The MUSIC Flow Graph - Theory, Demo, Profiling

Fluerătoru Laura

1 Introduction

This report outlines the theory on which the MUSIC algorithm is founded in Section §**??**, summarizes its main steps in Section §**??**, then Section §**??** explains how they are mapped on the flow graph for the MUSIC algorithm implemented by Ettus Research [**?**] that is used as a reference for the current project. Section §**??** shows the results of profiling the aforementioned C++ flow graph and its actual code can be found in Section §**??**.

2 Teoria din spatele algoritmului MUSIC

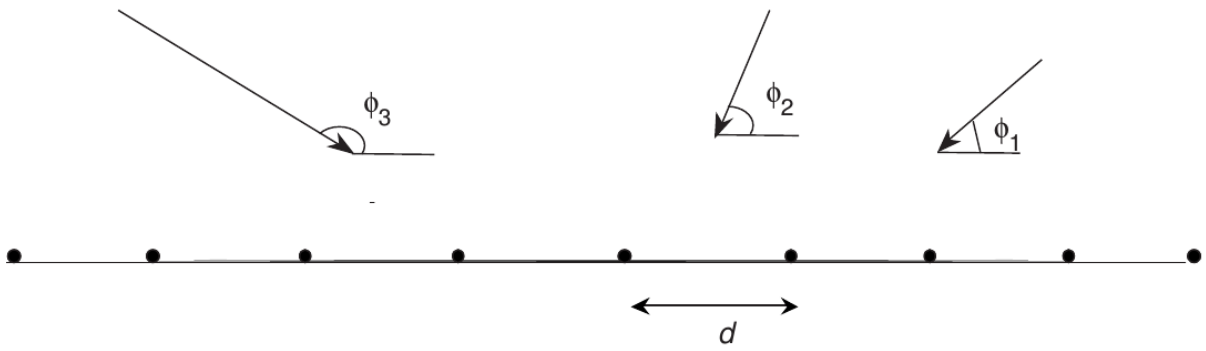

Figura 1: TODO: my own image in PS

În analiza algoritmului MUSIC presupunem că plecăm de la un șir liniar de M antene la care ajung D semnale $s_j(t), i = \overline{1, D}$ și se dorește estimarea unghiului de incidență θ_j măsurat față de axa x în sens trigonometric sub care ajunge fiecare semnal la șirul de antene. Presupunem că mediul de propagare nu afectează semnificativ semnalele când acestea se propagă de la un element din șir la altul, deci semnalul care ajunge la o antenă diferă de cel care ajunge la o altă antenă doar printr-o întârziere τ.

Considerăm că prima antenă se află în originea sistemului, la locația (0, 0) și exprimăm întarzierea semnalului de la celelalte elemente din șir relativ la semnalul care ajunge la elementul de referință. Astfel, pentru un șir liniar, semnalul j care ajunge la elementul $i = 2$ parcurge o distanță mai lungă cu $d\,cos\theta_j$ față de primul element și, în cazul general, putem scrie întârzierea τ_i la elementul i ca

$$\tau_i = \beta(i-1)d\,cos\theta_j \tag{1}$$

unde $\beta = \frac{2\pi}{\lambda}$ este factorul de defazaj.

Aici, am presupus că defazajul semnalului depinde doar de locațiile distincte ale elementelor șirului de antene. În realitate, antenele vor avea și ele un răspuns dependent de directivitatea lor și de frecvență, care poate fi modelat ca un câștig g_i. Obținem, astfel, vectorul director (*steering vector*) pentru un anumit unghi de incidență θ_j și frecvență ω:

$$\boldsymbol{a}(\omega, \theta_j) = \begin{bmatrix} g_1(\omega, \theta_j)e^{\tau_1(\theta_j)} \\ ... \\ g_M(\omega, \theta_j)e^{\tau_M(\theta_j)} \end{bmatrix} \tag{2}$$

Prin urmare, vectorul director este dependent de răspunsul individual al fiecărui element din șirul de antene, de geometria șirului, de frecvența semnalului și de unghiul de incidență al acestuia. Matricea care se formează din vectorii coloană directori pentru toate unghiurile de incidență și toate frecvențele se numește matricea colectoare a șirului (*array manifold matrix*).

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{a}(\omega, \theta_1) & ... & \boldsymbol{a}(\omega, \theta_D) \end{bmatrix} \tag{3}$$

Dacă banda semnalului este suficient de îngustă, se poate considera, cu aproximație, că vectorul director este independent de frecvență și că depinde doar de unghiul de incidență. Mai mult, dacă presupunem că elementele șirului sunt izotrope, putem elimina dependența vectorului director de câștigul $g_i, i = \overline{1, M}$. Vom continua analiza având în vedere aceste presupuneri.

Putem scrie următoarea relație pentru semnalele de la intrarea fiecărei antene din șir:

$$\boldsymbol{x}(t) = \begin{bmatrix} \boldsymbol{a}(\theta_1) & \boldsymbol{a}(\theta_2) & ... & \boldsymbol{a}(\theta_D) \end{bmatrix} \begin{bmatrix} s_1(t) \\ s_2(t) \\ ... \\ s_D(t) \end{bmatrix} + \begin{bmatrix} n_1(t) \\ n_2(t) \\ ... \\ n_D(t) \end{bmatrix} \tag{4}$$

$$\boldsymbol{x}(t) = \boldsymbol{A}\boldsymbol{s}(t) + \boldsymbol{n}(t) \tag{5}$$

Algoritmul MUSIC (MUltiple SIgnal Classification) face parte dintr-o clasă mai mare de algoritmi care se bazează pe metoda subspațiilor, care ia în considerare și zgomotul dintr-un sistem. Algoritmul oferă, în primul rând, informații despre numărul de semnale care ajung la un șir de antene și unghiul de incidență al acestora. Este un algoritm cu rezoluție mare, ceea ce înseamnă că poate distinge mai ușor decât alți algoritmi două semnale care vin din direcții foarte apropiate, dar are nevoie de o calibrare foarte precisă a șirului de antene. Calibrarea constă în obținerea matricei colectoare a șirului de antene; în practică, acest lucru se realizează măsurând răspunsurilor unor surse punctiforme ale șirului la diverse unghiuri și frecvențe. Pentru explicarea fundamentului matematic din spatele algoritmului MUSIC s-a folosit ca referință lucrarea [**?**], în care este tratat pe larg subiectul estimării unghiurilor de incidență folosind șiruri de antene.

În ecuația (**??**), vectorul $\boldsymbol{x}$ al semnalelor de la intrarea antenelor din șir și vectorii directori $\boldsymbol{a}(\theta_j), j = \overline{1, D}$ pot fi priviți ca vectori într-un spațiu cu M dimensiuni, ceea ce înseamnă că $\boldsymbol{x}$ poate fi scris ca o combinație liniară între vectorii directori, unde $s_j, j = \overline{1, D}$ sunt coeficienții combinațiilor.

Se poate calcula matricea de covarianță a intrării

$$\boldsymbol{R}_{xx} = E[\boldsymbol{x}\boldsymbol{x}^H] = \boldsymbol{A}E[\boldsymbol{s}\boldsymbol{s}^H]\boldsymbol{A}^H + E[\boldsymbol{n}\boldsymbol{n}^H] \tag{6}$$

$$\boldsymbol{R}_{xx} = \boldsymbol{A}\boldsymbol{R}_{ss}\boldsymbol{A}^H + \sigma_{zg}^2\boldsymbol{I} \tag{7}$$

S-a notat $\boldsymbol{R}_{ss} = E[\boldsymbol{s}\boldsymbol{s}^H]$ matricea de corelație a semnalului s. Se observă două proprietăți importante:

- Vectorii directori sunt liniar independenți, ceea ce înseamnă că matricea $\boldsymbol{A}$ este de rang maxim.
- $\boldsymbol{R}_{ss}$ este o matrice nesingulară dacă semnalele incidente sunt cel mult parțial necorelate. Dacă ar fi corelate, atunci cel puțin una dintre liniile/coloanele sale ar putea fi scrisă ca o combinație liniară a altor linii/coloane, ceea ce ar însemna că matricea ar avea determinantul egal cu 0, deci ar fi singulară și nu am mai putea folosi algoritmul.

Din aceste două proprietăți rezultă că, dacă $M < D$, atunci matricea $\boldsymbol{A}\boldsymbol{R}_{ss}\boldsymbol{A}^H$ este pozitiv semidefinită, având rangul D. Din această proprietate, se poate demonstra faptul că $M - D$ dintre valorile proprii ale matricei trebuie să fie nule. Folosind relația (**??**), reiese că atunci când $M - D$ dintre valorile proprii ale matricei $\boldsymbol{A}\boldsymbol{R}_{ss}\boldsymbol{A}^H$ sunt nule, cele mai mici valori proprii ale matricei $\boldsymbol{R}_{xx}$ vor fi egale cu puterea zgomotului σ_{zg}^2. Dacă notăm $\lambda_i, i = \overline{1, M}$ valorile proprii ale matricei $\boldsymbol{R}_{xx}$, atunci

$$\lambda_{D+1} = \lambda_{D+2} = ... = \lambda_M = \lambda_{min} = \sigma_{zg}^2 \tag{8}$$

În realitate, însă, nu se va îndeplini egalitatea, deoarece folosim pentru estimare doar un număr finit de eșantioane, dar valorile vor fi, într-adevăr, foarte apropiate. Dacă notăm K multiplicitatea celei mai mici valori proprii a matricei $\boldsymbol{R}_{xx}$, atunci, știind că $M = D + K$, putem estima numărul semnalelor care ajung la șirul de antene

$$\hat{D} = M - K \tag{9}$$

Din definițiile valorilor proprii și a vectorilor proprii [**?**], vectorii proprii $\boldsymbol{v}_i, i = \overline{1, M}$ corespunzători celor mai mici valori proprii trebuie să satisfacă egalitatea

$$\boldsymbol{R}_{xx}\boldsymbol{v}_i = \sigma_{zg}^2\boldsymbol{v}_i, \quad i = \overline{D+1, M} \tag{10}$$

Folosind ecuația (**??**), înseamnă că

$$\boldsymbol{A}\boldsymbol{R}_{ss}\boldsymbol{A}^H\boldsymbol{v}_i = 0, \quad i = \overline{D+1, M} \tag{11}$$

Știind că $\boldsymbol{A}$ este de rang maxim și că $\boldsymbol{R}_{ss}$ este nesingulară, atunci

$$\boldsymbol{A}^H\boldsymbol{v}_i = 0, \quad i = \overline{D+1, M} \tag{12}$$

ceea ce este echivalent cu a spune că vectorii coloană ai matricei $\boldsymbol{A}$, adică vectorii directori, sunt perpendiculari pe vectorii proprii ai matricei $\boldsymbol{R}_{xx}$.

$$\begin{bmatrix}\boldsymbol{a}(\theta_1) & \boldsymbol{a}(\theta_2) & ... & \boldsymbol{a}(\theta_D)\end{bmatrix} \perp \begin{bmatrix}\boldsymbol{v}_{D+1} & \boldsymbol{v}_{D+2} & ... & \boldsymbol{v}_M\end{bmatrix} \tag{13}$$

Recapitulând, avem două subspații ortogonale: cel al semnalelor și cel al zgomotului. Vectorii directori ai șirului de antene aparțin subspațiului semnalelor și, așadar, sunt perpendiculari pe subspațiul zgomotului, iar vectorii proprii ai matricei de covarianță $\boldsymbol{R}_{xx}$ pot aparține oricăruia dintre cele două subspații. Prin urmare, putem să căutăm printre vectorii directori pe aceia care sunt ortogonali pe subspațiul zgomotului, în care se vor afla o parte din vectorii proprii ai matricei de covarianță, și să determinăm unghiurile de incidență θ_j.

Se calculează spectrul MUSIC folosind una dintre următoarele două forumle:

$$P_{MUSIC}(\theta) = \frac{\boldsymbol{a}^H(\theta)\boldsymbol{a}(\theta)}{\boldsymbol{a}^H(\theta)\boldsymbol{V}_N\boldsymbol{V}_N^H\boldsymbol{a}(\theta)} \tag{14}$$

$$P_{MUSIC}(\theta) = \frac{1}{\boldsymbol{a}^H(\theta)\boldsymbol{V}_N\boldsymbol{V}_N^H\boldsymbol{a}(\theta)} \tag{15}$$

$$\boldsymbol{V}_N = \begin{bmatrix}v_{D+1}, & ..., & v_M\end{bmatrix} \tag{16}$$

și, în cazul în care vectorii directori sunt perpendiculari pe vectorii proprii, numitorul va fi minim, ceea ce va conduce la apariția unor vârfuri în spectru. Cunoaștem numărul de semnale estimat $\hat{D}$, deci cele $\hat{D}$ vârfuri din spectru corespund unghiurilor de incidență căutate.

3 Outline of the MUSIC Algorithm

The theory behind the MUSIC algorithm discussed in section §?? can be summarized in some elementary steps. We remind that M is the number of elements in the antenna array and D is the number of incoming signals at each element.

Step 1

Knowing the signal x_i arriving at the element number i of the antenna array, the input covariance matrix can be computed:

$$\boldsymbol{R}_{xx} = E[\boldsymbol{x}\boldsymbol{x}^H] \tag{17}$$

$$x = \begin{bmatrix} x_1 & x_2 & ... & x_M \end{bmatrix} \tag{18}$$

Step 2

Estimate the number of signals arriving at each element of the array. In order to do this, we need to compute the eigenvalues of R_{xx} and, from the multiplicity K of the smallest eigenvalue, we estimate the number of signals as follows:

$$\hat{D} = M - K \tag{19}$$

Step 3

Compute the MUSIC spatial spectrum using

$$P_{MUSIC}(\theta) = \frac{\boldsymbol{a}^H(\theta)\boldsymbol{a}(\theta)}{\boldsymbol{a}^H(\theta)\boldsymbol{V}_N\boldsymbol{V}_N^H\boldsymbol{a}(\theta)} \tag{20}$$

$$\boldsymbol{V}_N = \begin{bmatrix} v_{D+1}, & ..., & v_M \end{bmatrix} \tag{21}$$

Step 4

Find the peaks of the estimated MUSIC spectrum which give us direction of arrival for the $\hat{D}$ signals.

4 The Demo Chain

In order to assess the performance of the MUSIC algorithm, an implementation created by Ettus Research [?] has been used. They provide an application which proves the phase synchronization capability of their TwinRX daughtercards, inside the GNU Radio framework. Figure ?? shows the flow graph created with the GNU Radio Companion.

To explain the functionality of the graph, we need to define a set of parameteres used in various block. The names of the parameters do not necessarily reflect the actual names of the variables used in the code, but make it easier to identify the parameters of interest. Also, some parameters are also denoted by a shorter name, to make notations easier where it applies.

- `sample_rate` The rate at which the input signal is sampled
- `tone_freq_i` The frequency of the i^{th} input signal

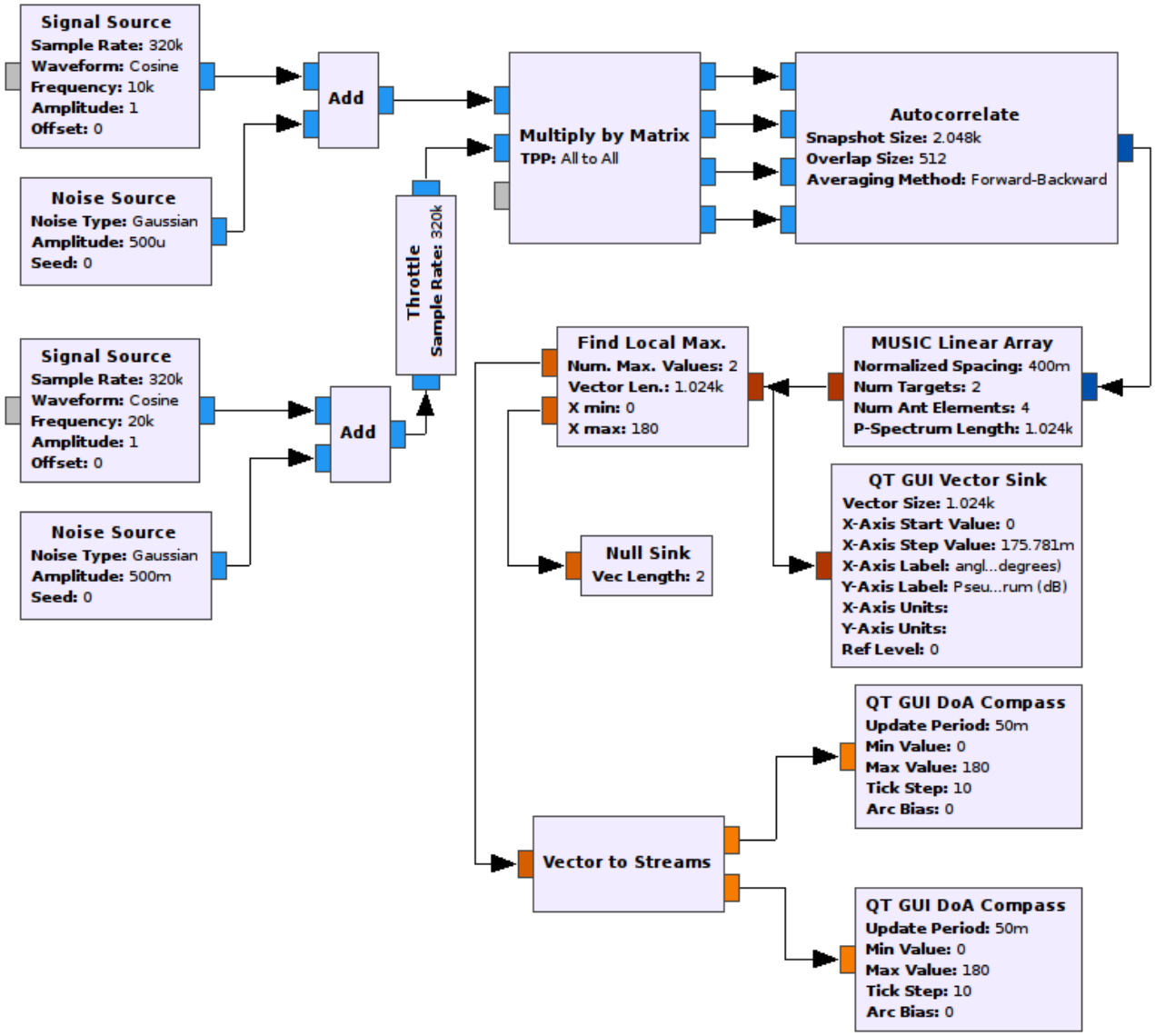

Figura 2: The flow graph used for the MUSIC algorithm

- `norm_spacing` The normalised distance between the elements of the antenna array
- `num_targets` The number of input signals, also denoted by N
- `num_array_elements` The number of elements in the antenna array, also denoted by M
- `p_spectrum_length` The length of the computed MUSIC spectrum
- `snapshot_size` The number of samples used in computing the autocorrelation, also denoted by K
- `overlap_size` The number of samples that overlap when computing successive values of the autocorrelation

4.1 The input data

This configuration uses two signal sources ($D = 2$) which generate cosines of frequencies 10 kHz and 20 kHz, over which we add Gaussian noise, and an antenna array with four elements

($M = 4$). The **Throttle** block is used to limit the data throughput to the frequency of the input signal's frequency, as it would behave in the case of a real signal arriving at an antenna.

4.2 The Multiply by Matrix block

This block is used to simulate the way signals arrive at the antenna array, by multiplying the array manifold matrix with an array which consists of samples of the incoming signals. The general behaviour of the block is that if $\boldsymbol{A}$ is the matrix that is given as a parameter to the block, of size M by N, and $\boldsymbol{X}_N$ is a column array built from the N inputs given to the block, then the result of the multiplication is

$$\boldsymbol{Y}_M = \boldsymbol{A}\boldsymbol{X}_N, \tag{22}$$

where $\boldsymbol{Y}_M$ is an column array constructed from the outputs of the block. Thus, the block should have a number of N inputs and M outputs.

The array manifold matrix takes into account the angles of arrival and the normalised distance between the elements of the array. The normalised spacing is the distance between the antenna elements divided by the wavelength of the carrier. According to [**?**], the normalised spacing should be at most half of the center wavelength of the signal, otherwise aliasing could interfere with the angle resolution performance of the MUSIC algorithm.

In our case, the array manifold matrix is:

$$\begin{bmatrix} \boldsymbol{a}(\theta_1) & \boldsymbol{a}(\theta_2) \end{bmatrix}$$

where $\boldsymbol{a}(\theta_i)$ is the steering vector corresponding to the angle of arrival of the i^{th} signal. The outputs of the **Multiply by Matrix** block represent the sum of the signals that arrive at different angles at each element of the antenna array.

4.3 The Autocorrelate block

The following block, **Autocorrelate**, corresponds to the step §1 of the MUSIC algorithm, although it actually computes an estimate of the autocorrelation of the signal in the form of a sample correlation matrix. This method implies collecting a sample over a snapshot time of the signal that arrives at each element of the antenna array. Thus, an M by 1 array is formed, denoted $x(k)$, and the sample correlation matrix is computed as follows:

$$C_x = \frac{1}{K}\sum_{k=1}^{K} \boldsymbol{x}(k)\boldsymbol{x}^H(k). \tag{23}$$

Another method of computing the sample correlation matrix is collecting a number of samples over a snapshot period denoted K, forming the matrix $\boldsymbol{X_K}$ of dimensions N by K, which leads to the following formula:

$$C_x = \frac{1}{K}\boldsymbol{X_K}\boldsymbol{X_K}^H. \tag{24}$$

In [**?**] it has been suggested that an additional step consisting of a Forward-Backward Averaging of the sample correlation matrix will increase the performance of the DoA estimation, so the computation can be adjusted as such:

$$C_x = \frac{1}{2K}\boldsymbol{X_K}\boldsymbol{X_K}^H + \frac{1}{2K}\boldsymbol{J}\boldsymbol{X_K}^*\boldsymbol{X_K}^T\boldsymbol{J}, \tag{25}$$

where J is a reflection matrix (elements on the cross-diagonal are equal to 1 and the rest are 0).

The parameters of the **Autocorrelate** block are:

- Snapshot size, previously denoted as K, representing the number of input samples used in computing the correlation matrix for an output item
- Overlap size, the number of samples that overlap in the computation for an output item to the next
- Averaging method, which can be either Forward-Backward or the standard Forward method

4.4 The MUSIC Linear Array block

In this application, we assume that we know the number of targets, so we do not need a separate block to perform step §2. Therefore we proceed directly to the **MUSIC Linear Array** block, which computes the MUSIC spatial spectrum from the step §3.

In the constructor of the block, an array manifold vector is generated, which uses a span of the possible angles between 0 and 180 degrees, with a resolution of `1/d_spectrum_length`. The way the algorithm works, the closer an angle used in generating the array manifold vector is to the actual angle of arrival, the closer the MUSIC spectrum is to 0. In theory, when they coincide, the MUSIC spectrum will reach 0. Therefore, for each input item a number of `d_spectrum_length` values of the MUSIC spectrum have to be computed.

To compute the MUSIC spectrum, the block takes as input the result of the autocorellation, on which an EigenValue Decomposition (EVD) is performed, and then the desired spectrum is computed according to forumla (**??**). The block takes as parameters the number of targets, the number elements in the antenna array and the normalised distance between them, and also the spectrum length.

4.5 The Find Local Max block

In order for the application to find the precise values of the angles of arrival, a further block named **Find Local Max.** is needed, which solves the step §4 of the algorithm. The block outputs the coordinates (that is, the angle) at which the peaks in the spectrum are found and looks precisely for D such maximums. The block also gives information about the amplitude of the spectrum at the given locations, but since they are of no importance for this application, they are directed towards a **Null Sink**.

4.6 Visualising the results

We can either visualise the spectrum immediately after the **MUSIC Linear Array** block, such as in Figure **??**, and form a good idea about the estimated angle of arrival with the **QT GUI Vector Sink** block or, using the **QT GUI DoA Compass** widget, we can see the estimated angle of arrival for the two signals after we found the peaks in the spectrum. One example of such an output for one of the signals can be found in Figure **??**. Using two sliders,

we can change the angle of arrivals in real time and observe how the algorithm is adapting to the changes.

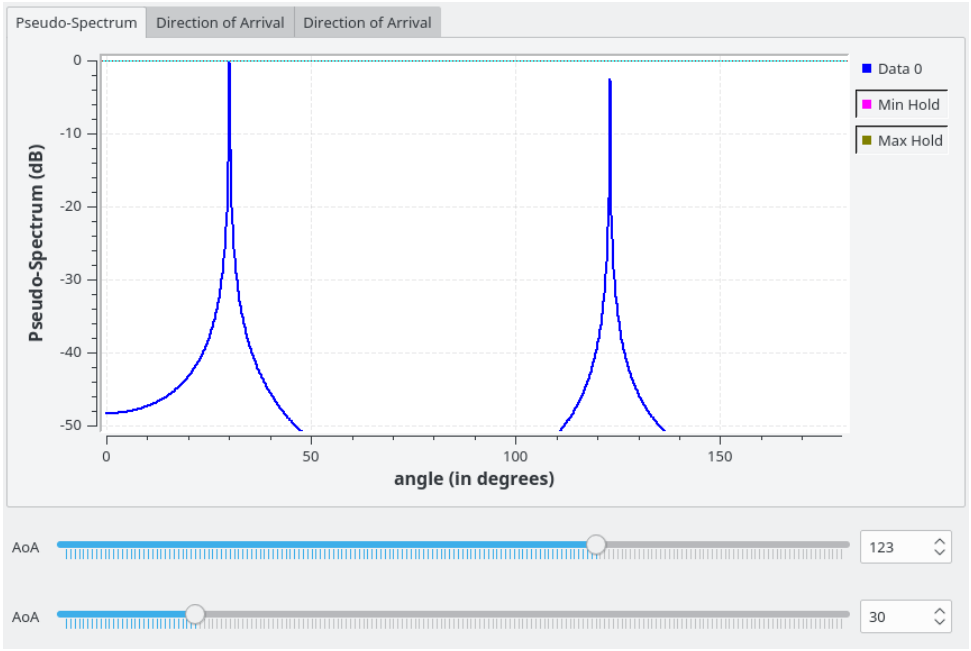

Figura 3: The MUSIC spectrum

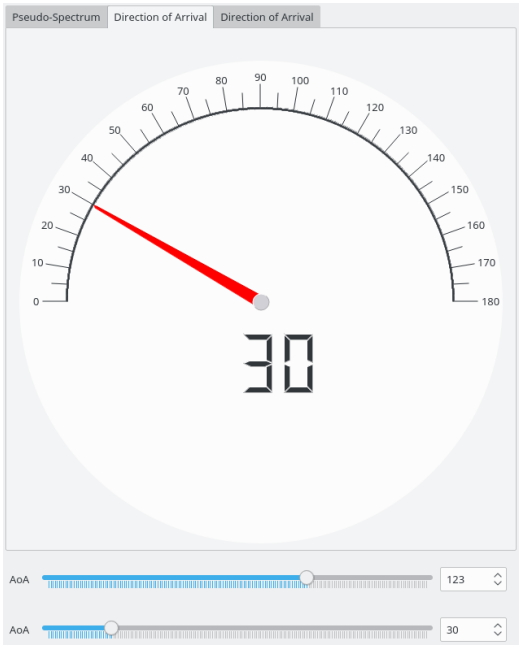

Figura 4: The estimated angle of arrival for one signal

5 The Profiling Chain

5.1 Profile Methodology

The GNU Radio Companion generates a Python script based on the flow graph created in the graphic interface. The Python script then uses SWIG [?], a tool that enables it to access the modules written in C++ and execute the flow graph. Because of this intermediary step, the

profiling performed on the script is not very conclusive, since functions needed in the wrapping process that are not directly related to the algorithm itself end up consuming much of the execution time. For this reason, the preferred solution was to create a C++ program that creates a flow graph similar to the initial one, which also eliminates the dependency on a visual interface.

In this test program all the Qt blocks previously used are eliminated and, where necessary, they are replaced with **Null Sink** blocks, which simply discard the data feeding them, so that there are no time consuming I/O operations. Additionally, the **Throttle** block is also discarded because we want to profile the program when it runs at full speed.

After the program starts the flow graph, it creates a new thread and, after it sleeps for 2 seconds (a period of time that can be adjusted), it calls for the flow graph to stop its execution, otherwise it would run indefinitely. The full code of the C++ flow graph and its Makefile can be found in Section **??**, in Listing **??** and Listing **??**, respectively.

For profiling the processing chain, both **perf** [**?**] and **callgrind** [**?**] have been used. **perf** is a powerful tool for measuring a large set of hardware events, which also displays assembly code for functions of interests, which can be helpful when wanting to target micro-optimizations. With **callgrind**, on the other hand, there is the option of using the visual interface **kcachegrind** [**?**] for an easier tracing of the call graph. In this case, the percentage of time spent in different functions is of interest, and both of the profiling tools gave similar results.

Using **perf**, we can obtain how many cycles are spent in each function by using the command:

```
perf record -e cycles ./run_MUSIC_profile
```

If we want to view the call graph, we can instead use:

```
perf record --call-graph lbr ./run_MUSIC_profile
```

Profiling the code with **callgrind** can be done with the following command:

```
valgrind --tool=callgrind ./run_MUSIC_profile
```

Since the results with both of the profilers are similar, only the ones obtained with **perf** will be outlined. Of particular interest is the case when the maximum level of optimization is being used, but since the call succession cannot be observed very well in this case, another analysis with the lowest level of optimization and debug symbols has been performed. The profiling flow graph uses functions from the *gr-doa* library of Ettus Research, as well as from *Armadillo* [**?**], a C++ library for linear algebra, which was compiled from sources, as the latest version needed cannot be found yet in the official repositories. The processing blocks rely internally heavily on functions implemented in *Armadillo* which may end up being computing intensive, so when compiling the profiling chain with level 0 of optimization, we have to make sure that both of the libraries are compiled in the same way in order to see the hot spots of the processing chain.

5.2 Profile results

Using the maximum level of optimization, the most important profiling results (for the functions which consume more than 2% of the execution time) are presented in Table **??**.

According to the documentation of the tool, *Overhead* is the percentage of time spent in a particular function, *Command* is the name of the task that can be read via `/proc/<pid>/comm`,

Overhead	Command	Symbol
49,14%	find_local_max6	arma::glue_hist::apply_noalias<unsigned int>
13,73%	autocorrelate4	cgemm_
10,51%	MUSIC_lin_array	cgemm_
3,63%	multiply_matrix	__mulsc3
3,56%	MUSIC_lin_array	cgemv_
3,06%	MUSIC_lin_array	gr::doa::MUSIC_lin_array_impl::work
2,33%	MUSIC_lin_array	__logf_finite
2,08%	multiply_matrix	gr::blocks::multiply_matrix_cc_impl::work

Tabela 1: Profiling Results With -O3

and *Symbol* is the name of the function executed at the moment of sampling.

Table **??** presents the output produced by `perf report` when the libraries and the project are compiled with the lowest level of optimization.

Overhead	Command	Symbol
49,87%	find_local_max6	arma::glue_hist::apply_noalias<unsigned int>
6,36%	autocorrelate4	cgemm_
4,81%	autocorrelate4	std::conj<float>
4,66%	MUSIC_lin_array	std::complex<float>::complex
3,38%	MUSIC_lin_array	cgemm_
2,46%	autocorrelate	arma::eop_core<arma::eop_conj>::apply <arma::Mat<std::complex<float> >, arma::Mat<std::complex<float> > >
1,82%	MUSIC_lin_array	arma::Mat<std::complex<float> >:: init_warm
1,57%	autocorrelate4	arma::op_strans::apply_mat_noalias <std::complex<float>, arma::Mat<std::complex<float> > >
1,54%	multiply_matrix	__mulsc3
1,49%	MUSIC_lin_array	cgemv_

Tabela 2: Profiling Results With -O0

With further investigation, it is found that the most time consuming function is `arma::glue_hist::apply_noalias<unsigned int>`, a function from the *Armadillo library* which is called from the function `find_local_max`, responsible for finding the peaks in the MUSIC spectrum. More precisely, the function is called from the `hist` functions which is used in Listing **??**.

```
105 // finding all peak locations by computing
106 // second-order difference of
107 // sign of first-order difference
108 uvec indx2 = find(diff(sign_fod_in_vec) == -2)+1;
109
110 uvec indx3;
111 uvec indx3_unique;
112 uvec hist_indx3;
113 uvec all_pk_indxs;
114 if (!indx2.is_empty())
115 {
116     // finding set-intersection between indx1 and indx2
117     // in order to identify local peaks
118     indx3 = join_vert(indx1, indx2);
119     indx3_unique = unique(indx3);
120     hist_indx3 = hist(indx3, indx3_unique);
121
122     all_pk_indxs = indx1(find(hist_indx3 == 2));
123 }
```

Listing 1: Computationally Intensive Part in find_local_max

TODO: explain what that part isn't necessary in most of the cases and which are those

By eliminating that processing overhead where it is not necessary, we obtain the new profiling results in Table **??**, where only the functions that make up more than 5% of the execution time are presented.

Overhead	Command	Symbol
12,73%	autocorrelate8	cgemm_
11,50%	MUSIC_lin_array	cgemm_
7,93%	noise_source_c3	__ieee754_log_avx
7,89%	noise_source_c4	__ieee754_log_avx
6,98%	noise_source_c3	gr::random::ran1
6,42%	noise_source_c4	gr::random::gasdev
6,09%	noise_source_c4	gr::random::ran1
5,32%	noise_source_c3	gr::random::gasdev
5,32%	MUSIC_lin_array	gr::doa::MUSIC_lin_array_impl::work

Tabela 3: Profiling Results After Corrections

Out of them, the ones related to the noise source are not of interest, since they are relevant only for the simulation process, and we will therefore focus only on the first two results.

Using the profiling tools, we find out that the hotspot in the **MUSIC Linear Array** block is the computation of the MUSIC spectrum using (**??**), which can be found in Listing **??**, on line 136.

```
for (int ii = 0; ii < d_pspectrum_len; ii++)
{
  Q_temp = as_scalar(d_vii_matrix_trans.row(ii) * U_N_sq * d_vii_matrix.col(ii))
      ;
  out_vec(ii) = 1.0 / Q_temp.real();
}
```

Listing 2: Computationally Intensive Part in the MUSIC Linear Array block

As far as the autocorrelation is concerned, the computationally intensive part is done in the first part of the calculation of the autocorrelation, corresponding to formula (**??**). Listing **??** depicts the part of the code that performs this specific function. The `d_input_matrix` is already stored in a transposed form, which explains the differences between the formula and the code.

```
out_matrix =
    (1.0 / d_snapshot_size) * d_input_matrix.st() * conj(d_input_matrix);
```

Listing 3: Computationally Intensive Part in autocorrelate

5.2.1 Profile results on the Xilinx ZedBoard Zynq-7000 ARM/FPGA SoC Development Board

The ConnexArray SIMD processor that will be used for offloading certain parts of the MUSIC algorithm is implemented on the FPGA on a Zedboard Zynq-7000 development board [**?**]. Therefore, we are also interested in profiling the application on the processor on the board, which is a Dual-Core ARM Cortex A9 processor. The time ratio spent in the main functions is similar to the previous results, the only difference lying in the number of samples processed in the same amount of time (in this case, 30 seconds) during which the chain is running - 5664 compared with 69879 on the Intel processor.

Table **??** presents the profiling results on the Dual-Core ARM Cortex A9 processor, leaving only the results that end up consuming more than 5% of the processing time.

Overhead	Command	Symbol
15,19%	autocorrelate8	cgemm_
10,73%	MUSIC_lin_array	cgemm_
5,73%	noise_source_c4	__logf_finite
5,62%	noise_source_c3	__logf_finite

Tabela 4: Profiling Results After Corrections on ARM Cortex A9

6 Code snippets

6.1 The C++ Flow Graph Used for Profiling

```cpp
#include <gnuradio/top_block.h>
#include <gnuradio/analog/sig_source_c.h>
#include <gnuradio/blocks/multiply_matrix_cc.h>
#include <gnuradio/blocks/null_sink.h>
#include <gnuradio/blocks/vector_to_streams.h>
#include <gnuradio/blocks/file_sink.h>
#include <gnuradio/blocks/vector_sink_f.h>
#include <gnuradio/blocks/vector_sink_c.h>
#include <gnuradio/gr_complex.h>

#include <doa/autocorrelate.h>
#include <doa/MUSIC_lin_array.h>
#include <doa/find_local_max.h>

#include <cmath>
#include <complex>
#include <chrono>
#include <thread>
#include <boost/math/constants/constants.hpp>

struct input_variables
{
    double sample_rate;
    double tone_freq1;
    double tone_freq2;
    float norm_spacing;
    int num_targets;
    int num_array_elements;
    int p_spectrum_length;
    int snapshot_size;
    int overlap_size;
};


int main(int argc, char **argv)
{
    using namespace std::complex_literals;

    gr::top_block_sptr fg = gr::make_top_block("run_MUSIC");

    const float pi = boost::math::constants::pi<float>();

    /*==========================================
     * Variables
     *==========================================*/
    struct input_variables in_vars = {
        320000, // sample_rate
        10000,  // tone_freq1
        20000,  // tone_freq2
        0.4,    // norm_spacing
        2,      // num_targets
        4,      // num_array_elements
        1024,   // p_spectrum_length
        2048,   // snapshot_size
        512     // overlap_size
    };
```

```
    int theta0_deg = 50;
    int theta1_deg = 140;
    float theta0 = pi * theta0_deg / 180;
    float theta1 = pi * theta1_deg / 180;

    std::vector<gr_complex> amv0(in_vars.num_array_elements);
    std::vector<gr_complex> amv1(in_vars.num_array_elements);
    std::vector<std::vector<gr_complex>> array_manifold_matrix(
        in_vars.num_array_elements,
        std::vector<gr_complex>(in_vars.num_targets)
    );
    float ant_loc_val;

    if (in_vars.num_array_elements % 2 == 1) {
        ant_loc_val = (float)in_vars.num_array_elements / 2;
    } else {
        ant_loc_val = (float)in_vars.num_array_elements/2 - 0.5;
    }

    // can't multiply by an integer (2) when we have -1if
    float temp_norm_spacing = in_vars.norm_spacing * 2;
    for (int i = 0; i < in_vars.num_array_elements; i++) {
        amv0[i] = std::exp(-1if * temp_norm_spacing * pi *
          ant_loc_val * cosf(theta0));
        amv1[i] = std::exp(-1if * temp_norm_spacing * pi *
          ant_loc_val * cosf(theta1));
        array_manifold_matrix[i][0] = amv0[i];
        array_manifold_matrix[i][1] = amv1[i];

        ant_loc_val--;
    }

    /*================================================
     * Blocks
     ================================================*/

    gr::analog::sig_source_c::sptr sig_src1 =
        gr::analog::sig_source_c::make(
            in_vars.sample_rate,
            gr::analog::GR_COS_WAVE, // waveform used
            in_vars.tone_freq1,      // wave frequency
            1,                       // amplitude
            0                        // offset
        );

    gr::analog::sig_source_c::sptr sig_src2 =
        gr::analog::sig_source_c::make(
            in_vars.sample_rate,
            gr::analog::GR_COS_WAVE, // waveform used
            in_vars.tone_freq2,      // wave frequency
            1,                       // amplitude
            0                        // offset
        );

    gr::blocks::multiply_matrix_cc::sptr mult_by_matrix1 =
        gr::blocks::multiply_matrix_cc::make(
            array_manifold_matrix,
            gr::block::TPP_ALL_TO_ALL
        );

```

```
117     gr::doa::autocorrelate::sptr autocorr1 =
118         gr::doa::autocorrelate::make(
119             in_vars.num_array_elements,
120             in_vars.snapshot_size,
121             in_vars.overlap_size,
122             1
123         );
124
125     gr::doa::MUSIC_lin_array::sptr music_lin_array1 =
126         gr::doa::MUSIC_lin_array::make(
127             in_vars.norm_spacing,
128             in_vars.num_targets,
129             in_vars.num_array_elements,
130             in_vars.p_spectrum_length
131         );
132
133     gr::doa::find_local_max::sptr find_local_max1 =
134         gr::doa::find_local_max::make(
135             in_vars.num_targets,
136             in_vars.p_spectrum_length,
137             0.0, // min value of index vector
138             180.0 // max value of index vector
139         );
140
141     // Null sink for locations
142     gr::blocks::null_sink::sptr null_sink1 =
143         gr::blocks::null_sink::make(
144             sizeof(float) * in_vars.num_targets // item size
145         );
146
147 #ifdef PROFILE_GR_DOA
148     // Null sink for directions of arrival ---> for profiling
149     gr::blocks::null_sink::sptr null_sink2 =
150         gr::blocks::null_sink::make(
151             sizeof(float) * in_vars.num_targets // item size
152         );
153 #else
154     // Vector to streams + Vector sinks ---> for testing
155     gr::blocks::vector_to_streams::sptr v_to_s1 =
156         gr::blocks::vector_to_streams::make(
157             sizeof(float),      // item size
158             in_vars.num_targets // number of streams
159         );
160     gr::blocks::vector_sink_f::sptr v_sink1 =
161         gr::blocks::vector_sink_f::make();
162     gr::blocks::vector_sink_f::sptr v_sink2 =
163         gr::blocks::vector_sink_f::make();
164 #endif
165
166     /*================================================
167      * Connections
168      *================================================*/
169
170     fg->connect(sig_src1, 0, mult_by_matrix1, 0);
171     fg->connect(sig_src2, 0, mult_by_matrix1, 1);
172     fg->connect(mult_by_matrix1, 0, autocorr1, 0);
173     fg->connect(mult_by_matrix1, 1, autocorr1, 1);
174     fg->connect(mult_by_matrix1, 2, autocorr1, 2);
175     fg->connect(mult_by_matrix1, 3, autocorr1, 3);
176     fg->connect(autocorr1, 0, music_lin_array1, 0);
```

```
    fg->connect(music_lin_array1, 0, find_local_max1, 0);
    fg->connect(find_local_max1, 0, null_sink1, 0);


#ifdef PROFILE_GR_DOA
    // Null sink ---> for profiling
    fg->connect(find_local_max1, 1, null_sink2, 0);
#else
    // Vector to streams + Vector sinks ---> for testing
    fg->connect(find_local_max1, 1, v_to_s1, 0);
    fg->connect(v_to_s1, 0, v_sink1, 0);
    fg->connect(v_to_s1, 1, v_sink2, 0);
#endif

    /*================================================
     * Run the graph
     *================================================*/
    fg->start();

    std::thread t([&fg]{
        // std::cout << "Sleeping...\n";
        std::this_thread::sleep_for(std::chrono::seconds(2));
        fg->stop();
        fg->wait();
    });
    t.join();

#ifndef PROFILE_GR_DOA
    // For testing
    std::vector<float> out_data1 = v_sink1->data();
    std::vector<float> out_data2 = v_sink2->data();
    std::cout << "angle1 = " << out_data1[0] << std::endl;
    std::cout << "angle2 = " << out_data2[0] << std::endl;
#endif

    return 0;
}
```

Listing 4: The C++ Flow Graph Used for Profiling

6.2 Makefile for the C++ Tester

```
WD = $(shell pwd)
GR_DOA_FP = /home/work/gr-doa
CXXFLAGS = -I$(GR_DOA_FP)/include -O0 -std=c++14 -g
LDFLAGS = -L$(GR_DOA_FP)/build/lib/
LDLIBS = -lgnuradio-runtime -lboost_system -lgnuradio-blocks \
    -lgnuradio-pmt -lgnuradio-analog -lgnuradio-doa \
    -lpthread

all: run_MUSIC
profile: run_MUSIC_profile

run_MUSIC: run_MUSIC.cc
  g++ $< $(LDFLAGS) $(CXXFLAGS) $(LDLIBS) -o $@

run_MUSIC_profile: run_MUSIC.cc
  g++ $< $(LDFLAGS) $(CXXFLAGS) -DPROFILE_GR_DOA $(LDLIBS) -o $@

```

```
18 clean:
19   rm run_MUSIC run_MUSIC_profile
```

Listing 5: Makefile for the C++ Flow Graph

Bibliografie

[1] https://github.com/EttusResearch/gr-doa/wiki

[2] TODO

[3] Rias Muhamed, *Direction of Arrival Estimation Using Antenna Arrays*, Master Thesis, Virginia Polytechnic Institute and State University, January 1996

[4] Golub, G.H., Van Loan, C.F., *Matrix Computations*, Second Edition, John Hopkins University Press, 1989

[5] TODO: take citation from doa paper

[6] http://www.swig.org/

[7] https://perf.wiki.kernel.org/

[8] http://valgrind.org/docs/manual/cl-manual.html

[9] https://kcachegrind.github.io/

[10] http://arma.sourceforge.net/