Target localization and tracking using a WSN

Antonio Minonne, Anna Paola Musio, Samuele Paone, Salvatore Pappalardo
s280095, s281988, s287804, s281621

Introduction

This project work is about simulating an indoor localization and tracking system through a wireless sensor network (WSN) in both centralize and distributed way; moreover we considered sensors deployment in a randomized and fixed grid cases. After implementing all the necessary code for the simulations, we performed a series of tests in order to compare the algorithms and evaluate their own performance.

Code explantion

There are three categories of files:

- experiments,
- algorithms implementations,
- helper functions

We shall explain each file within each category.

Experiments files

These are the most important files. They contain the code used for generating data and analyze it. The detailes of the experiments are discussed in the “Result and Analysis” section.

Algorihtms implementations files

There are 3 files: IST.m, DIST.m and ODIST.m each of which is exactly what it sounds like. They are implemented as functions, as everything in this project but the experiments, for ease to use.

IST

The Iterative Soft Thresholding algorithm is centered. It works as following:

1. Deploy the sensors: the sensors are deployed (using a helper function) inside the room and are positioned accordig to the requested type. If random-positioned, the deployment is the same, with the same seed.
2. Position the target: the target position might be received as an argument. If the target parameter is a scalar (i.e. size(target) == [1 1]), then, it is randomly positioned. When randomly-positioned, the coordinates are the same with the same seed.
3. Training of the sensors: in this phase the dictionar A is created using a helper function.
4. Q computation: this step is unnecesary, but it is historically put since this file was born after DIST.m
5. Initial condition inizialitazion: x0 gets computed. It can be passed as a parameter, this was done to use the file for the O-IST, which was nevere implemented. If the parameter x0 is a scalar (i.e. size(x0) == [1 1]), then x0 is initialized at 0.
6. Measurements: in this step each sensor gets its measure using an helper function.
7. Feng’s theorem: in order to reduce the coherence of the matrix A and to produce a less sparse solution (k > 1), Feng’s theorem is applied to A and y producing B and z.
8. IST algorithm: This is the most important step of the procedure. Here the algorithm is applied using the IST_step function (which is defined at the end of the same file). The complete history of the x vector is mantained, due to the relative small dimensions. Another important thing is that *early stopping* is applied. When the 1-norm (or the 2-norm depending on the case) of x drops below stopThreshold the algorithm is stopped. This is equivalent to saying that the algorithm converged.
9. Results: this section of the code contains an explantion of the output parameter of the function, including the sizes.
10. Plots: if showPlots is true, three figures are shown: the first contains a representation of the room, with target, best estimate and sensor positions; the second one shows the

difference x(t)-x(t-1) for each t and for each component of x; the last one show x when consensus is reached.

The IST_step function does two operations on x . It firstly computes applies the gradient descent to x and then applies the soft-thresholding operator to the result of the former operation.

DIST.m

The DIST algorithm is implemented pretty much as the IST one. We shall describe the differences using the same numerations of the previous paragraph.

4. Q computation: it is made through the init_Q helper function, but this time is very useful, since its informations will be used for applying the Distribute Iterative Soft Thresholding. A sidenote is that eps is a number in the range [0, 1] which correspond to the percentage for the actual ϵ in the Q matrix (using uniform weights). Specifically is the percentage between 0 and the maximum ϵ permitted, which is $\frac{1}{max(d_i)}$ where d_i is the in-degree of the i-th sensor.

5. DIST algorithm: during this cycle, the DIST algorithm is applied to each x of each sensor. In this code there are checks over the values of x. These were added when trying to apply the algorithm with more than one measurement per sensor or when the number of sensors is changed. The complete explanation is in the "Extensions" section. For each sensor is then computed the average state of the neighbors and the DIST_step function is called. When the difference of the 1-norm (or 2-norm) average of the vectors is below stopThreshold the algorithm stops. This is unfeasible in real-life, but it was implemented in this way in order to compare it with the DIST algorithm.

6. The big difference here is that x does not contain the full history of the x vector, but only the best estimate. Furhtermore, x_diff contains only the euclidean norm of x for each sensor for each time t. Finally, p_bar is computed as the average of the coordinates of all the best estimates.

Exactly as IST_step, DIST_step computes the gradient descent with respect to the average state of the neighbors and then applies the soft thresholing operator.

Helper functions files

We shall write a full list with the brief explanation of each file. Some custom notation is introduced.

- arrow - this file prints an arrow in a plot. It can be found on the matlab website
- cell2pos - converts an index of the room (which wi call cell) to a two coordinate position. It is also possibile to plot the pos as a green square.
- deploy - contains a function for deploying the senors. They might be deployed in a grid fashion or randomly. If the deployment is random, then the algorithm computes the positions such that a spanning-tree exists and the sensors are at least min_radius meters apart. Default: min_radius = 1.
- feng - computes B if only A is given. Computes [B, z] if also y is given as a second parameter.
- generateLegend - makes a custom legend for the active figure.

- get_in_degree - computes a vector in which each component is the in-degree of each row of the graph. The matrix Q has to be an adjacenct matrix for the graph in question.
- getPath - generates predefined path given an input number. The path is a matrix that has 2 columns (x and y) and as many rows as points in the path.
- init_A - coincide with the training step. It computes A given the sensors' positions
- init_Q - computes a suitable wheighted matrix for the graph, given the sensors' positions. It uses uniform weights (to garantee consensus) and checks whether the given ϵ belongs to the permitted interval.
- plotAgents - plots the agents inside a room given the sensors' positions. It can display the radius of the sensors, can display the sensors with random colors and can display the sensors using their ID.
- plotPath - uses arrow.m for plotting a path in the room
- pos2cell - converts a position (x,y) to the corresponding cell index.
- showRoom - creates a new figure and display the number of each cell.

Restults and Analysis

Experiment A

The objective of this experiment is to have a performance estimation of DIST and ODIST algorithms.

DIST algorithm

Evaluation Method

We evaluate DIST algorithm on 100 examples. For each example:

- If the estimate of the target is correct, we increase a counter (n_success) to compute, at the end, the succes rate.
- If the estimate of the target is not correct, we compute the distance between the target position and the estimate position and we increase a counter that counts the number of failed estimates.

Examples generation

We generate 100 examples in this way: - The first half, with a random position of the target and a grid deployment of sensors (all sensors are at the same distace from each others). - The second half, with a random position of the target, and a random deployment of sensors.

Parameter Settings

- Tmax = 1e5
- StopThreshold=1e-6
- x(0) = 0

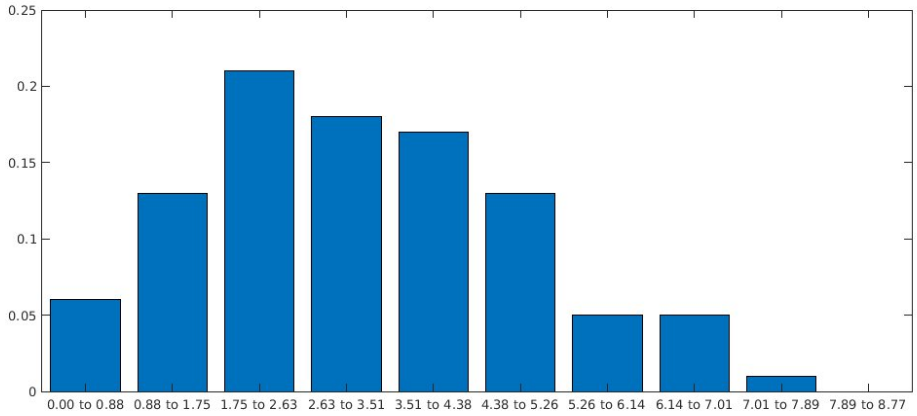
Results

With the previously shown settings we computed the successful rate and found out that none of the 100 runs could correctly localize the target. This result is peculiar, especially because the O-DIST performed so well in comparison. We think that somehow the initial conditions heavily influenced its behaviour. We didn't have the chance to investigate our hypotesis, though. We computed the average distance between the estimate and the real target using the following formula, obtaining the followig value:

$$\overline{d} = \frac{1}{n} \sum_{i=1}^{n_{example}} \| \overline{x}_{(i)} - \tilde{x}_{(i)} \|_2 = 4.3247$$

where: - $\overline{x}_{(i)}$ is the real target position - $\tilde{x}_{(i)}$ is the estimated position of the target

Furthermore we plotted the frequency histogram of the error using 10 range buckets.



The variance of this error is 2.9336

ODIST algorithm

Evaluation Method

We evaluated the ODIST algorithm on 4 different path. For each path we ran ODIST algorithm with 2 different deployment of sensors:

- Random deployment
- Grid deployment

At the end we computed the distance between the estimate position and true position for each target in the path, and also we computes the cumulative distance between the true path and the estimate path for each sensor deployment.

Path generation:

We generated 4 different path:

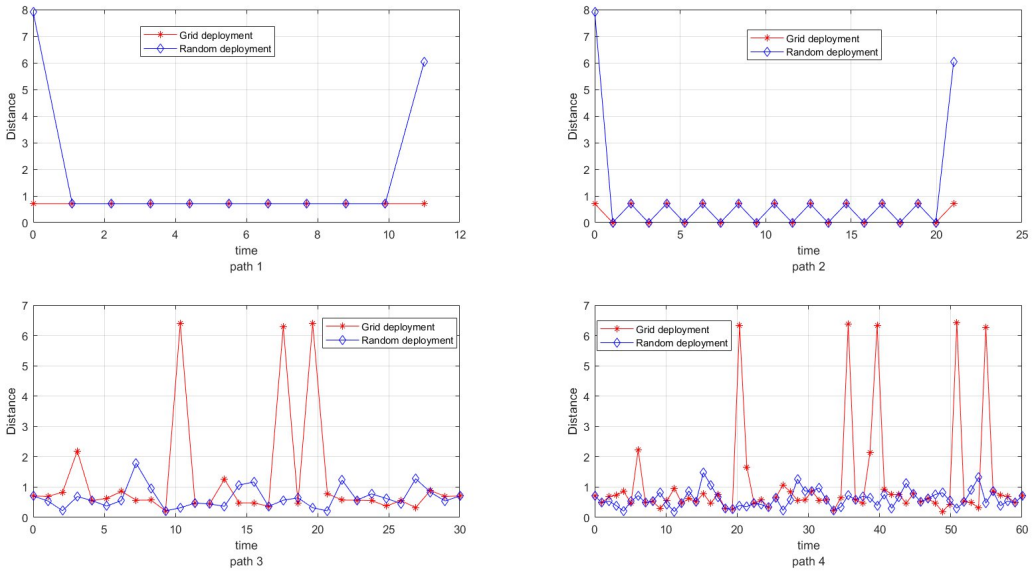
- Linear path composed by 10 points.
- Linear path composed by 20 points.
- Circular path composed by 30 points.
- Circular path composed by 60 points.

Parameter Settings

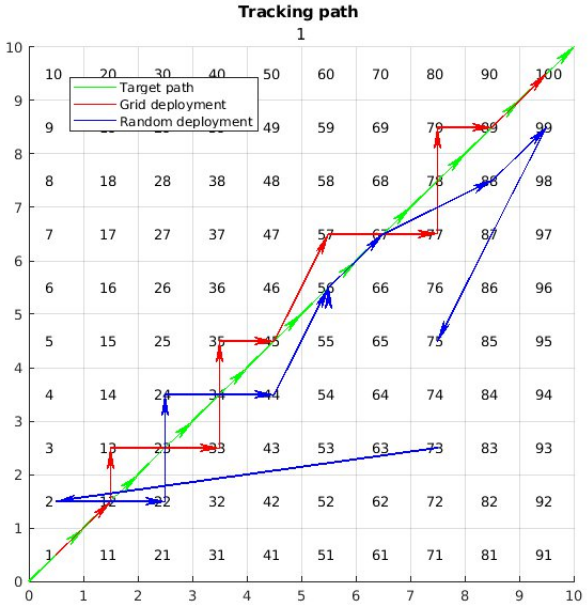
- Tmax = 1e5
- StopThreshold=1e-6
- x(0) = 0

Results

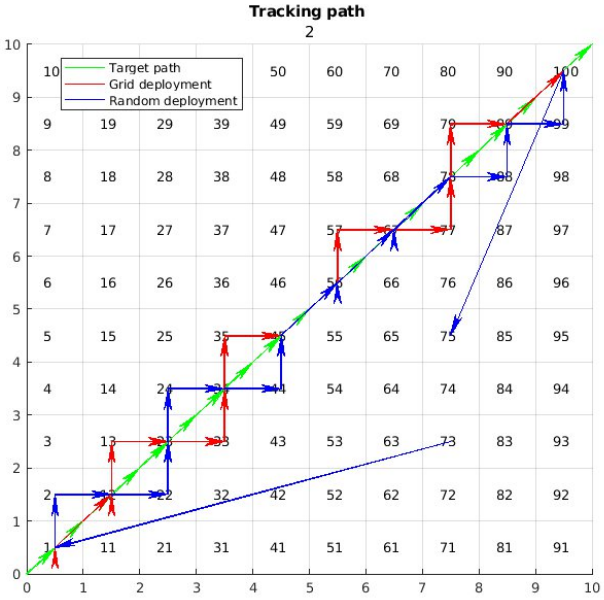
With the previously shown settings, we computed the cumulative distance for each point of the path. We saw that, in general, the algorithm performed quite well. The figure below shows the error for each point of the path.

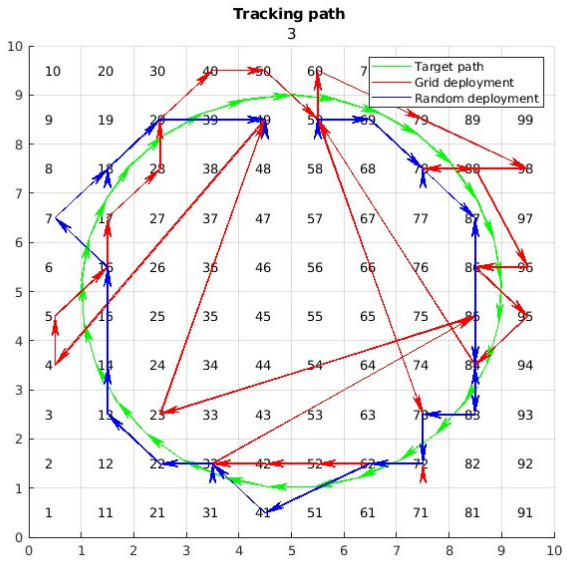


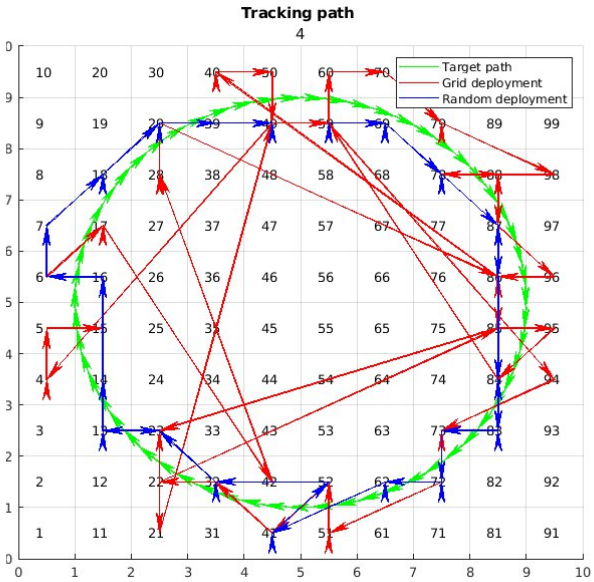
We noticed that the random sensors deployment performed much better than the grid deployment. The most likely reason for this behaviour is due to the intrinsic simmetries of this particular type of deployment, indeed the target position splits the sensors in such a way that at least two sensors measure the same value (without taking noises into account) with the exceptions of some extreme cases. The following figures show the 4 sample paths



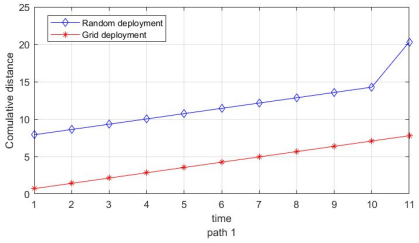
.

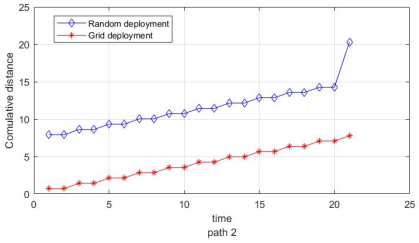


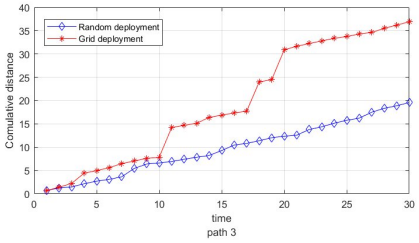


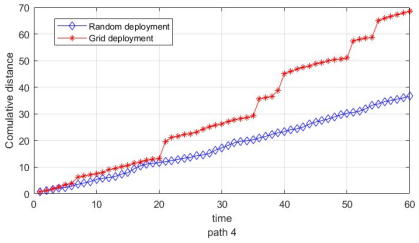


Finally, the next figure shows te comulative distances of this path.









Experiment B

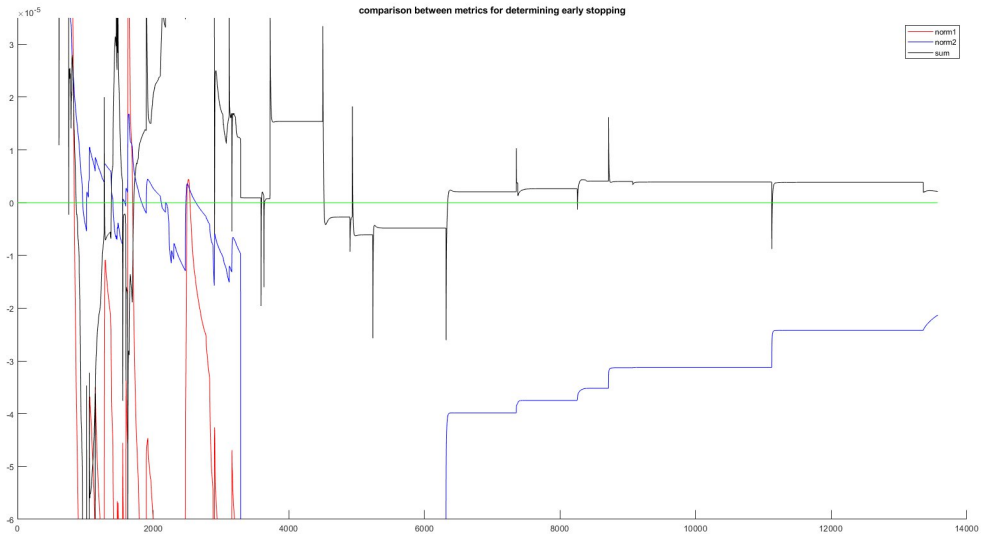
Experiment B is about comparing IST-DIST performances.

How to choose the right norm to compare the two algorithms?

In order to set a stop criterion we need to measure the "distance" between predictions taken at *t* and *t-1*, hence the question is which definition of distance to use?

We took into consideration l1 norm, l2 norm and the sum of all the states, so basically we want to know which norm has the same order of magnitude for both the algorithms.

The following figure shows how this three norms performed.



As notable from this figure, the most similar metrics we can use is the sum, but we can't use it because it goes under 0 (indeed it's not an actual norm). That's a great indication of "overshoot", meaning that some components of the state go over the minimum of the function and try to get back. Due to this considerations we decided to use the l1 norm.

How many iterations are necessary on average?

	IST	DIST
Avg absolute error	4.345	4.442
Error variance	3.089	3.390
Success rate	0.000	0.000
Accuracy	0.340	0.310
Avg. convergence time	1.728e+04	3.640e+03

These are the outcomes obtained after testing the algorithms 100 times moreover we decided to take into consideration other key performance indicators; the following explains the indicators that might be more ambiguous:

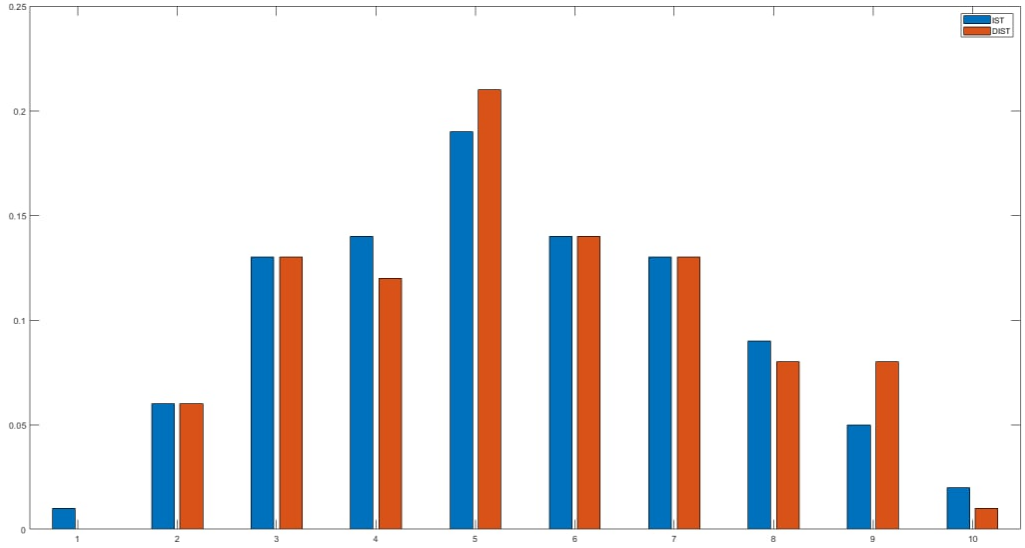
- avg absolute error is the euclidean distance between the real target position and the estimated one.

- success rate is the ratio between the number of correct estimations and the total experiment number. In this case the definition of success is if the algorithms predicts the cell in which the target really is.
- accuracy is the ratio between the number of correct estimations and the total experiment number. For success we mean all the predictions that have a distance from the target under a certain threshold that we fixed at 3.5. We choose this value based on the variance and average.

Probability mass distribution

Aside from the requested metrics, we decided to plot also the error probability distribution. We created this plot first dividing the x-axis into 10 homogeneous intervals inside this range $[\mu-\sigma, \mu+\sigma]$ (where μ and σ are respectively the mean and the variance of the error), the we counted how many times the algorithms made an error inside each interval.

In this case we took the absolute error, meaning the euclidean distance between the target and the estimate.



As we can see the error probability distribution for each algorithm is gaussian and they are really similar meaning that the two algorithms perform almost the same.

Experiment C

The objective of this experiment is to verify the relationship between the essential spectral radius of the matrix Q and the convergence time of the system. The setup for this experiment is the following:

- random sensor deployment,
- fixed target at the center of the room,
- 30 rounds for each sensor deployment with varying essential spectral radius manipulating ϵ between 20% and 80% the maximum permitted value,
- maximum time set to 1e5 cycles,
- early stopping if the 2-norm of the $\overline{x}$ gets below 1e-6, where $\overline{x}$ represents the average value of the state x over each sensor ($\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$).

A total of 60 different sensor deployment were run. For each run, a total of 30 points were evaluated, each point corresponding to a different value of the essential spectral radius.

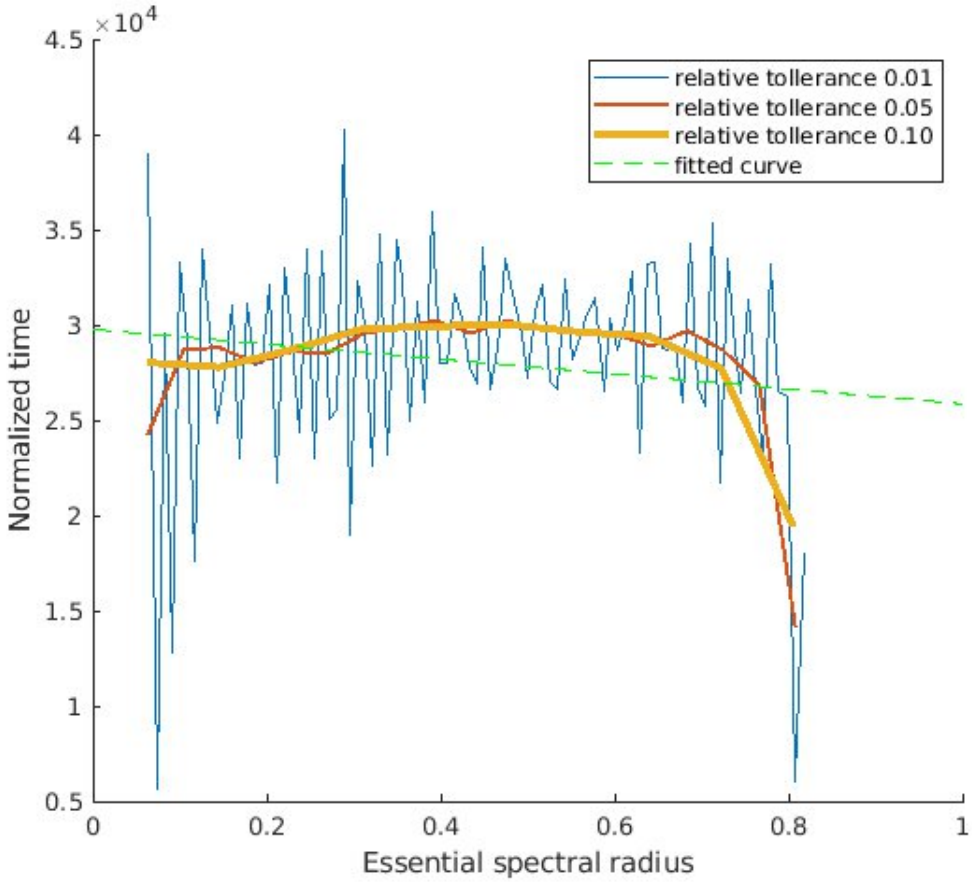
Results

Analyzing the result of this experiment wasn't simple. Each run produced a completely different outcome which wasn't simple to generalize. To address this problem, as explained above, we decided to run 60 different experiments each with 30 different eigenvalues.

The nex step was to analyze the obtained data as a whole and since there were no two equal eigenvalues we adopted the following procedure:

- firstly, we computed a *class of eignevalues*, i.e. we grouped them using a similarity criterion. In order to do that, we used the matlab function uniquetol. Two values u and v, based on the matlab reference, are whithin tolerance if abs(u-v) <= tolerance * max(vector) where vector correspond to the values we want to group toghether.
- The above step generated some *buckets*, bounds we could use to group times.
- The second step was to group the times based on the same criterion, so all the time values corresponding to an essential spectral radius class were averaged togheter. For performing this step, given the variety of the convergence times (between a minimum of 3000 cycles up to 10000) it was necessary to normlize them in such a way that for each specific sensor deployment the times would range between 0 and 1.

This process was repeated three times with three different relative tolerance values. In the figure the three results are shown.



Furthermore, the green line represents the linear regression of the curve in yellow, which is the one computed with a relative tolerance of 10e-2.

The linear regressions have a negative angular coefficient, even though small (-0.044), which describes the tendancy of the algorithm to faster converge given a higher essential spectral radius.

Extensions

In this section we shall present some extra implementations we made. No formal experiment was run, but a couple of interesting observations came out.

More measurements (m>1)

The first thing we tried to do was to make the sensors measure more than one time. A good way to generate a better dictionary might consist in mooving the target around inside the cell for the sensors to pick up a slightly different measurement for the specific cell. This was not done since in the simulation there was added noise. The noise guarantee different measures for each time instant. An argument one might do is as following: if the noise is too small, there is no added information in doing a second measurement, or at least is very small, so moving the target around inside the cell might increase the gathered informations needed for the target localization.

More targets (k > 1)

Another interesting fact was how the value of k changed when adding sensors or measurements. We noted that k could reach dramatically high values. This only happened,

though, applying Feng's theorem which is a great tool in this context. The only problem was that, at some point, if we only added measurements, some informations were lost applying the theorem. That's due to the fact that the matrix A stopped to being full matrix when m was too big.

More sensors (n>25)

For using more sensors, as explained in the previous paragraphs, we needed to disable Feng's theorem. This way we were able to localize the target in a pretty much perfect way. With 100 sensors, each performing 20 measurements, the results were stunning. Surely, it is not that practical to have one sensor per cell in real life. Another important point was the sensibility of the algorithm to the τ parameter. In order for the system to not diverge or go to zero, we needed to carefully change its value until we found the sweet spot for the given configuration. The next table shows three parameter for different configurations.

Configuration	τ
Using Feng, n = 25, 1<=m<=4	0.7
Not using Feng, 25<=n<=50, m=20	0.378e-6
Not using Feng, n=100, m=25	1e-7