Distributed Robot Perception Report

Italy {luca.sartore-1}@studenti.unitn.it

Abstract

In the scenario presented in this project a robot called "Runner" is trying to escape three other robots called "Chasers" (that are trying to catch him). Both the Runner and the Chasers are free to move in any direction, and can detect the presence and position of other robots if they are within a certain radius. In this project I implemented 3 different logics for the runners. One called "Baseline" where the Chasers are simply reacting to the latest input and don't have any form of memory; One called "Gaussian" where the chasers approximate the probability distribution of the runner's position using a gaussian, and use then a kalman filter for tracking; And finally one called "PF" (or Particle Filter) where the same probability is approximated using a particle filter. The results show how the particle filter (even though its computational cost is much higher) significantly outperforms the other two when evaluated using the "time to catch" as the key metric.

1 Introduction

1.1 Distributed particle filter

As we will see later on, the proposed solution leverage a particle filter, however a distributed implementation of such a filter is not trivial, therefore is worth going trough the math here, so that the process will be clearer later. The proposed implementation is taken from the "Cooperative and Graph Signal Processing" [1] book.

In the simplest version of a particle filter, each particle is assigned a weight, and at each iteration the weight is modified based on some observations. Then a dynamics is applied to the particles, and they are re-sampled (but we are interested in the distributed part, so I will not go into further details here). The weights update step can be written as:

$$w_n^{(m)} = w_{n-1}^{(m)} f(\mathbf{z}_n|\mathbf{x}_n^{(m)}), \quad m = 1, 2, \ldots, M.$$

Where n indicates the time iteration, m is the index of the particles and M is the number of particles in the filter. $f(\mathbf{z}_n|\mathbf{x}_n^{(m)})$ is the probability of measure z_n assuming x_n was the state of the system.

calculating $f(\mathbf{z}_n|\mathbf{x}_n^{(m)})$ in a centralized way can be done as such:

$$f(\mathbf{z}_n|\mathbf{x}_n) = \prod_{k=1}^{K} f(\mathbf{z}_{n,k}|\mathbf{x}_n).$$

Where $\mathbf{z}_{n,k}$ is the measurement of robot k. However, we will need to implement a distributed version, and in order to do so we will first need to use logarithm properties to transform the multiplication into a summation:

$$f(\mathbf{z}_n|\mathbf{x}_n) = exp(\sum_{k=1}^{K} \ln f(\mathbf{z}_{n,k}|\mathbf{x}_n))$$

In the next step we will approximate the function f with a summation of R kernel functions weighted with parameters α. There are many options for the kernel functions, in this work I choose chebyshev polynomials [3].

$$\ln f(\mathbf{z}_{n,k}|\mathbf{x}_n) \approx \sum_{r=1}^{R} \alpha_{n,k,r}(\mathbf{z}_{n,k})\varphi_r(\mathbf{x}_n)$$

putting everything together we get:

$$f(\mathbf{z}_{n,k}|\mathbf{x}_n) \approx exp(\sum_{k=1}^{K}\sum_{r=1}^{R} \alpha_{n,k,r}(\mathbf{z}_{n,k})\varphi_r(\mathbf{x}_n))$$

finally we can resolve the summation over K (the agents) and reduce the problem to a global set of parameters that will be the same for all agents:

$$f(\mathbf{z}_{n,k}|\mathbf{x}_n) \approx exp(\sum_{r=1}^{R} \alpha_{n,r}(\mathbf{z}_n)\varphi_r(\mathbf{x}_n))$$

Now all we need to do is to compute the coefficients α in a distributed way, and to do this we can start with an initial guess for each agent that only depends on the measurement taken by itself:

$$\zeta_{k,r}^{(0)} = \alpha_{n,k,r}(z_{n,k}).$$

Note that to find the coefficients $\alpha_{n,k,r}$ we just need to minimize the mean square error between the approximation function and the actual local log-likelihood of the agent. The mean square error is calculated w.r.t. all the points were particles are located, and the problem has a simple closed-form solution (more details in the book [1]).

From an initial guess we can then do a finite set of iterations that spread the information among the agents:

$$\zeta_{k,r}^{(i)} = \sum_{k' \in N_k} \omega_{k,k'}^{(i)} \zeta_{k',r}^{(i-1)}.$$

Parameters ω are the traditional weights used in average consensus algorithms and can be set in many ways. For this work we set all parameters at $1/K$ given that all the agents have the same sensor's precision, and the topology chosen is a fully connected graph.

In theory with enough iterations the agents should converge to a unified guess. For our work, one iteration was enough given the topology.

$$\lim_{i\to\infty} \zeta_{k,r}^{(i)} = a_{n,r}(\mathbf{z}_n)$$

1.2 Problem Formulation

In the proposed scenario 3 Runners and one Chaser are free to move in a rectangular map. All robots are equipped with a sensor that can measure the area around the robot.

The runner always moves in a straight line, and "bounces" whenever it hits a wall or detects a chaser. The bouncing angle is calculated using the specular reflection rule [2], with the addition of some noise.

In the same map there are also 4 "Fake Runners", those are robots that act exactly like a runner and can sometimes "trick" the chaser's sensors, and give false positive measures.

2 Adopted Models

2.1 Communication System

The proposed project is implemented entirely as a simulation, therefore there are not many details here. Each robot's controller is assigned to a different thread, and message queues are used to exchange any information. The robots are always connected to each others, and form a complete graph.

2.2 System Model

2.2.1 Chaser's sensors

The chasers are equipped with a "radar-like" sensor that can detect and measure the position of another robot. The sensor can be described with the following parameters:

- **Detection radius:** The radius/range of the sensor (set to 4 by default)
- **False negative rate:** The probability of a sensor not detecting a runner inside the detection radius (set to 25% by default)
- **False positive rate:** The probability of the sensor misclassifying a fake runner as a real runner (set to 25% by default)
- **Measurement STD:** The parameter of a gaussian noise added on top of the measure. (set to 0.5 by default)

2.2.2 Runner's sensors

The runner's sensor has the exact same behavior as the chaser's one, with the only difference being that it cannot generate false positives, but only false negatives.

2.2.3 Actuators

In the simulation both chasers and runner use holonomic motion (meaning that they can move in any direction instantly), however they have a limited speed.

In all tests, the runners and the chasers had the same maximum speed.

2.2.4 Chaser's controllers

The primary contribution of the project is in this subsection. I tested three different controllers, where two of them have to be considered "baselines" and the third one is instead the one we are interested in. The three controllers are:

- **Baseline:** This controller does not have any communication between the robots, and does not keep track of any probability distribution of any kind (the only state that it keeps internally is the position where a runner was last spotted)
- **Gaussian:** This controller models the probability distribution of the runner's position as a multivariate gaussian, and apply a kalman filter to track it.
- **PF:** This controller also tracks the probability distribution of the runner's position but it does so by using a Particle Filter.

3 Solution

In this section we will explain the implementation of all the three controllers that were tested in this project, with a particular focus on the "PF" one.

One thing that accompanies all three of the controllers is that they all have two separate operating modes:

- **Searching mode:** This mode is active when the chaser does not know where the runner is, and is moving "somewhat randomly" trying to find it.
- **Chasing mode:** This mode is activated when the chaser has a clear idea of where the runner is, and starts following in order to catch it.

3.1 Baseline

The "Baseline" chaser has a simple logic that can be described as follow:

It keeps an internal state of the current "objective" (i.e. a point of the map that it wants to reach). At each iteration the robot moves towards the objective. The objective can be set into two different ways:

- Set to the observed value every time a not-null measure is made
- Set to a random position within the map every time we reached current objective

The first strategy represent the "Chasing mode" while the second one represent "Searching mode".

3.2 Gaussian

The gaussian controller has an internal state that contains a mean and a covariance matrix of a multivariate normal distribution. This distribution represents the PDF of the runner's position and it is then used for the controller.

The distribution is kept up to date using a traditional distributed kalman filter (that uses information vector and information matrix).

3.2.1 Controller

The controller implementation is straightforward. It implements searching and chasing modes in the following ways:

- **Searching mode:** The chaser samples a random point from the gaussian distributed, and start moving towards it, when the point is reached a new point is sampled, and the cycle repeats.
- **Chasing mode:** The robot moves towards the center of the gaussian. This mode is activated only when the variance in all dimensions is below a certain threshold.

3.2.2 State and Transition function

The gaussian filter only tracks the chaser's x and y coordinates.

The state transition function add a gaussian noise on top of the state, The noise variance is a multiple of the runner's speed. This is an approximation, as a more realistic update step would require us to track the runner's speed as well, however this can't be done for two reasons:

- The sensors don't give speed as a measure
- When the chaser "bounce" on a wall, the update in the speed is non-linear

3.2.3 Handling negative measures

When the chasers get a negative measure (i.e., no runner found within the radius) they are getting some information, and we would like to update the gaussian accordingly, however the resulting PDF cannot be easily encoded in a gaussian (see figure 1 as an example). Any gaussian approximation of such a distribution would be a bad approximation, not to mention that it probably would not have a closed form solution. For this reason, I chose to simply treat negative measures as a measure centered in zero with a absurdly high variance.

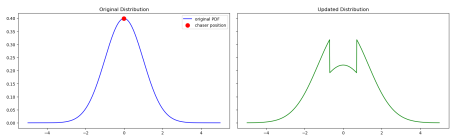

Figure 1: negative measure effect on a gaussian PDF

3.3 Particle Filter

The third and final controller implemented for this project is based on a particle filter. We already discussed the math behind a distributed particle filter in section 1.1. Now two things are left to explain:

- Calculating $f(\mathbf{z}_{n,k}|\mathbf{x}_n)$ (discussed in 3.3.1)
- Defining the controller (discussed in 3.3.2)

3.3.1 Calculating the probability of a measure

Calculating $f(\mathbf{z}_{n,k}|\mathbf{x}_n)$ can be done by splitting it into M parts: $f(\mathbf{z}_{n,k}|\mathbf{x}_{nk}^{(m)})$ (where M indicates the number of particles in the filter) and $\mathbf{x}_{nk}^{(m)}$ represent the m-th particle of agent k.

To calculate $f(\mathbf{z}_{n,k}|\mathbf{x}_{nk}^{(m)})$ we first need a way to model the "false positive" measures. In theory we could model them in a precise way assuming that we kept track of not only the probability distribution of the runner, but also of the probability distribution of the fake runners.

This however can be prohibitively expensive, and the particle filter is already quite heavy in terms of computational resources, therefore the second best option is to approximate the probability of a false positive measure as such:

$$p_{fp} = p_{inside} \times FPR \approx \frac{\pi R^2}{A} \times FPR$$

were p_{fp} is the probability that one fake runner is detected as a false positive, p_{inside} is the probability that the fake runner is inside the detection radius of the chaser and FPR is the false positive rate. p_{inside} is approximated by using R (the detection radius of the robot) and A (the area of the map).

This approximation has an underlying issue: with this formulation we are implicitly saying that false positive measures are independent from each other, but this is not the case. If a false positive measure occurs then more false positive are likelier to occur because p_{inside} will be higher. The results shows that even with this issue the particle filter is still able to outperform the other techniques, however it is worth keeping this limitation in mind.

Now we can split the calculation of $f(\mathbf{z}_{n,k}|\mathbf{x}_{nk}^{(m)})$ into two subcases depending on the measure being a negative or a positive:

Negative measure case: If the measurement is a negative (i.e. no runner detected) the probability of set measure given a particle $\mathbf{x}_{nk}^{(m)}$ can be defined as:

$$f = \begin{cases} (1 - p_{fp})^{N_{fr}} & \text{if } \mathbf{x}_{nk}^{(m)} \text{ is out of range} \\ (1 - p_{fp})^{N_{fr}} \times FNR & \text{otherwise} \end{cases}$$

Where N_{fr} represents the number of fake runners, and FNR is the false negative rate. The formulation considers two separate cases depending on the particle being inside or outside the range of the sensor.

Positive measure case: If the measurement is a positive (i.e. a runner is detected) the probability of set measure given a particle $\mathbf{x}_{nk}^{(m)}$ can be defined as:

$$f = \rho(\mathbf{x}_{nk}^{(m)}; \sigma^2, \mathbf{z}_{n,k}) \times (1 - FNR) + \frac{N_{fr}}{A} \times FPR$$

Where $\mathbf{x}_{nk}^{(m)}$ is the position of particle m, σ^2 is the variance of the measure and $\mathbf{z}_{n,k}$ is the measurement value. Then FNR and FPR are false negative and false positive rates respectively, N_{fr} is the number of fake runners and A is the area of the map. Here we are approximating the probability that a fake runner is in a specific position as a uniform distribution (N_{fr}/A).

In the formula we don't need to have two special cases for measurement inside and outside the detection radius as we will never get positive measures outside the range of the sensor.

Final step Once each agent compute the probability of each particle being true given the measure it proceed to find the kernel function multiplier that best approximate the probability function, and then proceed with the exchange of message that will result in a global guess (this was already explained in 1.1).

The kernel functions chosen were Chebyshev polynomials [3], as they are best suited for circumstances where the approximated function has a rectangular domain (such as our case).

I selected a 10th order approximation for the x axis, and a 10th order approximation for the y axis, this resulted in 121 coefficients.

3.3.2 Controller

The controller is straightforward, and is once again split into two separate modes:

Searching mode : This mode is the default one, and it picks a random particle (with the choice being conditioned by the particle's weight and the distance from the current position) and starts to move towards it.

It stops this movement if either it reaches the destination, or the controller notes a sharp decrease in the target's likelihood (meaning that probably a chaser already went there and verified that there was no runner to be found)

Chasing mode : This mode is activated only when the particles are concentrated within a certain area, and makes the chaser move towards the median of the particles.

3.3.3 Issues and solutions

In this section I will describe a few issues and solutions that are specific to my particle filter implementation:

The "Ghost" issue: Given that we are approximating the probability of measures with a finite number of kernel functions, it is natural that there will be some errors, and that some "ripple" effect will be generated (as we can see in figure 2). It can sometimes happen that those ripples are concentrated in the same zones for a few consecutive iterations, and therefore the probability of particles in that zone will increase even tho no "real" measure suggest that.

This issue was especially frequent in the borders of the map, where the approximation was worst due to lack of samples. To solve this issue I added "virtual" particles along the map's border. These particles were only considered when calculating the approximation coefficients and significantly reduced the "ghost" issue.

The loss of diversity issue: Sometimes particles converge into a single zone of the map (due to, for example, a ghost issue or some multiple false positives happening in a row). This meant that a measurement made in other zones of the map would be completely ignored (given that there were no particles there). This could be solved by adding more particles so that they would better approximate the distribution, but that is expensive, so the adopted solution was to apply a custom resampling logic:

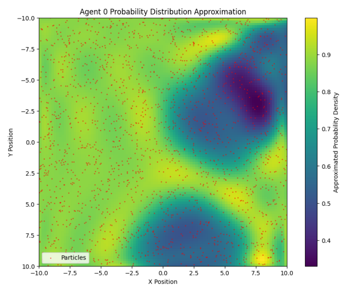

Figure 2: Approximated version of the probability of measures given particle position. ($f(\mathbf{z}_n|\mathbf{x}_n)$)

When resampling 400 out of the 2000 particles were spread randomly in the map, and were given a total probability of 1%.

The state transition issue: Unfortunately, to apply a correct state transition function, we would need to know the velocity of the robot. However, that variable is not tracked in the current version of the particle filter, and therefore the adopted solution uses a random movement proportional to the runner's velocity.

4 Implementation Details

The simulation was implemented in Python, with a frontend for visualization built with PyGame Community Edition and Matplotlib. All agents' controllers were implemented in a separate thread for better performance, and message queues were used for message exchange. The Python version used was 3.14, which thanks to the removal of the GIL allowed better multithreaded execution. All tests were run on a Ryzen 5 5600 desktop CPU.

5 Results

5.1 Proposed Methods Performance

In this section we will compare the results obtained by the three proposed methodologies using three key metrics:

- **TTC (or Time To Catch):** This is the total time (in simulation steps) that the chasers took to catch the runner.
- **TTFC (or Time To First Contact):** This is the time (in simulation steps) from the beginning to when the runner entered the detection radius of a chaser.
- **TChase:** defined as $TTC - TTFC$.

The figure 3 shows the three metrics across all three approaches, with the error bars representing the standard deviation of the scores.

The standard deviation is high across all samples due to the fact that the time for a catch is highly dependent on the initial position of the robot. To mitigate this issue, 100 tests were made for each strategy, and the seed was set to the same value in the i-th test of every strategy.

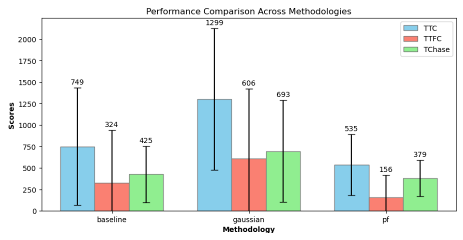

Figure 3: comparison between the proposed solutions

The table below shows the t-test's P-value of the comparison between the various metrics and solutions.

We can easily see from the table that, even though the variance is pretty high, thanks to the high number of samples we can say with high confidence that all the differences shown are statistically significant.

	TTest: P-Value		
	TTC	TTFC	TChase
baseline vs gaussian	<0,000001	0,000413	0,000008
baseline vs pf	0,005782	0,006214	0,310494
pf vs gaussian	<0,000001	<0,000001	<0,000001

Explanation of the observed results: Two interesting trends are clear when looking at figure 3. The first one is that the gaussian controller performs really poorly, even worse than the simple baseline one. This can be explained with three key reasons:

- The gaussian is not really suited for the search process. It cannot provide a reasonable approximation of where the runner is, making the search ineffective. In particular, if the variance is too low, the search process is focused too much on the center of the map, and if the variance is too high, most of the probability density is outside the borders of the map.

- As discussed in section 3.2.3, encoding the information associated with a negative measure in a gaussian is hard. This results in this controller having less information than the particle filter one, and it also made it more susceptible to false positive measures. (Once a false positive measure occurs, the associated information cannot be undone by a subsequent negative measure.)

- The fact that we do not keep the direction in the state of the system makes the tracking process less effective. At every iteration, the gaussian's variance simply increases, and then it is conditioned based on the measures. This makes it so that the center of the gaussian is always a bit behind the robot's position, and this causes the chasing process to take a bit longer.

Observing the comparison between the "Baseline" and the "PF" solution reveals some more interesting insights: the difference in TChase is really small at about 12%, while the difference in TTFC is much higher at approximately 107%. This suggests that the main benefits of using an informed search strategy is not really the tracking of the target once it has been spotted, but rather has more to do with the ability to search for the target in an efficient way even before the target has been spotted.

This effect can be seen in figure 4 where the chasers have a somewhat accurate guess on the position of the runner, even if none of them has yet been able to see it.

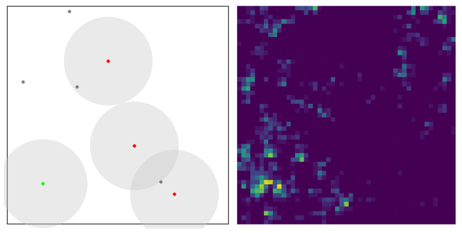

Figure 4: Chasers guess of the runner's position (left image show the map with chasers in red an runner in green, right image show a 2D histogram of the position of the particles)

5.2 Computational Costs

Figure 5 shows the computational cost of the proposed methods. There is nothing surprising here. The particle filter is, as expected, by far the most expensive.

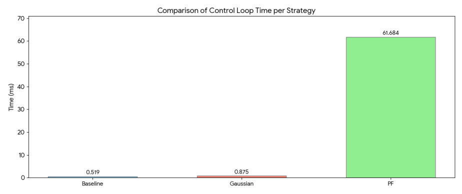

Figure 5: Time needed for one control loop of various techniques

5.3 Effect of Various Variables

In this section we will explore the effects that various variables have on the time to catch metric. The idea is to understand what is the most effective way to improve the performance if we have, for example, a limited budget for sensors.

The variables that were tested are:

- Number of chasers
- Detection radius
- False positive rate
- False negative rate

I should mention that the run-to-run variance on this is relatively high at about 60 points on average. The results were calculated with the same

100 fixed-seed tests used for the previous benchmarks. However, fixing the seed only makes the initial condition the same and does not have an influence on how the system evolves. Unfortunately, increasing the sample size and reducing the variance even further was complicated, as these tests already required 6+ hours of compute time.

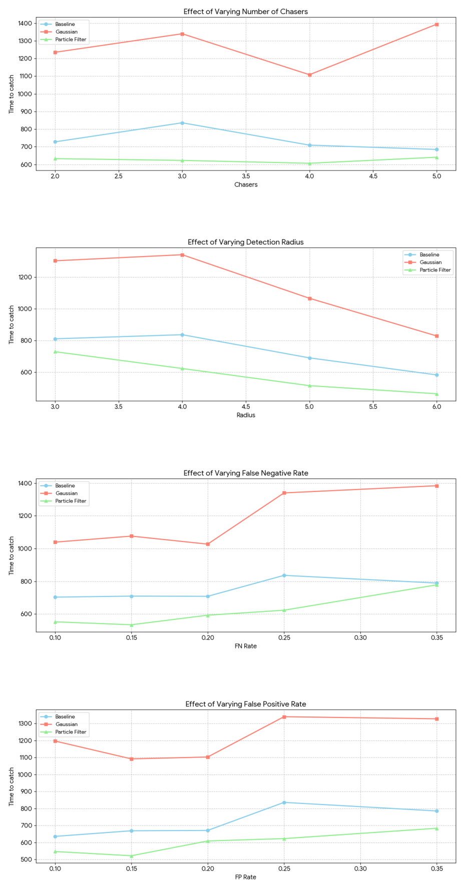

Figure 6: Effect of different variables on the time to catch metric

As we can see in figure 6, the most important variable seems to be the detection radius, with all other variables having a smaller (although still measurable) impact.

We can also see how sometimes a variable's effect seems to have a non-monotonic effect on the time to catch. However, as we said earlier, these tests have some high variance associated with them, so the correct way to interpret this result is that the effect has an impact that is comparably small.

6 Conclusions

In this report, we discussed three different approaches for controlling chasers in a runner-chaser scenario.

The results have shown that the biggest gains can be had in the first part of an episode (the search) rather than the second part (the chase). This resulted in methods that can effectively represent complex PDFs (namely, the particle filter) greatly outperforming other methods.

The work done still has some limitations. In particular, the way the probability of false positives has been approximated could be improved if we were to track the position of fake runners as well.

Furthermore, adding the runner's velocity to the state (rather than just tracking the position) could further improve the performance of the algorithms.

These proposed improvements are a great opportunity for potential future work; However, the main challenge still remains the computational cost of techniques such as the particle filter, and this challenge would only increase if we started tracking fake runners or adding components to the state.

Chaser coordination is also something that was not treated in this report, but could also improve results.

References

[1] A. H. Sayed, P. M. Djurić, and F. Hlawatsch, "Cooperative and Graph Signal Processing", Academic Press, 2018, pp. 169–207.

[2] `https://en.wikipedia.org/wiki/Specular_reflection`

[3] `https://en.wikipedia.org/wiki/Chebyshev_polynomials`

This report is the final document for the course of "Distributed Robot Perception".