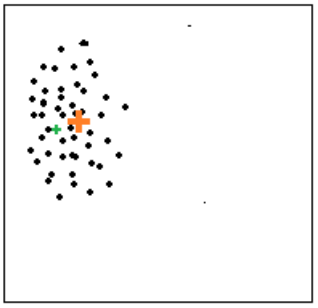
3- Get a measurement from your sensor. Using the your measurement model, judge each particle on how close it's measurement is to the true measurement. Particles that closely match the true measurement are given the heights priority (highest weight).



4-After a few updates you may notice that there are many non contributing, low weight particles. In the Re sample [8] step, we discard those particles and replace them with copies of higher probability particles.



5- Compute the estimate: based on the weight set, calculate the weighted mean state of the particles.



2.3.3 Deriving Particle Filter mathematically

Similar to the Bayes filter, the particle filter recursively uses the posterior from the previous step to compute the posterior of the next time step[33].
In order to describe the posterior, our filter uses particles. Each particle is a concrete hypothesis to what the state may be. The posterior is shown as:

$$\mathcal{X}_t := x_t^{[1]}, x_t^{[2]}, \ldots, x_t^{[M]} \tag{2.23}$$

Each one of the states, x_t, is a hypothesis by a particle of the true state at time t, of which we have a population of M.

1: **Algorithm Particle_filter**($\mathcal{X}_{t-1}, u_t, z_t$):
2: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
3: for $m = 1$ to M do
4: sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$
5: $w_t^{[m]} = p(z_t \mid x_t^{[m]})$
6: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: endfor
8: for $m = 1$ to M do
9: draw i with probability $\propto w_t^{[i]}$
10: add $x_t^{[i]}$ to $\mathcal{X}_t$
11: endfor
12: return $\mathcal{X}_t$

Fig 2.5: Particle filter algorithm [3]

Algorithm above explained:

1-Given the particle distribution from the previous time step, plus the control signal along with the measurement it caused,

2- We first empty our particle set

4- For each particle, we move it forward in time (predict step), using the control signal

5- Given the state that the particle describes, how likely is it to give us the current measurement our sensor is seeing? Assign that value as the corresponding weight

6-add the new weighted particle to our new set

Finally, in steps 8 to 11, we perform the resampling step, which we will discuss further in later sections.

Let's derive the algorithm above.
Let's think of our particles as a sequence of states. So for the particular particle m:

$$x_{0:t}^{[m]} = x_0^{[m]}, x_1^{[m]}, \ldots, x_t^{[m]} \tag{2.24}$$

we need to calculate the posterior distribution :

$$\text{bel}(x_{0:t}) \tag{2.25}$$

Let's call this distribution F. We can break it down as follows:

$$\begin{aligned} & p(x_{0:t} \mid z_{1:t}, u_{1:t}) = \\ & \text{Bayes } = \eta p(z_t \mid x_{0:t}, z_{1:t-1}, u_{1:t}) p(x_{0:t} \mid z_{1:t-1}, u_{1:t}) \\ & \text{Markov } \eta p(z_t \mid x_t) p(x_{0:t} \mid z_{1:t-1}, u_{1:t}) \\ & \qquad = \eta p(z_t \mid x_t) p(x_t \mid x_{0:t-1}, z_{1:t-1}, u_{1:t}) p(x_{0:t-1} \mid z_{1:t-1}, u_{1:t}) \\ & \text{Markov } \eta p(z_t \mid x_t) p(x_t \mid x_{t-1}, u_t) p(x_{0:t-1} \mid z_{1:t-1}, u_{1:t-1}) \end{aligned} \tag{2.26}$$

Let's now find our proposal distribution. In step 4 in the algorithm, the sample $x_t^{[m]}$ is generated from the following distribution. We will call this distribution G:

$$\begin{aligned} \text{bel}(x_t) = \; & p(x_t \mid x_{t-1}, u_t) \text{ bel } (x_{0:t-1}) \\ & = p(x_t \mid x_{t-1}, u_t) p(x_{0:t-1} \mid z_{0:t-1}, u_{0:t-1}) \end{aligned} \tag{2.27}$$

In statistics, importance sampling is a general technique for estimating properties of a particular distribution, while only having samples generated from a different distribution than the distribution of interest. Depending on the application, the term may refer to the process of sampling from this alternative distribution, the process of inference, or both. Using importance sampling, we can get the distribution F, from the particles representing the distribution G. The idea is this:

Given an original distribution, F:

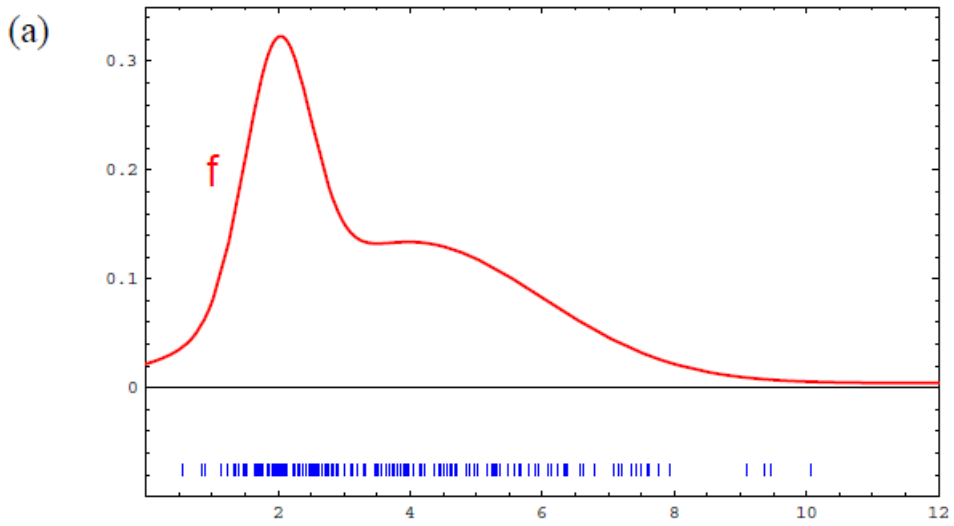


Fig 2.6 F distribution [3]

which we don't have a direct access to, plus another distribution, G:

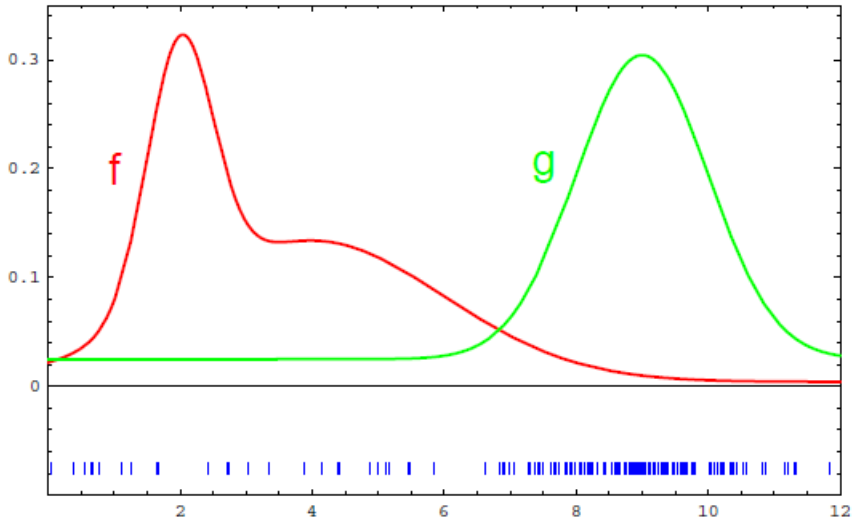


Fig 2.7 G distribution [3]

which is derived from a completely different distribution, we can “distort” how we view our particle density of G (by adding the appropriate weights), in order to get the density F

We are able to converge to the true density F, as long as $g(x) > 0$ whenever $f(x) > 0$

The distorting weights we will use will therefore be:

$$w_t^{[m]} = \frac{\text{target distribution}}{\text{proposal distribution}} \tag{2.28}$$

Now let us go back to our particle filter. We derived the target distribution, and the proposal distribution. Let us find the weights:

$$\begin{aligned} w_t^{[m]} &= \frac{\text{target distribution}}{\text{proposal distribution}} \\ &= \frac{\eta p(z_t \mid x_t) p(x_t \mid x_{t-1}, u_t) p(x_{0:t-1} \mid z_{1:t-1}, u_{1:t-1})}{p(x_t \mid x_{t-1}, u_t) p(x_{0:t-1} \mid z_{0:t-1}, u_{0:t-1})} \\ &= \eta p(z_t \mid x_t) \end{aligned} \tag{2.29}$$

Now let us put our weights to use, we arrive at our particle distribution for the next step:

$$\eta w_t^{[m]} p(x_t \mid x_{t-1}, u_t) p(x_{0:t-1} \mid z_{0:t-1}, u_{0:t-1}) = \text{bel}(x_{0:t}) \tag{2.30}$$

The idea behind the re sampling step is to avoid particle filter degeneracy. The idea is based on the Darwinian survival of the fittest. With each step, if the low weight particles are not eliminated, they will tend to distort the mean of our posterior. If we ignore this step, we end up with an inferior particle filter, which would calculate the weights illiterately like this:

$$w_t^{[m]} = p\left(z_t \mid x_t^{[m]}\right) w_{t-1}^{[m]} \tag{2.31}$$

the result is that a rising number of the particle stay in low probability regions. The non contributing resources end up taking more of our computational resources than necessary.

2.3.4 Particle filter degeneracy

The particle algorithm suffers from what’s called the *degeneracy problem*. It starts with uniformly distributed particles with equal weights. There may only be a few particles that actually represent the true state. As the algorithm runs any particle that does not match the measurements will acquire a very low weight. Only the particles which are able to provide a measurement close to the sensor data will have an appreciable weight. We could have 5,000 particles with only 3 contributing meaningfully to the state estimate! When this happens we say the filter has *degenerated*. This problem is usually solved by some form of *resampling* of the particles. Particles with very small weights do not meaningfully describe the probability distribution of the hand movement. The resampling algorithm discards particles with very low probability and replaces them with new particles with higher probability. It does that by duplicating particles with relatively high probability. The duplicates are slightly dispersed by the noise added in the predict step. This results in a set of points in which a large majority of the particles accurately represent the probability distribution.

There are many resampling algorithms. Let’s have a look.

2.3.5 Resampling methods

Which resampling algorithm we use matters a lot when it comes to performance. Suppose we simply resample by picking random particles, ignoring the weight value. The resulting distribution would be mostly corrupted, since we picked a lot of low weight particles.

Research on the topic continues, but a handful of algorithms work well in practice across a wide variety of situations. We need our algorithm to strike the optimal spot, between picking high probability particles to represent the true density, and also including a few low probability particles to allow us to still deal with non linear behaviour.

First we normalize the weights before proceeding with the actual resampling:

```
function obj= resample(obj)

        %normalize weights first
        %summation of weights
        sumWeights=sum(obj.weights);

        if(sumWeights~=0)
           obj.weights= obj.weights/sumWeights;
        end
```

2.3.6 Halving Resampling method

The idea is straightforward: we sort the particles according to weight, starting from 1 all the way to 0, remove the lower half and replace it with a duplicate of the upper half.

```
        %halving resample
        if(obj.resampleAlgorithm==0)

           %
           %temporarily stitch the weight array as an additional
           %column

           tempParticles = obj.particles;
           tempParticles(:,end+1)=obj.weights;
           sortAccordingTo=obj.n+1;%sort according to which row
           tempParticles=sortrows(tempParticles,sortAccordingTo,'descend');

           half = obj.M/2;

           topHalf=tempParticles(1:half,:);
           resampled = zeros(obj.M,obj.n+1);
           resampled(1:half,:)=topHalf;
           resampled(half+1:end,:)=topHalf;
           obj.particles=resampled(:,1:end-1);
           obj.weights=resampled(:,end);
        end
```

2.3.6 Multinomial Resampling

The idea is this: Compute the cumulative sum of the normalised weights. This gives you an array of increasing values from 0 to 1. Here is a plot which illustrates how this spaces out the weights. The colours are meaningless, they just make the divisions easier to see:

Fig 2.9 Multinomial Resampling [2]

To select a weight we generate a random number uniformly selected between 0 and 1 and use binary search to find its position inside the cumulative sum array. Large weights occupy more space than low weights, so they have a higher chance of being selected

```
%multinomial

if(obj.resampleAlgorithm==1)
   cumulative_sum = cumsum(obj.weights);
   cumulative_sum(end)=1.0;

   resampled=zeros(1,obj.n+1);
   randWeight=rand(1,1);
   [value,index]=obj.closest_value(cumulative_sum, randWeight);

   tempParticles = obj.particles;
   tempParticles(:,end+1)=obj.weights;

   resampled(1,:)=tempParticles(index,:);

   for i=1:(obj.M-1)
   randWeight=rand(1,1);
   [value,index]=obj.closest_value(cumulative_sum, randWeight);
   resampled(end+1,:)=tempParticles(index,:);
   end
   obj.particles=resampled(:,1:end-1);
   obj.weights=resampled(:,end);
end
```

2.3.7 Residual Resampling

Residual resampling both improves the run time of Multinomial resampling, and ensures that the sampling is uniform across the population of particles. The normalised weights are multiplied by the number of particles, and then the integer value of each weight is used to define how many samples of that particle will be taken. For example, if the weight of a particle is 0.0012 and we have N=3000 particles, the scaled weight is 3.6, so 3 samples will be taken of that particle. This ensures that all higher weight particles are chosen at least once.

However, this does not generate all *N* selections. To select the rest, we take the *residual*: the weights minus the integer part, which leaves the fractional part of the number. We then use a simpler sampling scheme such as Multinomial, to select the rest of the particles based on the residual. In the example above the scaled weight was 3.6, so the residual will be 0.6 (3.6 - int(3.6)). This residual is very large so the particle will be likely to be sampled again. This is reasonable because the larger the residual the larger the error in the round off, and thus the particle was relatively under sampled in the integer step.

2.3.8 Stratified Resampling

The last algorithm we will look at is stratified resampling. The space is divided into divisions. We then choose a random offset to use for all of the divisions, ensuring that each sample is exactly apart. It looks like this.:

Fig 2.10 Stratified Resampling [2]

```
%stratified
if(obj.resampleAlgorithm==2)
   cumulative_sum = cumsum(obj.weights);
   cumulative_sum(end)=1.0;

   randWeight=0;
   jump = 1.0/obj.M;

   resampled=zeros(1,obj.n+1);
   randWeight=rand(1,1);
   [value,index]=obj.closest_value(cumulative_sum, randWeight);

   tempParticles = obj.particles;
   tempParticles(:,end+1)=obj.weights;

   resampled(1,:)=tempParticles(index,:);
   randWeight=0;
   jump = 1.0/obj.M;
   for i=1:(obj.M-1)
      [value,index]=obj.closest_value(cumulative_sum, randWeight);
      resampled(end+1,:)=tempParticles(index,:);
      randWeight=randWeight+jump;
   end
   obj.particles=resampled(:,1:end-1);
   obj.weights=resampled(:,end);
end
```

2.3.9 Mean Particle Retrieval

First we ensure the weights are normalized

```
function [meanParticle,obj]= getMeanParticle(obj,withHistory)

    %ensure weights are normalized
    summation=sum(obj.weights);

    if(summation~=0)
       obj.weights= obj.weights/summation;
    end
```

For each state, we first retrieve a weighted average from all particles:

$$S_t = S_1 \cdot w_1 + S_2 \cdot w_2 + S_3 \cdot w_3 + \cdots \cdot S_M \cdot w_M \tag{2.32}$$

```
    sumOfStates=zeros(1,obj.n);
    for i=1:obj.n
       sumOfStates(i)=sum(obj.particles(:, i).*obj.weights);
    end
     meanParticle=sumOfStates;
```

S is a certain state, for example the amplitude, and the w is the particle weight.

Additionally we can also consider the previous mean states (this step can also be disabled, and the number of history samples can be edited in the code), to reduce the effect strong outliers, which are usually caused by radar noise. Repeated outliers however indicate a true change, so such changes are correctly captured when retrieving the mean.

$$\begin{aligned} &S_t = S_t \cdot W_{s_1} + S_{t-1} \cdot W_{s2} + S_{t-2} \cdot W_{s3} \cdots S_{t-10} \cdot W_{s10} \\ &W_{s1} = 1 \\ &W_{s2} = 2 \\ &W_{s3} = 3 \\ &W_{s4} = 4 \end{aligned} \tag{2.33}$$

```
        obj.meanParticleHistory(end+1,:)=meanParticle;

        %we don't need to keep more than 6 predictions, so discard the old
        %predictions
        if(size(obj.meanParticleHistory,1)>obj.meanHistoryLength)
            obj.meanParticleHistory=obj.meanParticleHistory(2:end,:);

        End

        %Here we form form the weights used for smoothing the particle mean with history

        N_hist=size(obj.meanParticleHistory,1);
        TotalWeight=1.0;
        Weights=[1];
        i=1;
        while (i<N_hist)
                Weights(end+1)=(1);
           i=i+1;
        end

        sumWeights=sum(Weights);

        if(sumWeights~=0)
           Weights= Weights/sumWeights;
        end
        Weights =transpose(Weights);
       %We won‘t apply the mean history averaging on the phi state, as this can introduce
        unwanted errors
        phi=meanParticle(2)

        if(withHistory)
           for i=1:(obj.n)
              meanParticle(i) = sum(obj.meanParticleHistory(:,i).*Weights);
           End
        meanParticle(2)=phi
        end
end
```

2.4 Player
2.4.1 Audio Stretching Basic Concept

One problem that we could have faced in our project is with scaling the audio. Since we will be playing a musical piece, and we want constantly perform adjustments on the playing speed, we could face a problem of unwanted varying pitch, if we simply used sample rate conversion.
The idea behind sample rate conversion, is to take a sound wave form, capture it's samples, then rebuild a new wave form but with a different rate. This ends up changing the speed, but also the pitch. This maybe fine for a drum track with no melody, but not for our use case.

The main theory behind time stretching, is to first split our wave form into equal segments

Fig 2.12 Original Audio

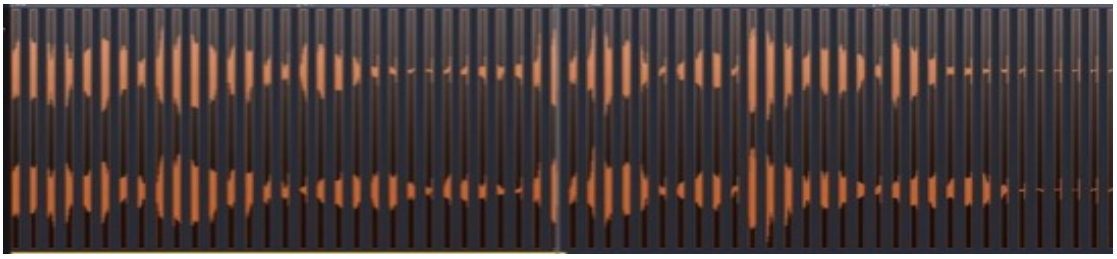


Fig 2.13 Stretched audio with spaces

By adding spaces between the segments, we essentially can stretch our audio without changing pitch. One issue though, is the spaces we introduced, which will cause our audio to sound choppy.

To solve this, we copy all the segments and place them in the next space section to fill in the gaps.

Fig 2.14 After duplicating audio segments

This will sound better, but will still have artefacts. This is mainly because we are currently splicing audio segments at non zero points, as seen here if we zoom in.

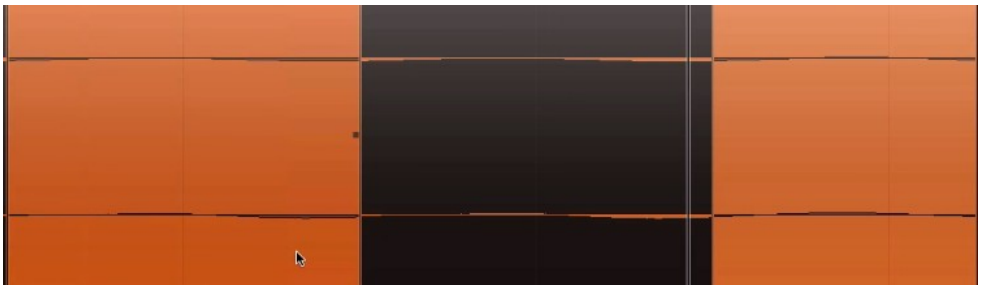


Fig 2.15 Zoom in between audio segments

this is then fixed by cross fading, which is applied between all the segments, resulting in smooth audio.

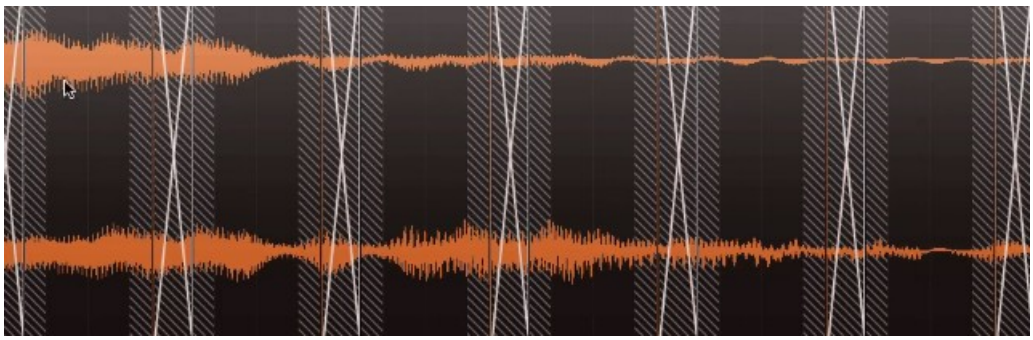


Fig 2.16 After performing cross fading

2.4.2 Phase Vocoder

Let's build upon what we learned to know how a better time stretching algorithm is implemented: the Phase Vocoder. This is the way time stretching is implemented in Matlab[55][56].
As before, the input audio is broken into short segments of sound which are reassembled to create an output sound. In the phase vocoder, the output signal is always constructed from equally spaced segments. To achieve time stretching, the spacing of the segments extracted from the input is variable. If segments are taken with lower amounts of overlap, the resulting output is longer so the input is apparently stretched. If segments have more overlap than in the input sound, the output will sound like it's played faster.

The distinctive aspect of the phase vocoder is that when segments are assembled into the output, the phase of each frequency component is adjusted to avoid phase cancellation. The algorithm finds these frequencies using Fast Fourier Transform, which composes the sound into the individual frequency components. Normally, if the spacing of segments is changed from input to output, different frequencies will be shifted by different amounts of phase. Some will add constructively and some will add destructively. Thus, some frequencies will be emphasised over others, often creating a buzz. The key move here is to adjust the phases of all the components to avoid this issue.

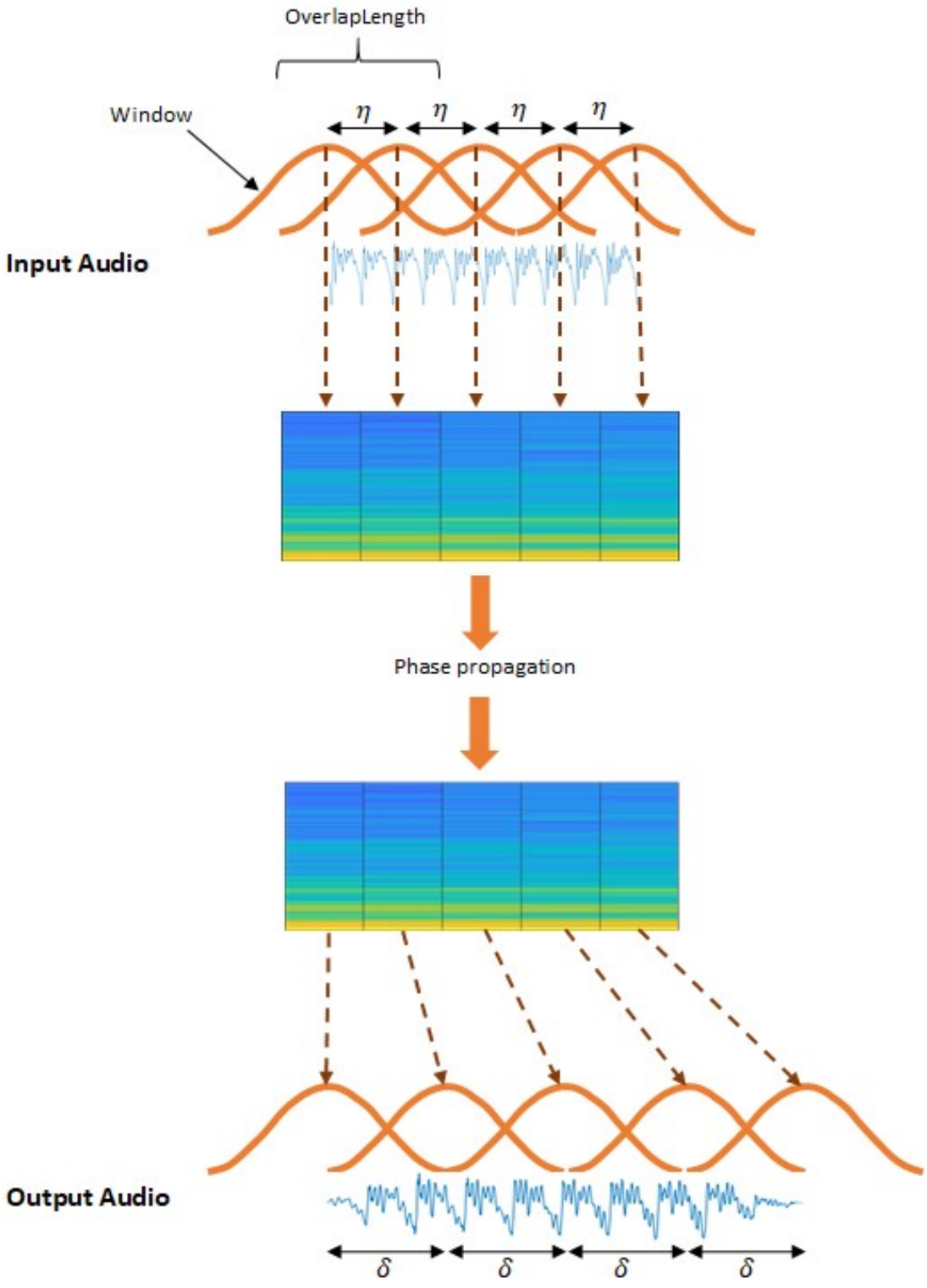


fig 2.17: Phase Vocoder Ilustration [57]

2.4 Radar

2.4.1 CFAR algorithm

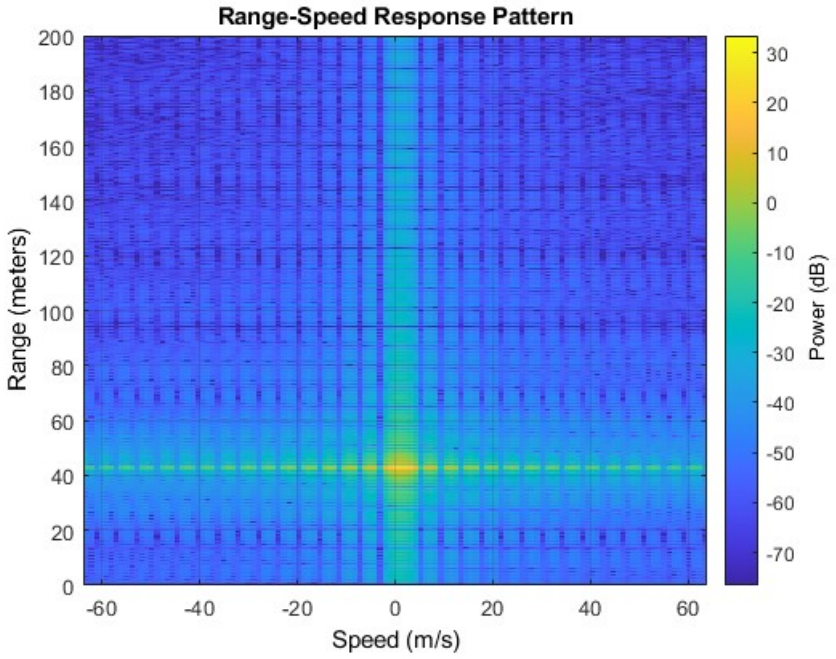


Fig 2.18: Typical Range Doppler Map

The radar uses what's called a Constant False Alarm Rate in order to detect the targets. Let's start by considering the False alarm algorithm. First it looks at the incoming echoes received by the antenna. There will always be a background noise level. The key is to consider using a threshold for signal power. This threshold is set to be more than the background noise. If a signal has more power than the threshold, then we know we have a real target. This idea is not perfect however, as some false alarms are still triggered. In most radar detectors, the threshold is set in order to achieve a required probability of false alarm

Constant False Alarm Rate builds on this concept, but instead uses a variable threshold, depending on the background noise. This way it tries to keep the false alarm rate constant.

If there are multiple targets in radar view, our algorithm picks the strongest signal. The velocity and range of the corresponding target are sent to the Matlab script for further processing.

3 METHODOLOGY

3.1 Motion Model

3.1.1 First motion model

We first start by thinking about a motion model. The motion model basically describes what our range and velocity (of the hand) will be, given certain values for our states. We first start by assuming a simple sinusoidal motion. Our parameters will be volume, range offset and phase offset, and frequency:

$$y = y_0 + y_a \cos(2\pi(\phi_0 + ft)) \tag{3.1}$$

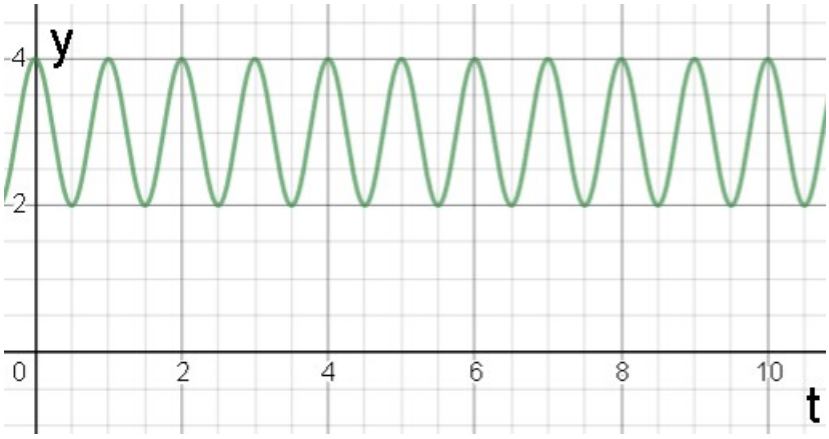


Fig 3.1: Model 1

3.1.2 Motion Model with harmonics

Humans don't wave their hands in perfect sinusoidal motion, though. We can describe this extra complexity, by adding a second harmonic to our model.

$$y = y_0 + y_a(a_1 \cos(2\pi\phi_r) + a_2 \cos(4\pi\phi_r)) \tag{3.2}$$

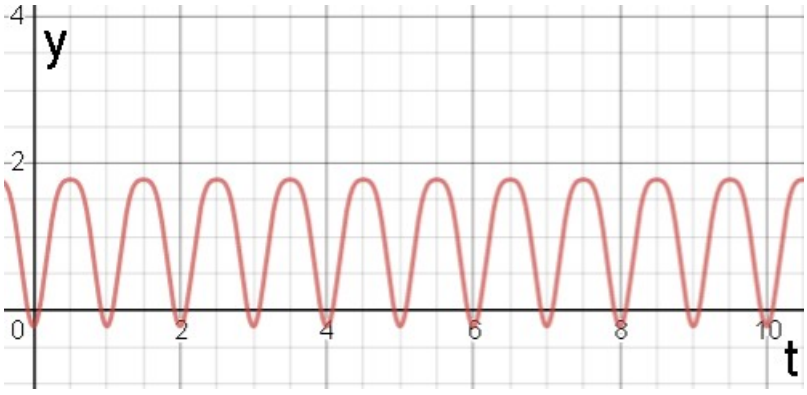


Fig 3.2: Model with added Harmonic

Where:

$$\begin{aligned} \phi_r &= f(t + t_0) \\ a_2 &= a_r \cdot a_0 \\ a_1 &= 1 \end{aligned} \tag{3.3}$$

The derivative of our first equation, the velocity would be:

$$v = -2\pi y_a f((a_1 \sin(2\pi\phi_r)) + (2a_2 \sin(4\pi\phi_r))) \tag{3.4}$$

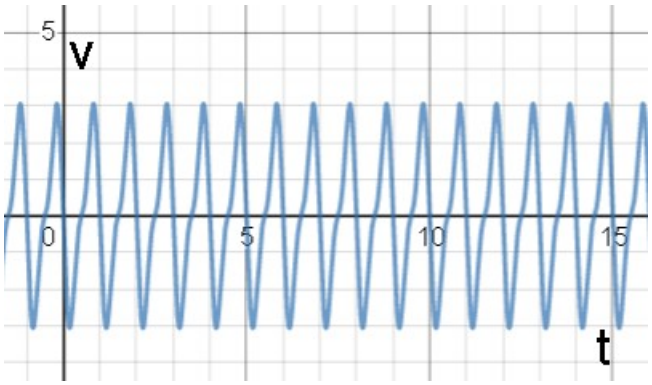


Fig 3.3: Velocity graph

This let's us not only model the amplitude, frequency, range and phase offset, but also the shape of the motion so that our particle filter can better predict and follow the hand.

We can now model any hand movement, by adjusting the 5 states:

y_0 : Hand offset (position of centre of oscillation with respect to radar sensor)
y_{a0}: General Amplitude of hand movement
ϕ_r: Temporal time
f: Hand waving frequency
a_r: The ratio of a2 to a1, which determines the shape of the movement

3.1.3 State prediction

For the prediction step in the particle filter we used the following equations:

The amplitude is assumed to stay the same. However we do add some noise to allow our particle filter to follow the amplitude state if it changes.

$$y_{a0_{t+1}} = y_{a0_t} + w \cdot \sqrt{Q_{y_a}} \tag{3.5}$$

We also assume that the offset remains the same:

$$y_{0_{t+1}} = y_{0_t} + w \cdot \sqrt{Q_{y_0}} \tag{3.6}$$

As well as the frequency:

$$f_{t+1} = f_t + w \cdot \sqrt{Q_f} \tag{3.7}$$

The phase will move at the rate of the frequency:

$$\phi_{t+1} = \phi_t + f_t \cdot t_s + w \cdot \sqrt{Q_\phi} \tag{3.8}$$

where

$$Q_\phi = Q_f . t_s$$

where w is white noise (a random variable between -1 and 1, with a mean of 0)and Q is the process noise matrix.

The standard deviations of the process noise is set to:

$$Q = \begin{bmatrix} Q_{ya} & 0 & 0 & 0 \\ 0 & 0 & Q_\phi & 0 \\ 0 & 0 & Q_f & 0 \\ 0 & 0 & 0 & Q_{y_0} \end{bmatrix} \tag{3.9}$$

Np is the number of samples per period (of hand motion)

$$N_p = \frac{1}{t_s} \cdot \frac{1}{f_0} \tag{3.10}$$

$$t_s = 0.016s \tag{3.11}$$

where ts is the sample rate of our radar
f_0 is the mean song frequency we expect the hand motion to be around (depends on the chosen song).
Ex: For our first song, 135 BPM, so we will set this to 2.25

To configure our Q matrix, we start out with how much standard deviation we can expect per period. For example, a change of 8mm in amplitude per period sounds like a good place to start. We tuned our Q matrix further with experimentation. The Q :

$$Q_{a0} = \left(\frac{0.0065}{\sqrt{N_p}}\right)^2 \tag{3.12}$$

$$Q_{y0} = \left(\frac{0.0189}{\sqrt{N_p}}\right)^2 \tag{3.13}$$

$$Q_f = \left(\frac{0.041}{\sqrt{N_p}}\right)^2 \tag{3.14}$$

$$Q_\phi = Q_f . t_s \tag{3.15}$$

```
function obj = predict(obj)


    Qu=obj.Q*obj.Qsigma;
    %obj.Qsigma is a scaling factor that scales the entire Q matrix (only used when tuning Q, to help the programmer find the best Q values faster. Once the Q matrix is found, this scaling factor is set back to 1 ).


    %Ya0
    obj.particles(:, 1) =obj.particles(:, 1) +(randn(obj.M,1).*sqrt(Qu(1,1)));

    %Phi r
    obj.particles(:, 2) =obj.particles(:, 2) + obj.particles(:,3).*obj.filterSampleRate
   +rand(obj.M,1).*sqrt(Qu(3,3))*obj.filterSampleRate;

    %F
    obj.particles(:, 3) =obj.particles(:, 3) +(randn(obj.M,1).*sqrt(Qu(3,3)));

    %Y0
    obj.particles(:, 4) =obj.particles(:, 4) +(randn(obj.M,1).*sqrt(Qu(4,4)));

end
```

3.1.4 Measurement

Each particle is, essentially, a list of states trying to describe the hand motion at the current time. These states are not directly usable, as we need to pass them through our measurement model mentioned above.

```
function h=measurment(obj,particle)

        h=zeros(2,1);

        a1=1;
        a2=a1*0;

        h(1) = particle(4)+particle(1)*((a1*cos(1*2*pi*particle(2)))+(a2*cos(2*2*pi*particle(2)))) ;

        h(2) = -particle(1)*2*pi*particle(3)*((a1*1*sin(1*2*pi*particle(2)))+
                (a2*2*sin(2*2*pi*particle(2))));

end
```

3.1.5 Update function

In the update step we will judge our particles based on both the velocity and range. We can change the parameters in the update step to flavor the range over the velocity or the other way around. For example one can choose to set equal weight to the velocity and range reading, if both are equally noisy.

$$w_y = 0.5, w_v = 0.5 \tag{3.16}$$

As a good starting point, one can use the range resolution and velocity sampling rate from the radar parameters sheet to determine the weight mix.

In our radar, the sampling periods for the range and velocity are:

$$\begin{aligned} Ys &= 0.0344s \\ Vs &= 0.0886s \end{aligned} \tag{3.17}$$

each particle is then passed through the measurement function:
we also implemented a simple alpha beta filter in this step, which allows us to set a level of trust we have in the radar (can be edited).

Matlab's vectorized form is utilised in the next step to increase performance, where we will calculate the range error for each particle. The same is also repeated for the velocity. The weights are set as the error, and as a final step the weight vector is normalised.

```
function obj = basicUpdate(obj,measurmentMean)

        wy=0.7;
        wv= 0.3;

        %pass all the particles through the measurment function

        h_array=zeros(obj.measurmentDimensions,1);
        h_array(:,1)=obj.measurment(obj.particles(1,:));
        for i =1:obj.M
            h_array(:,i)=obj.measurment(obj.particles(i,:));
        end
        obj.errorPf=[];

        %implementing a simple alpha beta filter

        trustRatio=95/100;%our level of trust in the radar

        %calculate the difference between pf measurment and real reading
        %vectorized form for faster computation

        TrueZ=measurmentMean.*ones(obj.M,1);
        ParticleZ=h_array.';

        ey=TrueZ(:,1)-ParticleZ(:,1);
        ey=ey*trustRatio;
        ey=ey.*ey;
        ev=TrueZ(:,2)-ParticleZ(:,2);
        ev=ev*trustRatio;
        ev=ev.*ev;
        obj.errorPf=wy*ey + wv*ev;

         %now we have errorPf, calculate the new weight

          obj.weights=obj.errorPf;

           %Max wieght:

           maximum = max(obj.weights);

           %flip the score

           obj.weights=maximum-obj.weights;

           %summation of weights

           sumWeights=sum(obj.weights);
           if(sumWeights~=0)
               obj.weights= obj.weights/sumWeights;
           end

end
```

3.2 Particle Filter setup

3.2.1 Modelling radar noise

The biggest noise source for our radar is the thermal noise, caused by the radar's own circuitry [4]. The thermal noise can be very well modelled with Gaussian distribution of the form

$$pdf = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{3.19}$$

3.2.2 Chosen Parameters

In this project we developed a robust particle filter platform. The main class, Particle Filter, implements functionalities common to all applications (such as the resampling). A subclass was developed for our particular setup. The following parameters were used:

Number of particles	1000	
Chosen Resampling Algorithm	Multinomial	
Radar Sample Rate	62.5 samples/sec	
State	Initial guess	Standard Deviation
Amplitude	15mm	0.1
Offset	25mm	0.2
Phi	0	0.5
Frequency (tempo in Herz)	2.25 (or 135 BPM)	0.1
Amplitude ratio	0	0.1
*Consider History?	Yes	Up to 10 past state spaces

* Consider History: This is an added functionality, that helps to further smooth the frequency output, by averaging the latest state with the past 10 versions. This helps with reducing the effect of strong outliers.

3.3 Program Setup

3.3.1 Python and Matlab Program

The program is based on two parts: one written in python and another written in Matlab. The python program deals with the User Interface and with time stretching, while the Matlab program communicates directly with the radar and handles the particle filter algorithm.

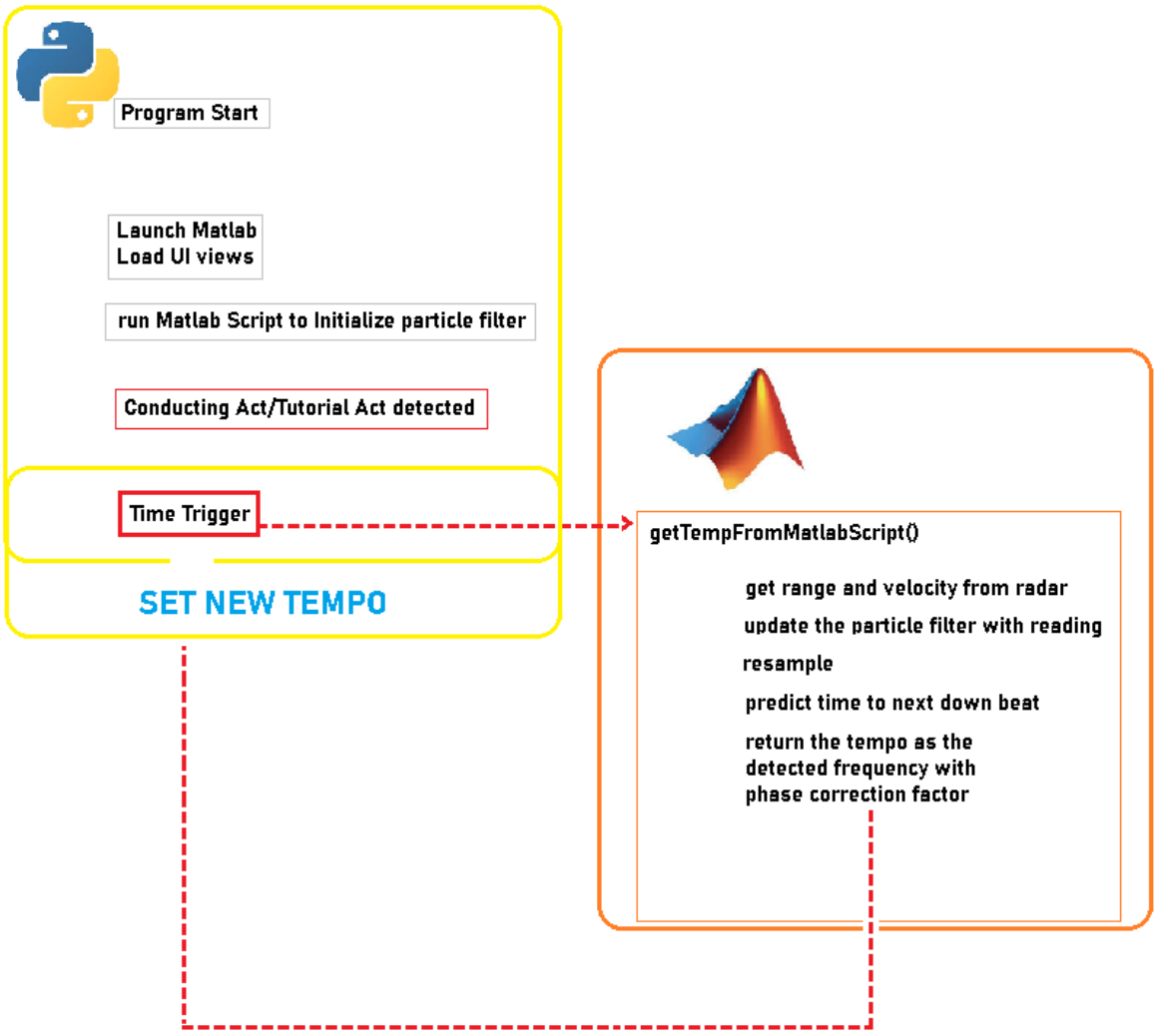


Fig 3.4 Porgram diagram

3.3.2 Beat Prediction algorithm

In the Matlab program, the locations of all the beats are marked in samples. With each speed update request coming from the python player, the current location in the music is given to the Matlab script. Using this location we can find the location of the next beat, and the previous beat (note that in the musical pieces used, the beat to beat time is not constant).

After finding the mean particle, the time left till the next down beat is found as:

$$timeLeft = (ceil(\phi - 0.5) - \phi + 0.5)/f \tag{3.20}$$

using the current location in the music, we can find how much time is left till the next beat (in the music).

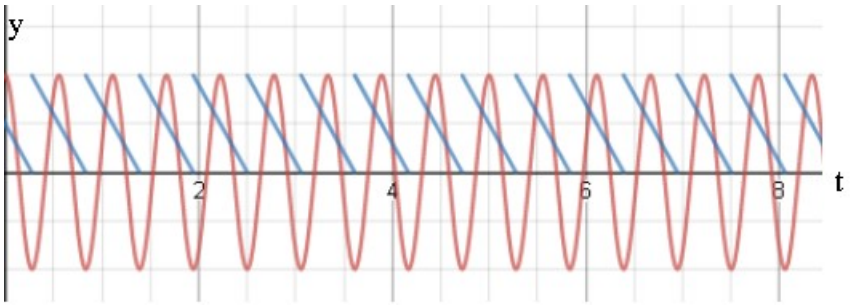


Fig 3.5: Time left till next down beat (in blue)

The new tempo signal, which captures:
a- the detected frequency
b- the adjustment needed to realign the phase

is found as follows:

$$tempo = f * 60 * phaseCorrectionFactor \tag{3.21}$$

where the phase correction is how much musical time we have left, divided by the time we would have in ideal situations (ie, when the phase is correct). A large phase error results in a larger correction and vice versa.

$$phaseCorrectionFactor = temporalDistanceLeftToNextBeat / fullBeatToBeatTemporalDistance \tag{3.22}$$

4 EXPERIMENTAL WORK AND RESULTS

4.1 Experimental Setup

Similar to the Haus Der Musik exhibition in Vienna, our setup is also meant to be displayed in an upcoming exhibition. A large screen with sound system, coupled with a computer and radar sensor were used.

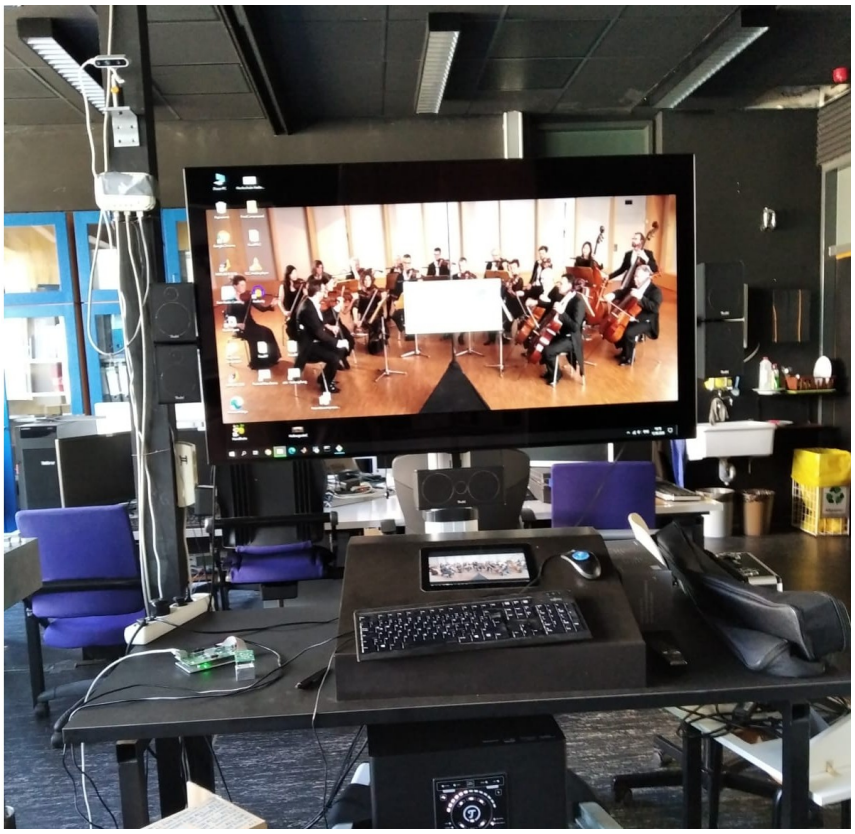


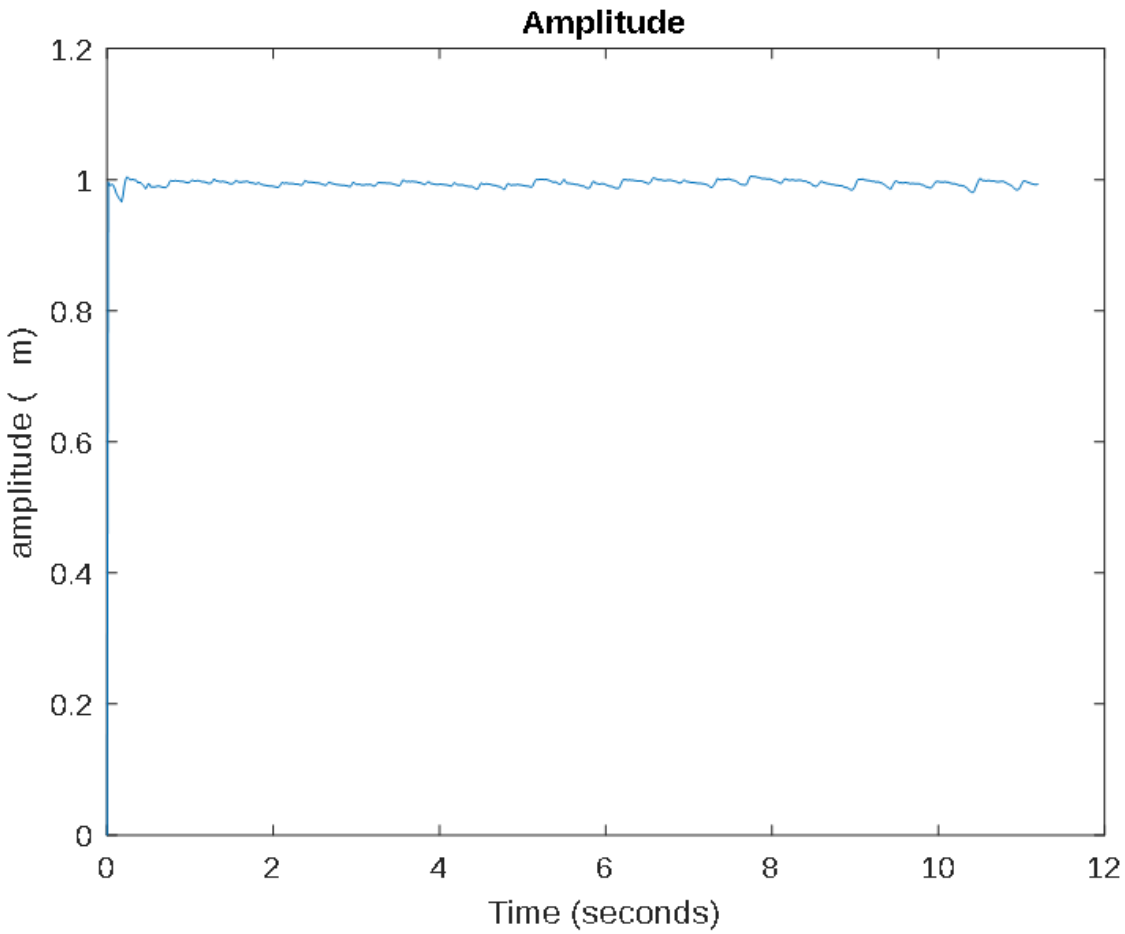
Fig 4.1: setup

4.2 Experimentation with artificial data

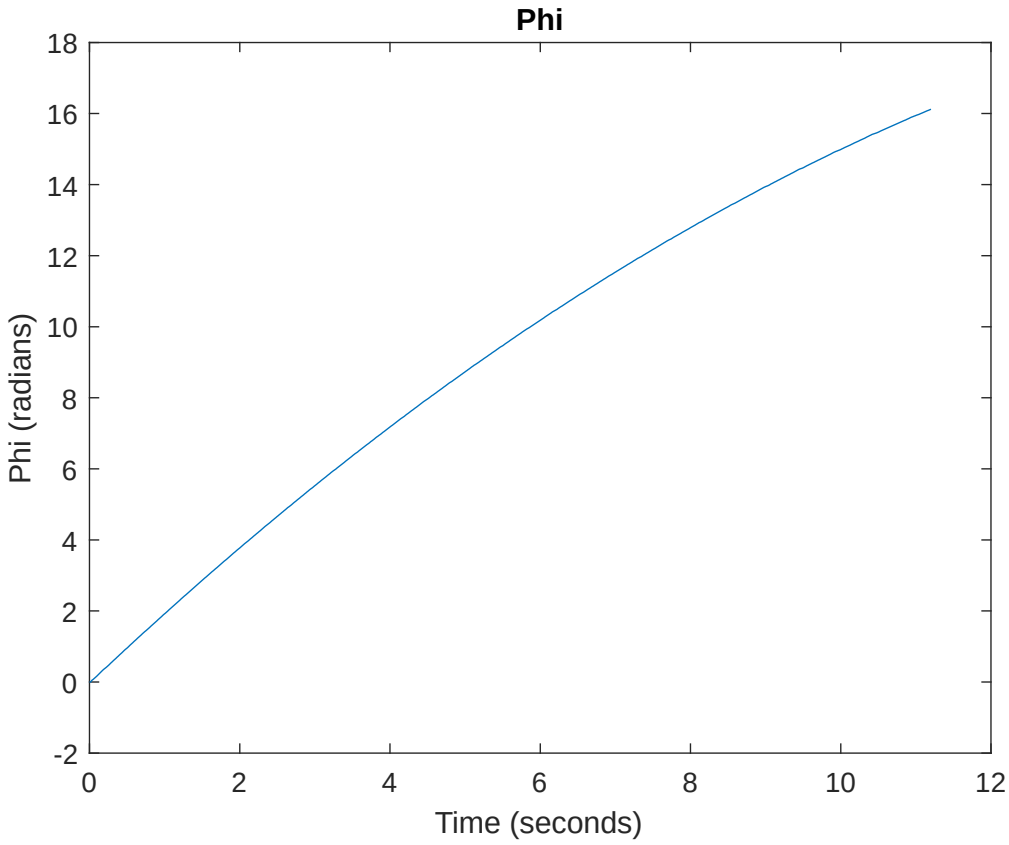
Before proceeding with experimentation on the real radar, we decided to first test our particle filter with artificially generated data. We used the class FakeRadar, which can be used to test several conditions, such as increasing/decreasing Frequency/Amplitude/Offset, with our without noise. The Class also gives us the ability to define our noise profile, such as the type (Gaussian, Logarithmic, etc), as well as the variance of said noise.

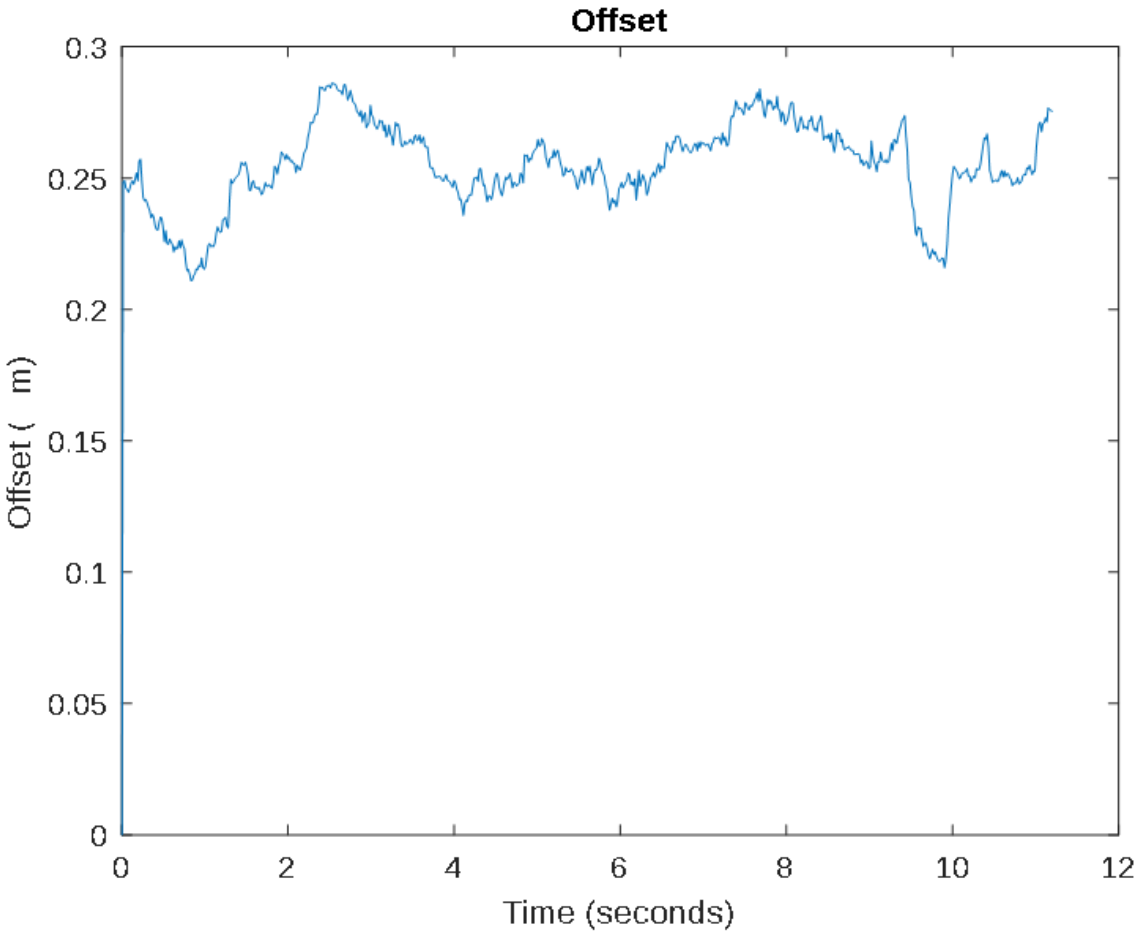
4.2.1 Experiments with noiseless data

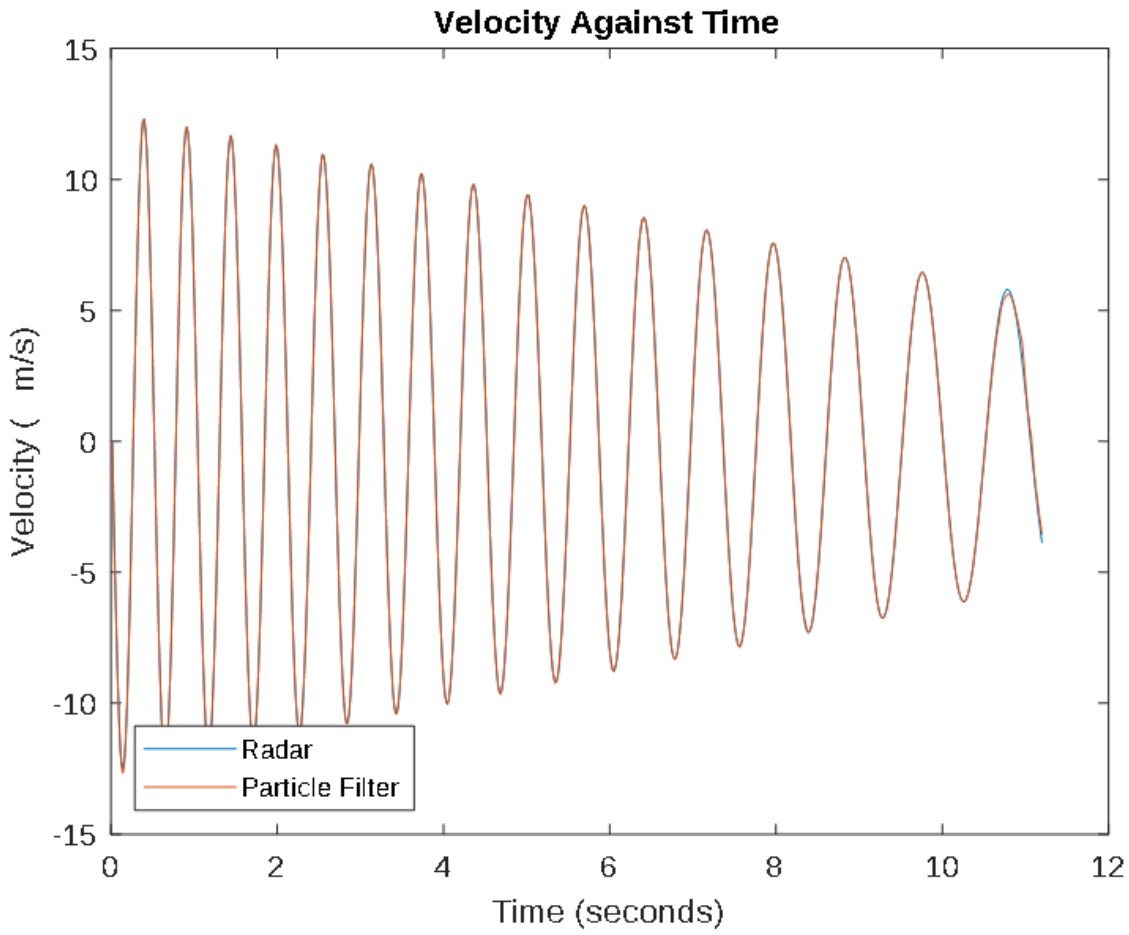
The following test was performed with a constantly decreasing frequency

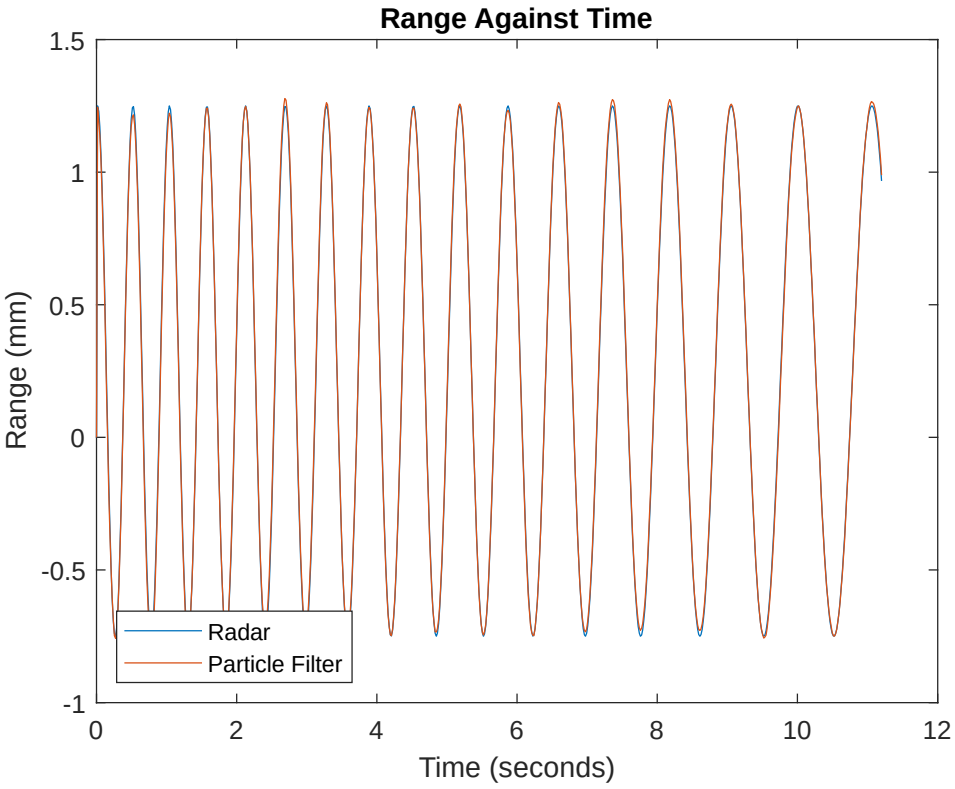


predictably the particle filter filter performed very well:


Phi
Phi (radians)
Time (seconds)
18
16
14
12
10
8
6
4
2
0
-2
0
2
4
6
8
10
12

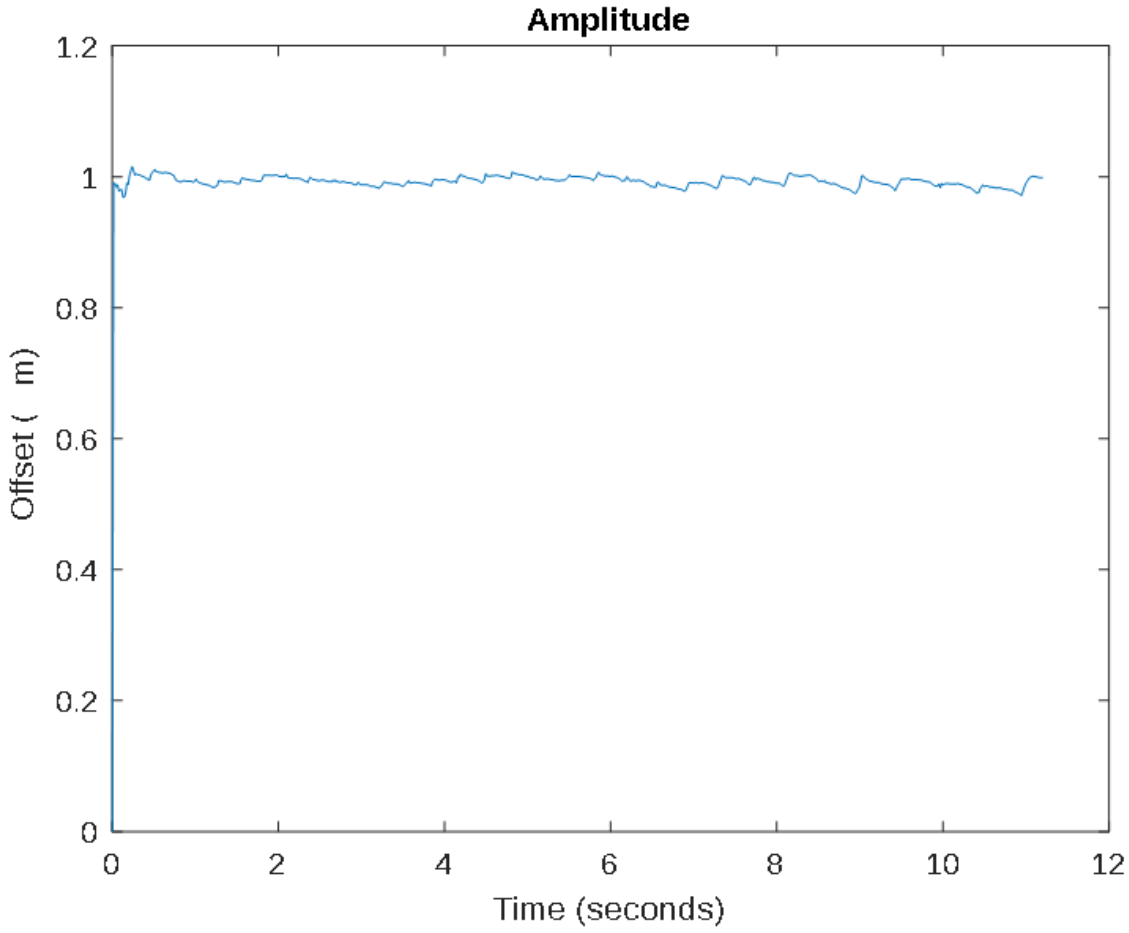

Offset
0.3
0.25
0.2
0.15
0.1
0.05
0
Offset (m)
0
2
4
6
8
10
12
Time (seconds)

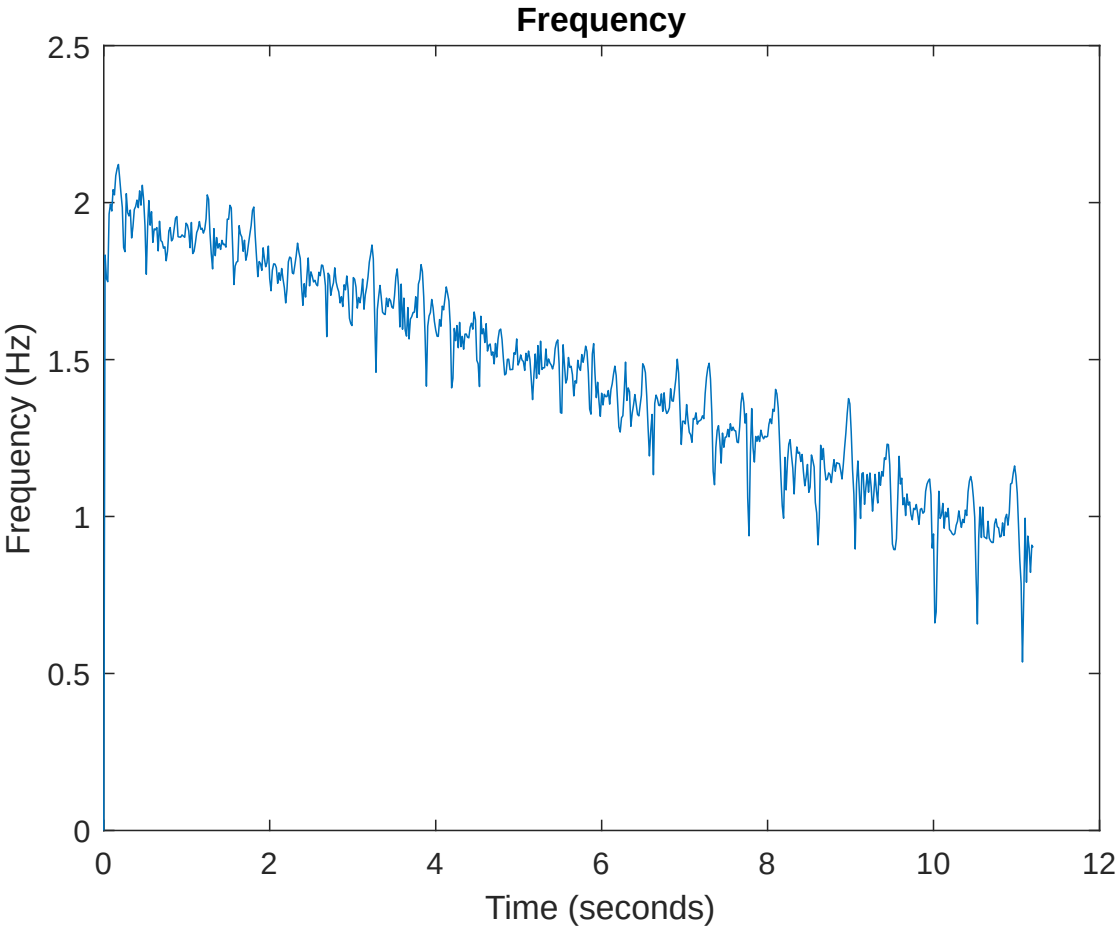

Velocity Against Time
15
10
5
0
-5
-10
-15
Velocity (m/s)
0
2
4
6
8
10
12
Time (seconds)
Radar
Particle Filter

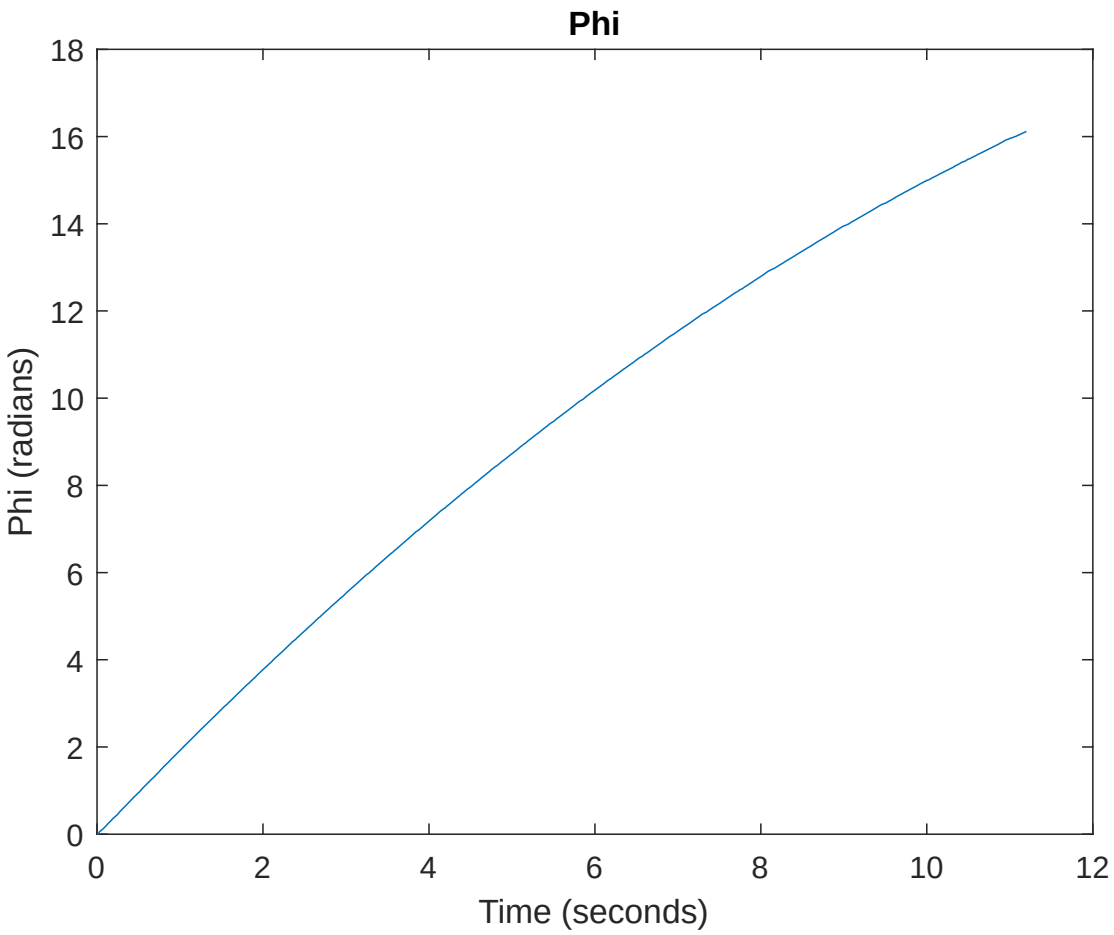

Range Against Time
1.5
1
0.5
0
-0.5
-1
Range (mm)
0
2
4
6
8
10
12
Time (seconds)
Radar
Particle Filter

4.2.1 Experiments with noisy data

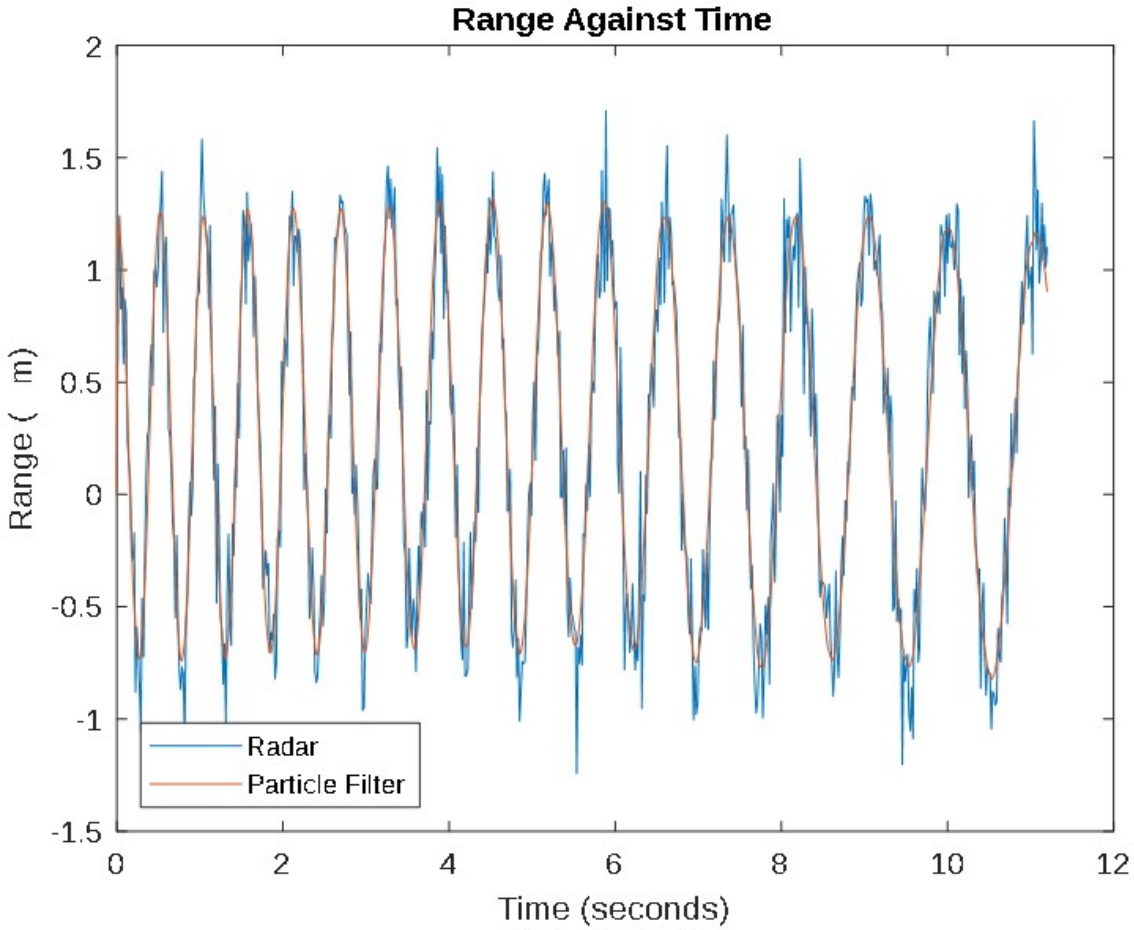
The same experiment was performed with an added Gaussian noise (variance of 0.2), with fairly good prediction results:

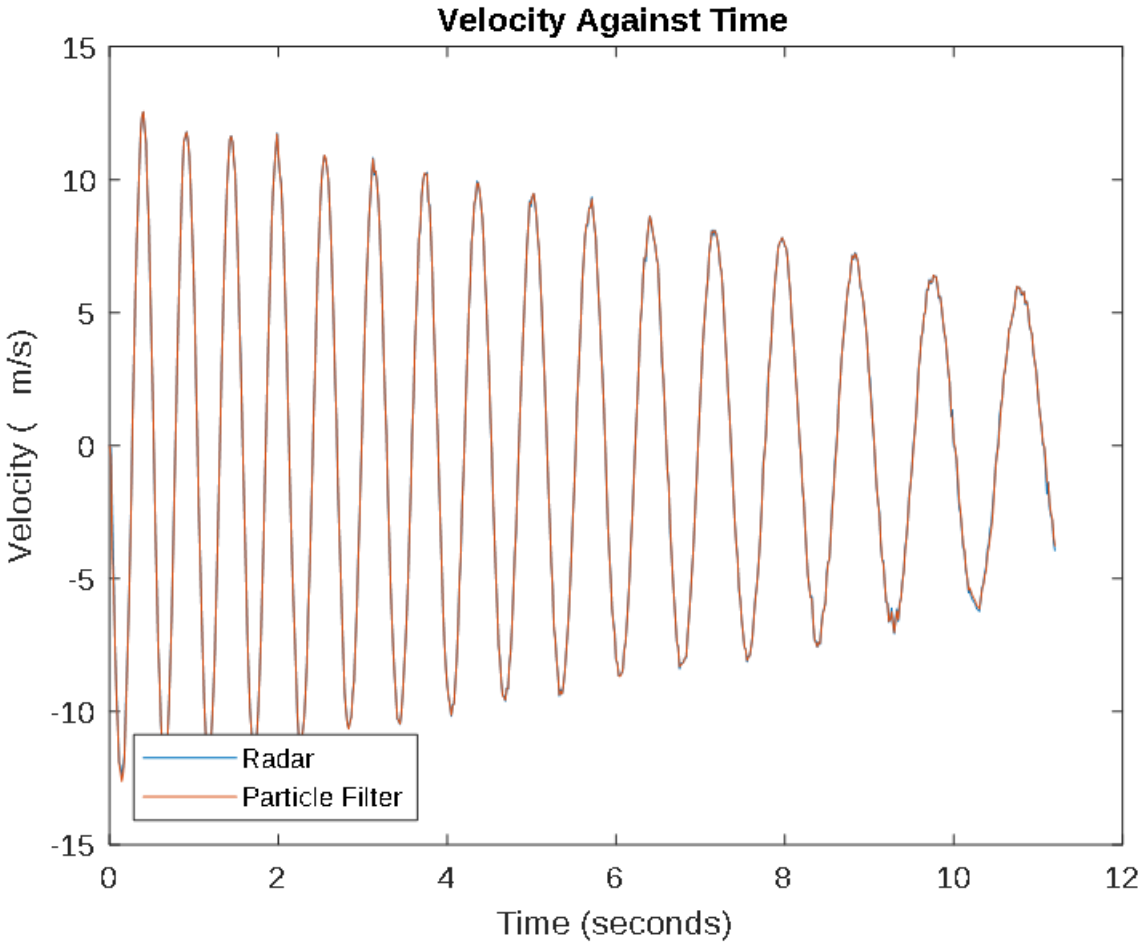



Frequency
Frequency (Hz)
Time (seconds)
0
0.5
1
1.5
2
2.5
0
2
4
6
8
10
12



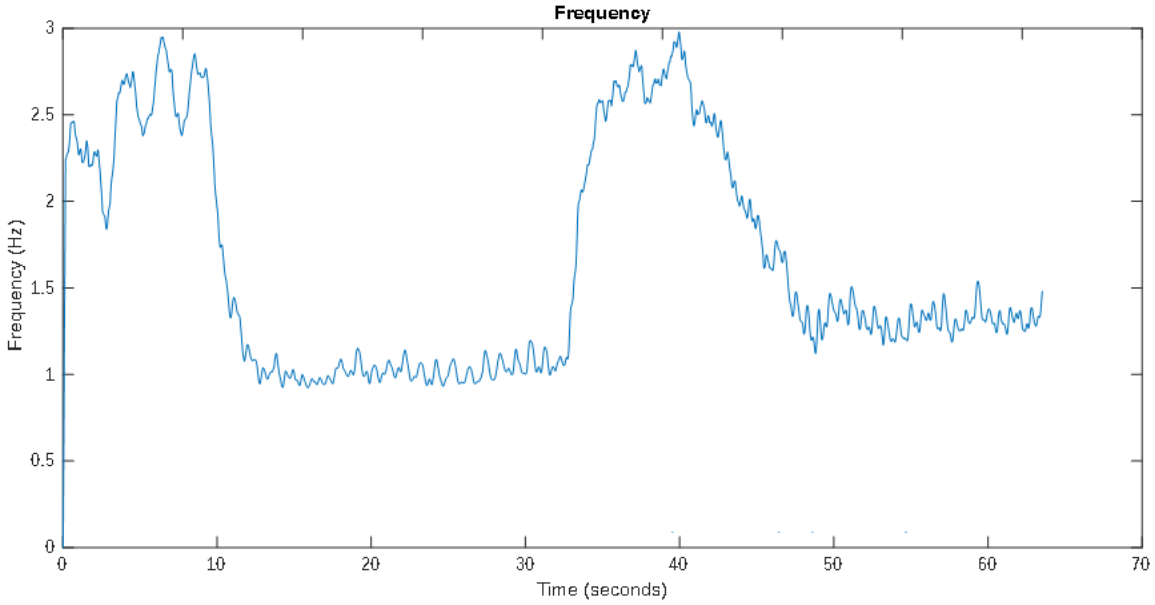
The filter was also able to correctly follow and identify the decreasing frequency.

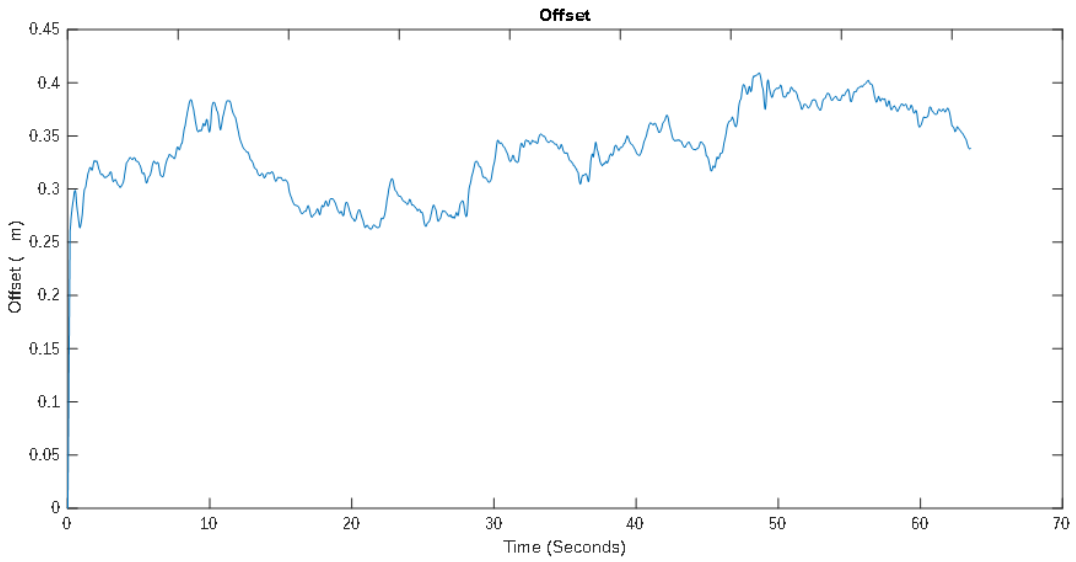

Range Against Time
Range (m)
Time (seconds)
Radar
Particle Filter
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10
12

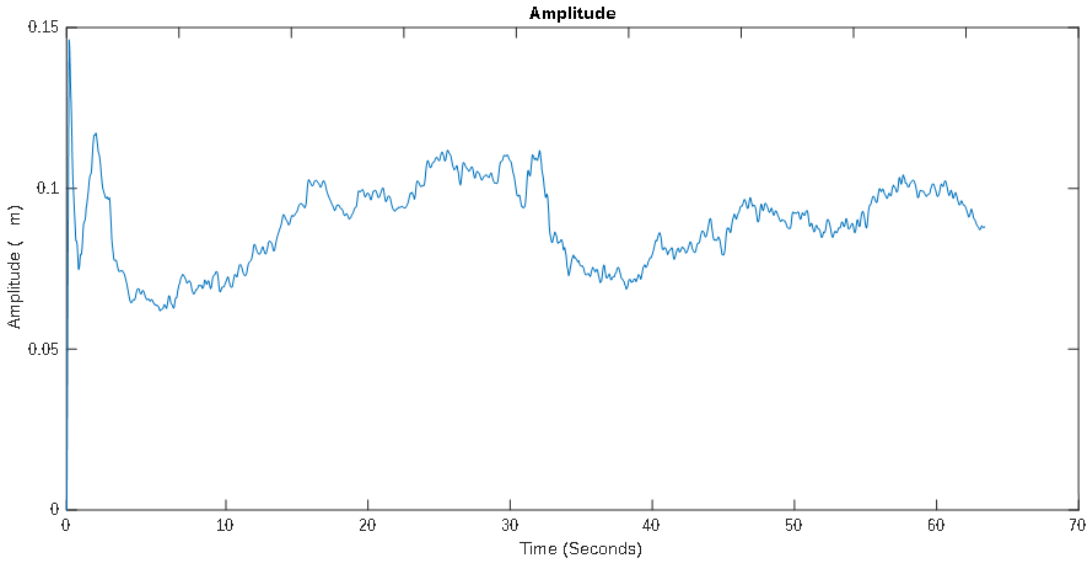

Velocity Against Time
Velocity (m/s)
Time (seconds)
Radar
Particle Filter
15
10
5
0
-5
-10
-15
0
2
4
6
8
10
12

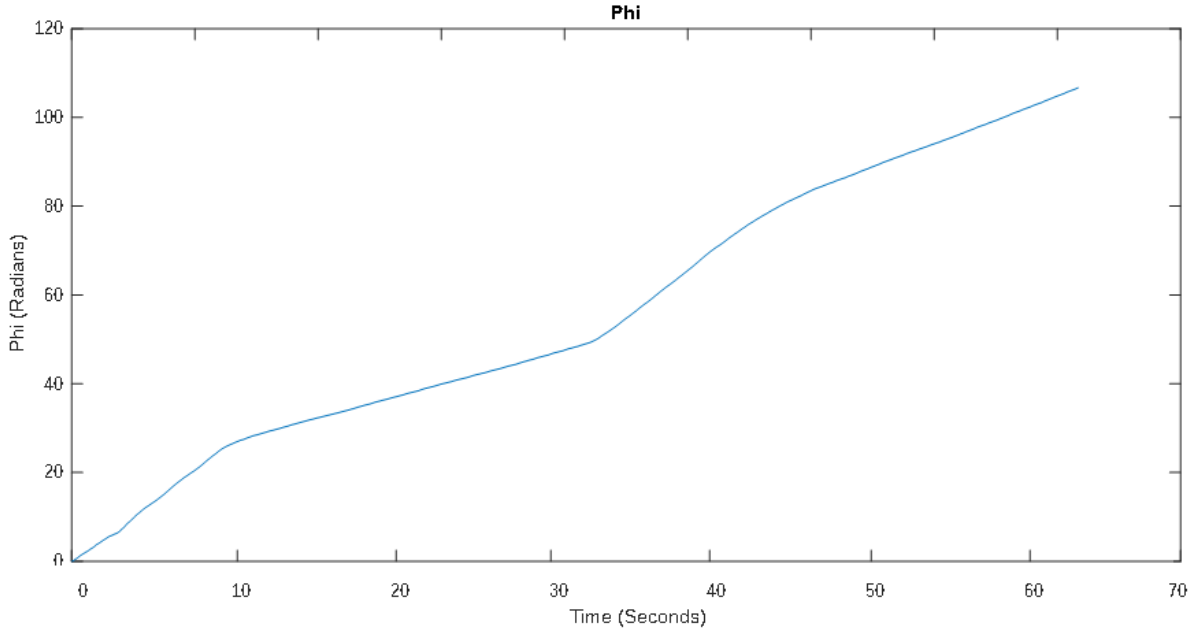
4.3 Experimentation with real radar data

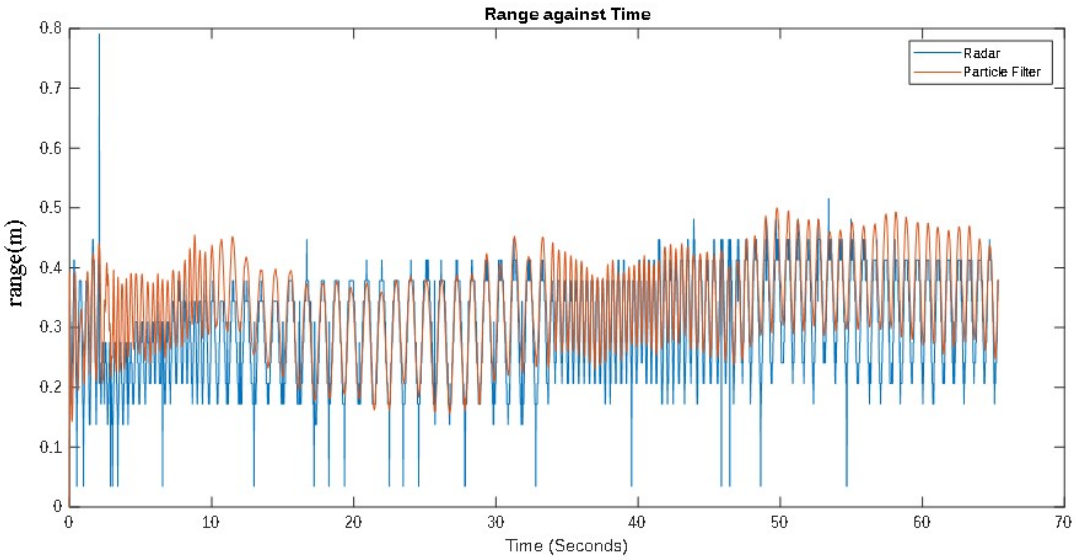
The program automatically records the performance analysis graphs after each run. An example from the most recent run shows fairly good beat prediction. The good performance was confirmed by the following graphs, as well as actual testing (beats of the orchestra fall on the down beats of the hand)

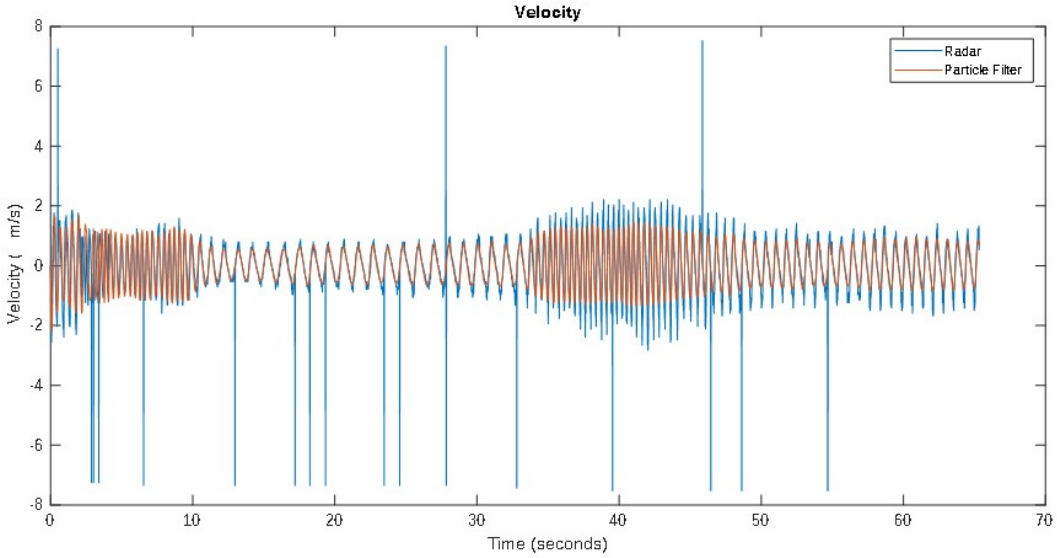





Amplitude
Amplitude (m)
Time (Seconds)
0.15
0.1
0.05
0
0
10
20
30
40
50
60
70


Phi
Phi (Radians)
Time (Seconds)
120
100
80
60
40
20
0
0
10
20
30
40
50
60
70


Range against Time
Radar
Particle Filter
range(m)
Time (Seconds)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
10
20
30
40
50
60
70


Velocity
Radar
Particle Filter
Velocity (m/s)
Time (seconds)
8
6
4
2
0
-2
-4
-6
-8
0
10
20
30
40
50
60
70

5 CONCLUSION AND FUTURE WORK

5.1 Conclusion and Future Work

While the project does currently fulfil what it aimed to do, there are still some areas for improvement. One issue that was still not fully figured out, is how to effectively track the shape of the hand movement. One of the attempts made to do this was to implement a second, higher level filter, with the sole purpose of tracking the amplitude ratio for the first and second harmonic. A fourier window was implemented, which collected the range data during conduction. The duration of this fourier window is set to be the longest possible period (from the slowest possible frequency). We chose the slowest possible frequency as 2/3 Hz, or 40 BPM.

Once our window is full, a fourier transform algorithm is applied, which carries our wave to the frequency domain. From here we can identify the amplitude of our 2 harmonics. The captured amplitudes were fed to a basic alpha beta filter for smoothing, then fed back to our particle filter to help improve prediction.

Due to the nature of this mechanism (having to collect a full fourier window), the tracking for the amplitude ratio ends up rather slow, which can infact lead to degraded accuracy. Figuring out a faster way to track the amplitude ratio can be a point for future improvement.

One drawback of the current implementation is the fact that it's based on only one axis. While this does seem to be enough for the purpose of the project, namely the showcasing of an application of radar technology, it doesn't have much real use otherwise. By adding the capability to follow the motion in 3D space, this project could have a lot of potential for musicians. Such an addition would allow our system to interpret the hand motion like a real orchestra and conductor. This can be used by musicians to extend their musical expression capacities. Ex: changing the emotional signature of the music in a live performance depending on the hand motion pattern. A suggested device for this application would be the Intel RealSense.

5.2 Intel RealSense

The Intel RealSense is a device containing vision processors, depth and tracking capabilities, and depth cameras, all based on an open source, cross platform SDK. Already it's used in several applications such as autonomous drones, robots, AR and VR, smart homes and much more. Using the reals sense depth camera, we can add motion tracking capabilities to our project in an easy to setup form.

Fig 4.17 Real sense radar

APPENDIX

Radar technical data

Dimensionen 2D-FFT	2K…8K x 8…256
Abtastrate ADC	100 MS/s
Maximale getestete Aktualisierungsrate	ca. 500 Hz @ 2K x 32
Minimale Periodendauer der Rampe	54 µs @ 2K 74 µs @ 4K 115 µs @ 8K
Rampen-Bandbreite	5 GHz
Kleinster Abstand zwischen Range-Bins	3,1 cm @ 8K
Kleinster Abstand zwischen Doppler-Bins	0,04 m/s @ 8K x 256
Maximaler Abstand	127 m @ 8K
Maximale rel. Geschwindigkeit	±11,4 m/s @ 2K
Leistungsaufnahme	7 W

Bibliography

[1] Kroese, D. P., Brereton, T., Taimre, T., & Botev, Z. I. (2014). Why the Monte Carlo method is so important today. *Wiley Interdisciplinary Reviews: Computational Statistics, 6*(6), 386-392. doi:10.1002/wics.1314

[2] Rlabbe. (n.d.). Rlabbe/Kalman-and-Bayesian-Filters-in-Python. Retrieved August 28, 2020, from https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

[3] Thrun, S., Burgard, W., & Fox, D. (2006). *Probabilistic robotics*. Cambridge, MA: The MIT Press.

[4] Malanowski, M., & Kulpa, K. (2011). Target Detection in Continuous-Wave Noise Radar in the Presence of Impulsive Noise. *Acta Physica Polonica A, 119*(4), 467-472. doi:10.12693/aphyspola.119.467

[5] Skolnik, M. I. (1990). *Radar handbook*. New York, NY: McGraw-Hill.

[6] Dansereau, D. G., Brock, N., & Cooperstock, J. R. (2013). Predicting an Orchestral Conductor's Baton Movements Using Machine Learning. *Computer Music Journal, 37*(2), 28-45. doi:10.1162/comj_a_00173

[7]Joyce, James (2003), "Bayes' Theorem", in Zalta, Edward N. (ed.), The Stanford Encyclopedia of Philosophy (Spring 2019 ed.), Metaphysics Research Lab, Stanford University, retrieved 2020-01-17

[8] Stachniss, C. (2012). Particle Filters for Robot Navigation. doi:10.1561/9781601987594

[9] Wolff, D. (n.d.). Radar Basics. Retrieved August 28, 2020, from https://www.radartutorial.eu/11.coherent/co06.en.html

[10]Hochschule Heilbronn, Laufende Projekte, Retrieved August 28, 2020, from https://www.hs-heilbronn.de/laufende-projekte-622f0368808f2e11

[11]G Welch and G Bishop. An introduction to the kalman filter. department of computer science, university of north carolina at chapel hill, chapel hill, nc. Technical report, TR 95-041 (http: flwww. cs. unc. edu/Research/tech-report. Html), 1995.

[12] Keisuke Fujii, The ACFA-Sim-J Group Extended kalman filter. Refernce Manual, 2013.

[14]Doucet, A., Freitas, N., & Gordon, N. (2001). An Introduction to Sequential Monte Carlo Methods. *Sequential Monte Carlo Methods in Practice,* 3-14. doi:10.1007/978-1-4757-3437-9_1

[15]Markov Chain Monte Carlo. (2004). *Markov Chain Monte Carlo Simulations and Their Statistical Analysis,* 128-195. doi:10.1142/9789812703637_0003

[16]Crisan, D., & Doucet, A. (2002). A survey of convergence results on particle filtering methods for practitioners. *IEEE Transactions on Signal Processing, 50*(3), 736-746. doi:10.1109/78.984773

[17]Dartmouth College, Monte Carlo Integration. Referenced from: https://cs.dartmouth.edu/~wjarosz/publications/dissertation/appendixA.pdf

[18]Bevilacqua, F., Zamborlin, B., Sypniewski, A., Schnell, N., Guédy, F., & Rasamimanana, N. (2010). Continuous Realtime Gesture Following and Recognition. *Gesture in Embodied Communication and Human-Computer Interaction Lecture Notes in Computer Science,* 73-84. doi:10.1007/978-3-642-12553-9_7

[19]Gordon, N., Salmond, D., & Smith, A. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F Radar and Signal Processing, 140*(2), 107. doi:10.1049/ip-f-2.1993.0015

[20]Morita, H., Hashimoto, S., & Ohteru, S. (1991). A computer music system that follows a human conductor. *Computer, 24*(7), 44-53. doi:10.1109/2.84835

[22] Orio, Nicola & Lemouton, Serge & Schwarz, Diemo. (2003). Score Following: State of the Art and New Developments. In Proceeding of the International Conference on New Interfaces for Musical Expression, pp. 36–41.

[23]Welch, G., and G. Bishop. 1995. “An Introduction to the Kalman Filter.” Technical Report TR95-041, University of North Carolina, Department of Computer Science, Chapel Hill, North Carolina.

[25] Yuen, S. G., Novotny, P. M., & Howe, R. D. (2008). Quasiperiodic predictive filtering for robot-assisted beating heart surgery. *2008 IEEE International Conference on Robotics and Automation*. doi:10.1109/robot.2008.4543806

[26]Balázs Csanád Csáji (2001) Approximation with Artificial Neural Networks; Faculty of Sciences; Eötvös Loránd University, Hungary

[27] Kurt Hornik (1991) , Neural Networks, 4(2), 251–257.doi:10.1016/0893-6080(91)90009-T

[28] Leshno, M., Lin, V. Y., Pinkus, A., & Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks, 6*(6), 861-867. doi:10.1016/s0893-6080(05)80131-5

[29] Hassoun, M. (1995) Fundamentals of Artificial Neural Networks MIT Press, p. 48

[30] Del Moral, Pierre (1996). "Non Linear Filtering: Interacting Particle Solution" (PDF). Markov Processes and Related Fields. 2 (4): 555–580.

[31] Liu, J. S., & Chen, R. (1998). Sequential Monte Carlo Methods for Dynamic Systems. *Journal of the American Statistical Association, 93*(443), 1032-1044. doi:10.1080/01621459.1998.10473765

[32] Liu, J. S. (2004). Sequential Monte Carlo in Action. *Springer Series in Statistics Monte Carlo Strategies in Scientific Computing,* 79-104. doi:10.1007/978-0-387-76371-2_4

[33] Maskell, S. (2001). A tutorial on particle filters for on-line nonlinear/non-Gaussian Bayesian tracking. *IEE International Seminar Target Tracking: Algorithms and Applications*. doi:10.1049/ic:20010246

[34] Hollmann. (n.d.). Radar World -. Retrieved August 28, 2020, from https://www.radarworld.org/

[35] D. Simon, Optimal state estimation: Kalman, H innity, and nonlinear approaches. John Wiley & Sons, 2006.

[36] Wan, E., & Merwe, R. V. (n.d.). The unscented Kalman filter for nonlinear estimation. *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*. doi:10.1109/asspcc.2000.882463

[37] Julier, S. J., & Uhlmann, J. K. (1997). New extension of the Kalman filter to nonlinear systems. *Signal Processing, Sensor Fusion, and Target Recognition VI*. doi:10.1117/12.280797

[39]A Tutorial on Particle Filters for Online Nonlinear/NonGaussian Bayesian Tracking. (2009). *Bayesian Bounds for Parameter Estimation and Nonlinear Filtering/Tracking*. doi:10.1109/9780470544198.ch73

[40] Hausman, D. (1999). Independence, invariance and the causal Markov condition. *The British Journal for the Philosophy of Science, 50*(4), 521-583. doi:10.1093/bjps/50.4.521

[41] Baba, T., Hashida, M., & Katayose, H. “VirtualPhilharmony”: A Conducting System with Heuristics of Conducting an Orchestra

[42] Borchers, J., Samminger, W., & Muhlhauser, M. (n.d.). Personal Orchestra: Conducting audio/ video music recordings. *Second International Conference on Web Delivering of Music, 2002. WEDELMUSIC 2002. Proceedings.* doi:10.1109/wdm.2002.1176198

[43] Borchers, J., Lee, E., Samminger, W., & M. Hlh User, M. (2004). Personal orchestra: A real-time audio/video system for interactive conducting. *Multimedia Systems, 9*(5), 458-465. doi:10.1007/s00530-003-0119-y

[46] Dillon, R., Wong, G., & Ang, R. Virtual orchestra: An immersive computer game for fun and education

[48] Morita, H., Hashimoto, S., & Ohteru, S. (1991). A computer music system that follows a human conductor. *Computer, 24*(7), 44-53. doi:10.1109/2.84835

[51] Schertenleib, S., Gutierrez, M., Vexo, F., & Thalmann, D. (2004). Conducting a virtual orchestra. *IEEE Multimedia, 11*(3), 40-49. doi:10.1109/mmul.2004.5

[54] Charles, R. Q., Su, H., Kaichun, M., & Guibas, L. J. (2017). PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2017.16

[55] Driedger, J., & Müller, M. (2016). A Review of Time-Scale Modification of Music Signals. *Applied Sciences, 6*(2), 57. doi:10.3390/app6020057

[56] Driedger, Johnathan. "Time-Scale Modification Algorithms for Music Audio Signals", Master's thesis, Saarland University, Saarbrücken, Germany, 2011

[57] Audio Stretching in matlab, Retrieved August 28, 2020, from https://www.mathworks.com/help/audio/ref/stretchaudio.html