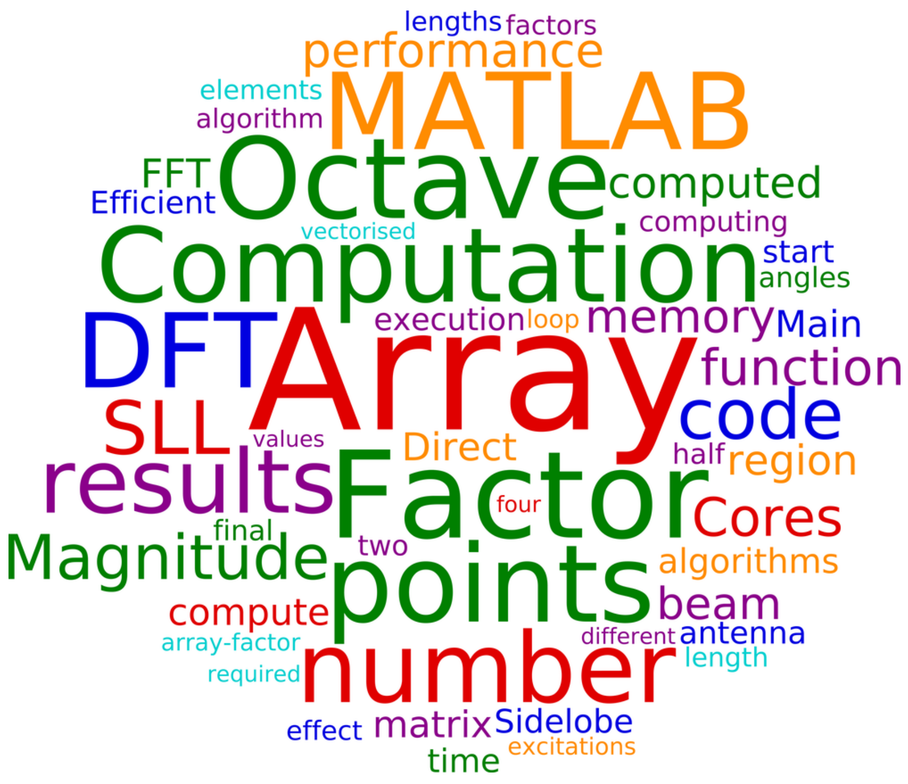


ABBREVIATIONS

ACO	ant-colony optimisation
CMA-ES	covariance matrix adaptation evolutionary strategy
CPU	central processing unit
CSIR	Council for Scientific and Industrial Research
DFT	discrete Fourier transform
FFT	fast Fourier transform
FFTW	Fastest Fourier Transform in the West
FOSS	free and open-source software
GA	genetic algorithm
IDFT	inverse discrete Fourier transform
NRF	National Research Foundation of South Africa
OS	operating system
PSO	particle-swarm optimisation
SA	simulated annealing
SLL	sidelobe level

Efficient Computation of Array Factor and Sidelobe Level (SLL) of Linear Arrays

Warren P. du Plessis

***Abstract*—The implementation of code to efficiently compute the array factor and sidelobe level (SLL) of linear antenna arrays in MATLAB and GNU Octave is considered. The use of a fast Fourier transform (FFT) to compute the array factor is shown to be more efficient than other approaches. The automatic determination of the sidelobe region as a necessary step to computing the SLL is addressed. A number of code-optimsation techniques in MATLAB and Octave are evaluated, including vectorisation, memory allocation and the use of built-in functions. Finally, an efficient function which can be used for the computation of the array factor and SLL of linear arrays in MATLAB and Octave is presented.**

***Index Terms*—Linear arrays, antenna radiation patterns, software libraries.**

I. Introduction

THE advent of high-performance desktop computing has led to the use of a wide range of numerical methods for the synthesis of antenna arrays. These numerical methods include simulated annealing (SA) [1], genetic algorithms (GAs) [2], ant-colony optimisation (ACO) [3], particle-swarm optimisation (PSO) [4], and the covariance matrix adaptation evolutionary strategy (CMA-ES) [5], among others. These algorithms have the benefit that they are able to search an entire problem space, thereby avoiding getting trapped in poor, but locally-optimal, solutions. The availability of vast computing resources have even led to exhaustive searches becoming viable in some cases [6].

The use of these numerical synthesis techniques has allowed the generation of solutions to problems which were previously considered intractable, or at best, extremely challenging. A good example is the synthesis of thinned and sparse antenna arrays, where the underlying theory was formulated in the mid 1960s [7]. In the case of thinned arrays, the density-taper algorithm proposed in 1964 [8] was the most effective synthesis technique for a number of decades, even surpassing some later approaches [9]. However, significant improvements to these pioneering results have been achieved since the mid 1990s when algorithms such as those mentioned above became viable.

Despite their significant benefits, the algorithms listed in the first paragraph all suffer from one major drawback: They require the computation of vast numbers of array factors (antenna patterns). While high-performance computing hardware does reduce the need for efficient software implementations somewhat, efficient implementations remain crucial to obtaining good results in a reasonable time. Faster code allows more options to be considered or better characterisation of the performance of algorithms [10]. And as stated above, extremely efficient implementations even allow the opportunity to use exhaustive searches for surprisingly large problems [6].

This paper thus considers a number of factors relevant to the efficient implementation of the computation of the array factor and sidelobe level (SLL) of linear arrays. This discussion will be used to develop an efficient function which can be used in MATLAB and GNU Octave, and which can be adapted to other programming languages. It is believed that this function will be useful in at least two ways. Firstly, it will provide efficient code to researchers considering linear-array synthesis which should help to speed their progress, and secondly, the use of standardised code will facilitate comparisons of the execution times of different algorithms. Furthermore, it is hoped that the application of MATLAB and Octave code-optimisation techniques to a relatively well-known problem will provide electromagnetics researchers with a better insight into how to optimise their code than the application of similar techniques to synthetic problems.

The remainder of the paper starts with Section II providing a description of the software and hardware systems which will be used to generate the presented results. Section III describes the use of the fast Fourier transform (FFT) to compute the array factor of a linear array, with special attention being paid to the angular distribution of the points at which the array factor is computed. Section IV describes a number of optimisations which can be applied to the code including vectorisation, memory allocation and the use of built-in functions. The final algorithm which results is then described and evaluated in Section V, followed by a short conclusion in Section VI.

This work is based on the research supported in part by the National Research Foundation of South Africa (NRF) (Grant specific unique reference number (UID) 85845). The NRF Grantholder acknowledges that opinions, findings and conclusions or recommendations expressed in any publication generated by the NRF supported research are that of the author(s), and that the NRF accepts no liability whatsoever in this regard.

TABLE I: The computer hardware used to test the algorithms.

		C_1	C_2	C_3
CPU	Type	Intel Xeon 5140	2×Intel Xeon 5355	2×Intel Xeon E5-2630
	Architecture	Woodcrest	Covertown	Sandy Bridge-EP
	Cores	2	2×4 = 8	2×6 = 12
	Clock speed	2.33 GHz	2.66 GHz	2.30 GHz
	Cache	4 MB	8 MB	15 MB
Main memory		6 GB	8 GB	32 GB

II. Testing Systems

A brief description of the computing hardware and software used to generate the run-time results presented in this work is provided below.

The computers were all running Debian GNU/Linux test (stretch) 4.3.5-1 (2016-02-06) x86_64 with Linux kernel 4.3.0-1-amd64 as their operating systems (OSs). The software used to run the algorithms was MATLAB R2015a (8.5.0.197613) [11] and GNU Octave 4.0.0 [12]. MATLAB is a powerful tool for scientific computing, while Octave is a free and open-source software (FOSS) tool which is compatible with many MATLAB scripts, especially when paired with the Octave-Forge extensions [13].

The main features of the three computers used for testing the algorithms are summarised in Table I. The central processing units (CPUs) of the three computers use three different architectures, and thus represent a wide range of test conditions.

III. Calculation of Array Factor

The computation of the array factor using an FFT and the number of points necessary to obtain accurate SLL values are considered in

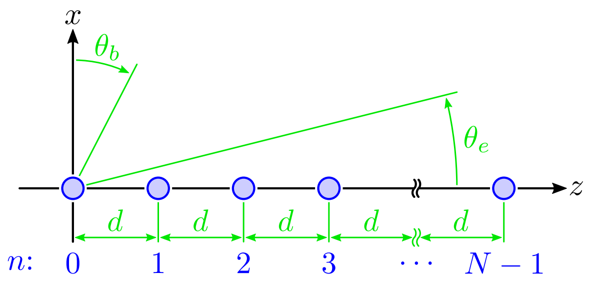


Fig. 1: The geometry of a linear antenna array with the antenna elements denoted by circles.

this section.

A. Mathematical Analysis

The geometry of a linear antenna array is shown in Fig. 1, and its pattern can be computed using [14]

$$P(\theta) = F(\theta)\, AF(\theta) \tag{1}$$

where $P(\theta)$ is the array pattern and $F(\theta)$ is the element pattern with θ denoting either θ_b or θ_e in Fig. 1. The array factor $AF(\theta)$ is given by [14]

$$AF(\theta_b) = \sum_{n=0}^{N-1} a_n e^{j\beta n d \sin(\theta_b)} \tag{2}$$

$$AF(\theta_e) = \sum_{n=0}^{N-1} a_n e^{j\beta n d \cos(\theta_e)} \tag{3}$$

where n is the index of the antenna elements, N is the number of antenna elements, a_n are the excitations of the antenna elements, β is the propagation phase constant defined by $2\pi/\lambda$ with λ denoting the wavelength, and d is the spacing between the antenna elements.

Defining

$$u = \frac{d}{\lambda}\sin(\theta_b) = \frac{d}{\lambda}\cos(\theta_e) \tag{4}$$

allows the array factor to be rewritten as

$$AF(\theta) = \sum_{n=0}^{N-1} a_n e^{j2\pi n u}. \tag{5}$$

The periodicity of the complex exponential means that

$$\begin{aligned} AF(u+l) &= \sum_{n=0}^{N-1} a_n e^{j2\pi n(u+l)} && (6)\\ &= \sum_{n=0}^{N-1} a_n e^{j2\pi n u} e^{j2\pi n l} && (7)\\ &= \sum_{n=0}^{N-1} a_n e^{j2\pi n u} && (8)\\ &= AF(u) && (9)\end{aligned}$$

where $l \in \mathbb{Z}$ and $n \in \mathbb{Z}$, so the array factor thus only needs to be computed for $u \in [0, 1)$, giving

$$AF_k = \sum_{n=0}^{N-1} a_n e^{j2\pi k n/M} \tag{10}$$

where $k \in \mathbb{Z}$, $k \in [0, M-1]$ and M is the number of angles at which the array factor is computed. Equation (10) is the equation of the inverse discrete Fourier transform (IDFT) of the antenna-array excitations, demonstrating the well-known result that the array factor of an antenna array can be computed using a discrete Fourier transform (DFT) [14].

TABLE II: The effect of the number of angular points on the accuracy of the SLL computation for a 200-element array.

Points	SLL error (dB)				
	Minimum	Median	Maximum	Mean	Std. dev.
2 000	0.000	0.033	0.254	0.044	0.042
2 500	0.000	0.024	0.151	0.032	0.030
3 000	0.000	0.017	0.109	0.023	0.022
3 500	0.000	0.015	0.084	0.019	0.016
4 000	0.000	0.012	0.062	0.015	0.013
4 500	0.000	0.009	0.051	0.012	0.011
65 536	0.000	0.000	$2.07\cdot10^{-4}$	$3.67\cdot10^{-6}$	$1.71\cdot10^{-5}$

The importance of the DFT has led to the development of FFT algorithms which apply algorithmic and code optimisations to allow efficient computation of the DFT [15], [16]. The main implication of (10) is that these efficient FFT algorithms can be exploited to achieve rapid computation of the array factor.

The angles corresponding to each value of k in (10) can be computed from

$$\begin{aligned} \frac{k}{M} &= u \bmod 1 && (11)\\ &= \left[\frac{d}{\lambda}\sin(\theta_b)\right] \bmod 1 = \left[\frac{d}{\lambda}\cos(\theta_e)\right] \bmod 1 && (12)\end{aligned}$$

where the modulo function, mod, is the remainder after division defined by

$$x \bmod y = x - y\left\lfloor \frac{x}{y} \right\rfloor. \tag{13}$$

For the special case of half-wavelength spacing of the antenna elements,

$$\sin(\theta_b) = \cos(\theta_e) = \begin{cases} 2\frac{k}{M} & \text{if } k \in \left[0, \frac{M}{2}\right] \\ 2\left(\frac{k}{M} - 1\right) & \text{if } k \in \left[\frac{M}{2}, 1\right) \end{cases} \tag{14}$$

can be used to relate k to the angles θ_b and θ_e.

The main consequence of (12) is that the equal spacing of the values of k leads to a unequal spacing in terms of the angles θ_b and θ_e. The nature of the sinusoidal functions means that the points will be most closely spaced in the broadside direction ($\theta_b = 0°$ and $\theta_e = 90°$).

B. Number of Points Required for Accurate SLL

The number of points at which the array factor is computed is clearly crucial to determining the accuracy with which the SLL is determined. Unfortunately, this critical value is not normally provided, making it difficult to assess both the accuracy of computed SLL values and to compare the execution times of different algorithms.

Table II investigates the error of the computed SLL as a function of the number of points used to compute the array factor with a FFT. Two thousand 200-element arrays using Taylor excitations [17] with $\bar{n}$ uniformly distributed over the range 2 to 30 and the SLL specification uniformly distributed over the range -10 dB to -30 dB were used. Taylor excitations were used because they offer parameters to control the pattern and generally have a single sidelobe which dominates the SLL.

From Table II, it can be seen that the SLL error decreases as the number of points increases, as anticipated. The median and mean errors for 4 000 points are less than 0.02 dB, and the maximum error is only 0.062 dB. Furthermore, the improvement from 4 000 to 4 500 points is not dramatic. Finally, 4 096 points is a power of two which leads to efficient computation of the array factor when using a FFT (see Section III-B), so 4 096 points will be used for the computation of the array factors of 200-element arrays.

IV. Code Optimisation

This section considers a number of optimisations which can be applied when MATLAB and Octave are used to compute the array factor.

The code provided in Fig. 2 will be used to generate the majority of the results provided in this section. While the details of each subsection of the code will be considered in detail below, the general-purpose control code is first briefly described.

Lines 1 to 8 of Fig. 2 set a number of constants which control the execution of the algorithm. The time required for each array-factor computation is determined using a **tic**-**toc** pair and is stored in the variable times which is initialised on Line 10 and updated in Lines 36, 40, 47, 53 and 58. The loop between Lines 11 and 60 steps through the variable n_angles which determines numbers of angular points at which the array factor is computed. Line 13, in which the number of angular points used in the computation is selected, is intentionally not terminated by a semi-colon to allow the value to be displayed as a form of progress monitor. Lines 19 to 24 initialise the random-number generator to a fixed initial state to ensure that all tests are conducted using the same array excitations. The array excitations for n_simul arrays each of which has n_elements elements are set by Line 31. Lastly, the loop between Lines 28 and 59 ensures that each test is run n_runs times so that reliable timing results can be achieved through averaging. The output variables are all cleared in Line 29 to ensure that all computations start from the same state.

Unless otherwise specified, the results presented below are the average execution times of 100 independent runs of the relevant algorithm, with each run performing 2 000 simultaneous computations of the array factors of 200-element arrays at the specified numbers of angles. The excitation of each antenna element was randomly selected from a uniformly distribution over the range $[0, 1]$.

A. *Vectorisation of Operations*

1) Description: Vectorisation is the term used to describe the process of rewriting an equation to allow large numbers of computations to be performed simultaneously by exploiting vector and matrix operators and built-in functions. The way software such as MATLAB and Octave functions means that vector and matrix operations and built-in functions are significantly faster than using loops to perform computations. Vectorisation of both the direct and FFT approaches to computing the array factor are considered below.

The vectorised form of the direct computation of the array factor in software such as MATLAB and Octave is most easily understood by noting that (5) can be rewritten as

$$\mathbf{AF} = \mathbf{U}\,\mathbf{a} \tag{15}$$

where

$$\mathbf{AF} = \begin{bmatrix} AF(\theta_0) & AF(\theta_1) & \cdots & AF(\theta_m) & \cdots & AF(\theta_{M-1}) \end{bmatrix}^T \tag{16}$$

$$\mathbf{a} = \begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_n & \cdots & a_{N-1} \end{bmatrix}^T \tag{17}$$

$$\mathbf{U} = \begin{bmatrix} e^{j\,2\pi\,0\,u_0} & e^{j\,2\pi\,1\,u_0} & \cdots & e^{j2\pi(N-1)u_0} \\ e^{j\,2\pi\,0\,u_1} & e^{j\,2\pi\,1\,u_1} & \cdots & e^{j2\pi(N-1)u_1} \\ \vdots & \vdots & \ddots & \vdots \\ e^{j\,2\pi\,0\,u_m} & e^{j\,2\pi\,1\,u_m} & \cdots & e^{j2\pi(N-1)u_m} \\ \vdots & \vdots & \ddots & \vdots \\ e^{j\,2\pi\,0\,u_{M-1}} & e^{j\,2\pi\,1\,u_{M-1}} & \cdots & e^{j2\pi(N-1)u_{M-1}} \end{bmatrix} \tag{18}$$

```
 1  % Array length.
 2  n_elements = 200;
 3  % Number of times each test is repeated.
 4  n_runs = 100;
 5  % The number of simultaneous computations.
 6  n_simul = 2e3;
 7  % The number of angles considered
 8  n_angles = [ 1999 2000 2048 2500 2503 3000
        3001 3499 3500 4000 4001 4093 4096 ];
 9
10  times = zeros(n_runs, numel(n_angles), 5);
11  for i_n_angles = 1:numel(n_angles)
12    % Initialise the angles.
13    c_angle = n_angles(i_n_angles)
14    temp = linspace(0, 2, c_angle + 1)';
15    index = temp >= 1;
16    temp(index) = temp(index) - 2;
17    fft_angles = asin(temp(1:(end - 1)));
18    dir_angles = sort(fft_angles);
19    % Intialise random number generator.
20    if exist('OCTAVE_VERSION', 'builtin')
21      rand('seed', 1); % Octave
22    else
23      rng(1); % MATLAB
24    end
25    % Initialise direct-computation matrix.
26    n = 0:(n_elements - 1);
27    U = exp(1i*2*pi*0.5*sin(dir_angles)*n);
28    for i_n_runs = 1:n_runs
29      clear dir_af fft_af lp_af lpi_af mem_af
30      % Create the arrays.
31      arrays = rand(n_elements, n_simul);
32      % Time computation of the array factors.
33      % Direct computation.
34      tic
35        dir_af = U*arrays;
36      times(i_n_runs, i_n_angles, 1) = toc;
37      % FFT computation.
38      tic
39        fft_af = ifft(arrays, c_angle);
40      times(i_n_runs, i_n_angles, 2) = toc;
41      % FFT computation - loop, pre-allocate.
42      tic
43        lpi_af = zeros(c_angle, n_simul);
44        for i_fft = 1:n_simul
45          lpi_af(:, i_fft) =
                ifft(arrays(:, i_fft), c_angle);
46        end
47      times(i_n_runs, i_n_angles, 3) = toc;
48      % FFT computation - loop.
49      tic
50        for i_fft = 1:n_simul
51          lp_af(:, i_fft) =
                ifft(arrays(:, i_fft), c_angle);
52        end
53      times(i_n_runs, i_n_angles, 4) = toc;
54      % FFT computation - wrong dimension.
55      t_arrays = arrays';
56      tic
57        mem_af = ifft(t_arrays, c_angle, 2);
58      times(i_n_runs, i_n_angles, 5) = toc;
59    end % for i_n_runs = 1:n_runs
60  end % for i_n_angles = 1:numel(n_angles)
```

Fig. 2: Code suitable for MATLAB and Octave which determines the execution time of a number of different options to compute the array factor.

with

$$u_m = \frac{d}{\lambda}\sin\left(\theta_{b\,m}\right) = \frac{d}{\lambda}\cos\left(\theta_{e\,m}\right). \tag{19}$$

The main benefit of using (15) to compute the array factor is that the matrix multiplication implicitly performs the summation in (5).

Further improvement can be achieved by assembling the column vectors of a number of excitations into a matrix, giving

$$\mathbf{A} = \begin{bmatrix} \mathbf{a_0} & \mathbf{a_1} & \mathbf{a_2} & \cdots & \mathbf{a_p} & \cdots & \mathbf{a_{P-1}} \end{bmatrix} \tag{20}$$

where there are P array excitations. The array factors can then be computed simultaneously using

$$\mathbf{AF_{all}} = \mathbf{U}\,\mathbf{A} \tag{21}$$

with

$$\mathbf{AF_{all}} = \begin{bmatrix} \mathbf{AF_0} & \mathbf{AF_1} & \cdots & \mathbf{AF_p} & \cdots & \mathbf{AF_{P-1}} \end{bmatrix} \tag{22}$$

where the required array factors are the columns of the result.

Vectorisation is even simpler when using the FFT approach to computing the array factor because MATLAB and Octave implement the required function (**ifft** ()) with parallel execution in mind. Each column of the argument passed to **ifft** () is treated as an independent computation of the FFT, with the results being returned in the corresponding columns of the result. Mathematically, this corresponds to

$$\mathbf{AF_{all}} = \mathcal{F}^{-1}\left(\mathbf{A}\right) \tag{23}$$

where $\mathcal{F}^{-1}$ denotes the inverse Fourier transform. This result is almost identical to (21), except that the matrix multiplication in (21) has been replaced by an FFT algorithm in (23).

2) Code: Lines 14 to 18 of Fig. 2 show the computation of the broadside angles (θ_b) used for the array-factor computation. Note that while the code in Fig. 2 uses the same angles for both the direct and FFT computations, this was only done to allow direct comparison between the results and is not a requirement. The formulation of the direct computation places no restrictions on the angles at which the array factor is computed, so any angles may be used. This one clear benefit of the direct approach over the FFT approach.

The implementation of (18) to compute the matrix $\mathbf{U}$ is seen in Lines 25 to 27 of Fig. 2. The variables dir_angles and n are column and row vectors respectively, so that the multiplication dir_angles *n in Line 27 results in the necessary matrix structure. Note that the computation of $\mathbf{U}$ is not included in the direct-computation timing results as it is a fast operation which only needs to be performed once.

The direct computation of the array factor in (21) is implemented by Line 35 of Fig. 2, while the FFT computation of the array factor from (21) is seen in Line 39. In both cases, the lack of loops is evident, and the simple form of the relevant code mimics that of the simplicity of (21) and (23) rather than the complexity of the underlying operations.

For comparison purposes, a version of FFT computation which is not vectorised is included in Lines 43 to 46 of Fig. 2. In this case, a loop is used to compute the array factors which were computed simultaneously in the two previous cases.

3) Results: The times taken to compute the array factor using the direct, FFT and looped FFT approaches to computing the array factor for an array with half-wavelength spacing are shown in Table III.

Table III clearly shows that the use of a FFT to compute the array factor is faster in almost every case when MATLAB is used. The only exception is on computer C_3 when the number of points used to compute the array factor is a large prime, but Section IV-C shows that this number of points is an extremely poor choice. The results for Octave differ somewhat with the FFT only being faster than the direct computation on computer C_1.

As expected, using a loop rather than a vectorised computation of the array factor leads to significant performance reductions in Table III. The benefit of vectorising operations is thus clearly demonstrated.

These results are explored further in Fig. 3 where the effect of the number of cores available for the array-factor computation is explored. Fig. 3(a) presents results for the direct computation of the array factor, while Fig. 3(b) shows the comparable results for the FFT computation.

Significantly, the improvement achieved by using q CPUs in Fig. 3 is less than q times better than the case where only one CPU is used. This result is actually well-known and applies in general, but is sometimes not as widely realised as may be expected.

The most important observation from Fig. 3 is that MATLAB clearly outperforms Octave for the cases considered.

TABLE III: A comparison between the performance of the direct and FFT approaches to computing the array factor.

Points	Case	MATLAB C_1 (ms)	MATLAB C_2 (ms)	MATLAB C_3 (ms)	Octave C_1 (ms)	Octave C_2 (ms)	Octave C_3 (ms)
4 093	Direct	915	256	48	997	375	206
	FFT	629	165	68	998	592	404
	Loop	1 188	1 060	740	1 289	2 173	1 038
4 096	Direct	959	255	48	996	374	206
	FFT	209	86	25	451	477	273
	Loop	361	335	274	643	894	522

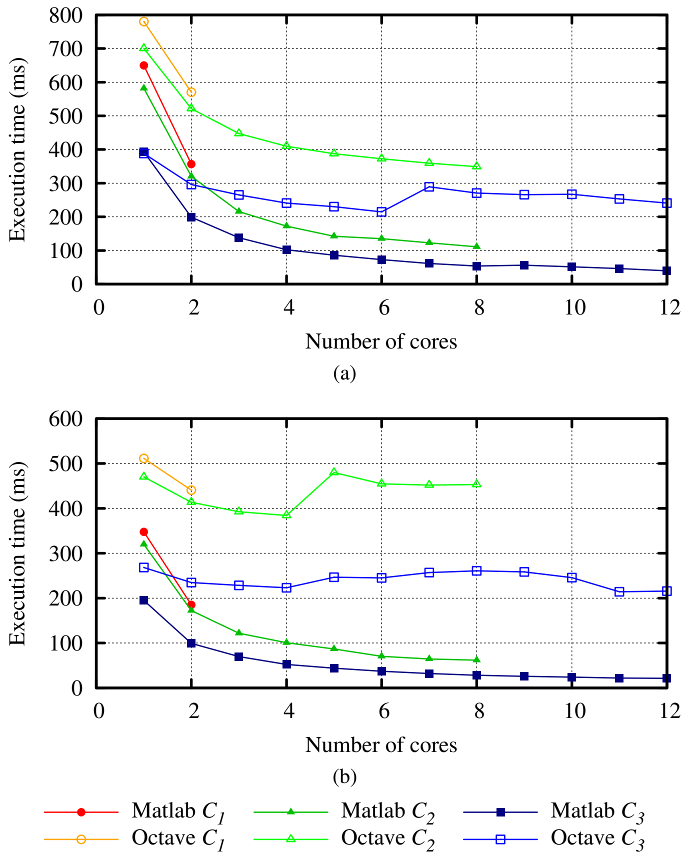


Fig. 3: The time to compute the array factor at 4 096 angular points as a function of the number of CPU cores for (a) direct and (b) FFT computation.

TABLE IV: A comparison between the performance of the direct and FFT approaches when no more than four cores are used.

Points	Case	MATLAB C_1 (2) (ms)	MATLAB C_2 (4) (ms)	MATLAB C_3 (4) (ms)	Octave C_1 (2) (ms)	Octave C_2 (4) (ms)	Octave C_3 (4) (ms)
4 093	Direct	915	456	106	997	682	535
	FFT	629	305	182	998	707	446
	Loop	1 188	1 058	746	1 289	1 538	716
4 096	Direct	959	463	106	996	691	533
	FFT	209	140	55	451	395	230
	Loop	361	335	267	643	657	424

Another important difference is that the MATLAB performance always increases when additional cores are available, while this is not always true for Octave. Specifically, the increase from six to seven cores in Fig. 3(a) leads to a substantial decrease in performance for Octave on computer C_3, and the best performance with Octave is obtained with four cores in Fig. 3(b). Possible reasons for these observations are considered in Section V-B.

Table IV shows the same results as Table III when only a maximum of four cores are available. Now the FFT results are better than the direct-computation results in all cases except for some cases where a prime number of angles is used (a poor choice as outlined in Section IV-C). Given its improved performance in most cases, the FFT approach to computing the array factor is recommended.

B. Memory Allocation

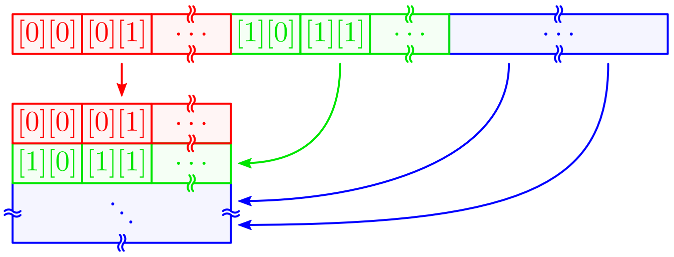
1) Description: There are two primary considerations when allocating memory in software such as MATLAB and Octave. The first is that memory should be allocated before it is used, and the second is to ensure that contiguous memory is used wherever possible.

While MATLAB and Octave allow the sizes of vectors and matrices to be changed throughout a program, this approach is inefficient. When the sizes of vectors and matrices are changed, new memory is allocated, the old values are copied to the new memory, the old memory is de-allocated, and finally, the new values are stored. This process is clearly far less efficient than simply storing the new values in pre-allocated memory.

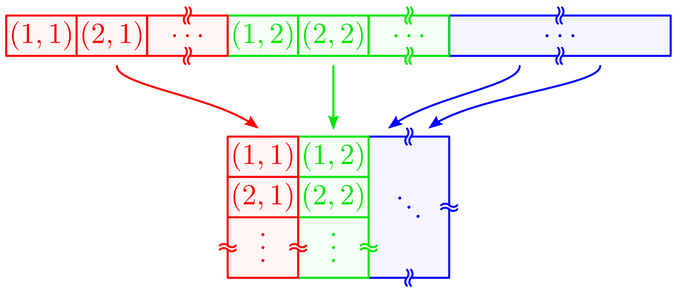
Computer memory is conceptually arranged in a linear fashion, and is optimised for accesses to nearby locations. Fig. 4 shows how physical memory is mapped to matrices. Most programming languages, including C, C++ and NumPy for Python, map contiguous memory to the rows of matrices (row-major order) as shown in Fig. 4(a), while software such as MATLAB, Octave and Fortran map the columns of matrices to contiguous memory (column-major order) as shown in Fig. 4(b).

The implication of the way memory is allocated to matrices is that accessing the next element of a column entails accessing the adjacent space in physical memory in MATLAB, Octave and Fortran. However, accessing the next element of a row means skipping across large portions of physical memory to access to next value. Values which will often be accessed together should thus be allocated to adjacent positions in columns in MATLAB, Octave and Fortran. Obviously, the opposite is true in programming languages, such as C, C++ and NumPy for Python, due to their different approach to memory allocation.

2) Code: Lines 50 to 52 in Fig. 2 are identical to Lines 43 to 46 except that the variable lp_af is not initialised outside the loop in Lines 50 to 52. This was done to illustrate the effect of failing to allocate the memory necessary in a loop, resulting in the variable having to be resized during each iteration.



(a)



(b)

Fig. 4: The physical memory layout (top) mapped to the conceptual memory layout (bottom) for (a) row-major order and (b) column-major order.

TABLE V: The effect of memory allocation on execution time.

Points	Case	MATLAB C_1 (ms)	MATLAB C_2 (ms)	MATLAB C_3 (ms)	Octave C_1 (ms)	Octave C_2 (ms)	Octave C_3 (ms)
4 093	Loop	1 188	1 060	740	1 289	2 173	1 038
	LoopI	1 285	1 159	802	188.3 s	193.9 s	90.5 s
	FFT	629	165	68	998	592	404
	Mem.	835	299	128	1 222	752	473
4 096	Loop	361	335	274	643	894	522
	LoopI	467	434	335	187.8 s	193.4 s	93.2 s
	FFT	209	86	25	451	477	273
	Mem.	406	217	83	675	829	377

The effect of accessing memory along the incorrect dimension of a loop is illustrated in Lines 55 and 57 of Fig. 2. Line 55 obtains the transpose of the array excitation so that each excitation runs along a row instead of a column. The **ifft** ()function in Line 57 is then configured to compute the array factors along rows by its third argument.

3) Results: The effect of memory allocation on algorithm execution time is shown in Table V.

The results in Table V clearly demonstrate the improvement which can be obtained by initialising arrays outside loops (“Loop”) rather than resizing arrays in loops (“LoopI”). The performance penalty is significantly lower in MATLAB than in Octave, suggesting that MATLAB includes code optimisations to address this surprisingly common suboptimal coding style.

Table V also demonstrates that allocating memory along the wrong array dimension (“Mem.”) leads to noticeably slower execution than when the correct dimension is used (“FFT”). In this case, the performance penalty is lower for Octave than for MATLAB.

C. Number of Angular Points

1) Description: While many FFT algorithms require the number of points to be a power of two ($N = 2^m$ with $m \in \mathbb{Z}$ and $M \geq 1$) [15], algorithms such as those in the Fastest Fourier Transform in the

TABLE VI: The effect of the FFT length on the performance of the FFT algorithms.

FFT length	Prime factors	MATLAB C_1 (ms)	MATLAB C_2 (ms)	MATLAB C_3 (ms)	Octave C_1 (ms)	Octave C_2 (ms)	Octave C_3 (ms)
1 999	1999	310	110	40	435	273	161
2 000	$2^4 \cdot 5^3$	138	71	23	286	292	164
2 048	2^{11}	109	72	42	203	232	126
2 500	$2^2 \cdot 5^4$	155	68	18	368	386	181
2 503	2503	357	111	40	570	349	241
3 000	$2^3 \cdot 3 \cdot 5^3$	185	62	22	441	453	216
3 001	3001	465	118	49	705	419	301
3 499	3499	580	148	63	901	510	365
3 500	$2^2 \cdot 5^3 \cdot 7$	220	73	25	516	514	250
4 000	$2^5 \cdot 5^3$	251	84	29	590	583	286
4 001	4001	624	159	66	937	550	376
4 096	2^{12}	209	86	25	451	477	273

West (FFTW) library do not have this restriction [16]. The FFTW library is arguably the most important FFT library as a result of its use in software packages such as MATLAB [18] and Octave [19].

The manual for the FFTW library states that it performs best when the length of the FFT computations is of the form [20]

$$N = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f \tag{24}$$

where a, b, c, d, e and f are integers with minimum values of zero and $e+f = 0$ or 1. This means that the FFTW algorithm is expected to give the best results when the FFT length is the product of small prime factors, and to perform poorly when the FFT length is a large prime number.

2) Code: Line 39 of Fig. 2 was used to compute the array factor using the FFT for the specified numbers of points.

3) Results: The effect of using different numbers of angular points (different FFT lengths) in MATLAB and Octave is shown in Table VI. The FFT lengths chosen ranged from 2 000 to 4 000 in steps of 500, along with the closest primes to these values and the powers of two closest to this range.

The most significant observation from Table VI is that the execution time does not increase linearly as the FFT length increases. This is anticipated in light of (24).

When the FFT lengths are a power of two (2 048 and 4 096), the execution time in Table VI is significantly lower than at nearby values. Again, this is anticipated as FFT algorithms are extremely efficient in such cases [15]. This leads to the conclusion that setting the FFT length to a power of two remains the recommended approach even when advanced FFT algorithms such as FFTW are used.

The longest execution times in Table VI are obtained when the FFT length is a prime number, as expected. Using FFT lengths which are prime factors should thus be avoided.

However, the most interesting result from Table VI is that the execution times for FFT lengths of 3 000 were faster than for FFT lengths of 4 096. Furthermore, the penalty for using 4 000 points instead of 4 096 points is surprisingly small with all but one of the penalties being 20% or less. The FFT length does thus not have to be a power of two to be efficient as long as the FFT length is a product of small primes. This leads to the surprising conclusion that rounding up to the nearest power of two can actually lead to slower execution in many cases.

The selection of the number of angular points at which the array factor is computed is thus more strongly determined by the required accuracy of the array-factor computation than by FFT algorithm performance.

TABLE VII: Comparison of two methods of computing the pattern magnitude.

Computed	MATLAB C_1 (ms)	MATLAB C_2 (ms)	MATLAB C_3 (ms)	Octave C_1 (ms)	Octave C_2 (ms)	Octave C_3 (ms)
$\lvert AF\rvert$	168	44	12	340	306	151
$\lvert AF\rvert^2$	272	239	118	348	366	183

D. Pattern Magnitude Computation

1) Description: The magnitude of the array factor is usually more important than the phase because the magnitude determines key antenna parameters such as the gain, beamwidth and SLL.

The natural approach to compute the magnitude of the array factor is to make use of the complex-magnitude function **abs**(). However, this results in the computation of

$$|AF| = \sqrt{\mathfrak{Re}\{AF\}^2 + \mathfrak{Im}\{AF\}^2} \tag{25}$$

which requires two multiplications to compute the squares, an addition and square root. Unfortunately, computation of a square root is a complex process on most computing systems and can thus lead to reduced performance.

The need to compute a square root can be removed by computing the squared magnitude of the array factor from

$$|AF|^2 = \mathfrak{Re}\{AF\}^2 + \mathfrak{Im}\{AF\}^2 . \tag{26}$$

While it may be argued that this is not the desired result, the squared magnitude and the magnitude can be used interchangeably. The key observation in this regard is that

$$|AF_1| < |AF_2| \tag{27}$$

is identical to

$$|AF_1|^2 < |AF_2|^2 . \tag{28}$$

This means that algorithms which, for example, optimise the SLL of an array factor can compare either the magnitude or the squared magnitude without affecting the result. Furthermore, the decibel value of the array factor can be computed from

$$\begin{aligned} AF_{dB} &= 20\log_{10}\left(|AF|\right) & (29)\\ &= 10\log_{10}\left(|AF|^2\right) & (30)\end{aligned}$$

so cases where, for example, the 3-dB beamwidth must be determined are merely computed by comparing to a different constant ($|AF| = 1/\sqrt{2}$ and $|AF|^2 = 1/2$) without otherwise changing the computation. The magnitude of an array factor is in units of field strength, so the conversion of the magnitude to decibels should use a factor of 20 as shown in (29).

2) Code: The computation of the magnitude using **abs**(fft_af) and the squared magnitude using **real** (fft_af) .^2 + **imag**(fft_af) .^2 were compared. The value of fft_af was computed using the code in Fig. 2.

3) Results: The comparison between the computation of the magnitude and the magnitude squared is shown in Table VII. Surprisingly, the computation of the magnitude using **abs**() is faster than the computation of the magnitude squared, despite the fact that the magnitude requires the additional computation of a square root.

The reason that the magnitude computation is faster is that **abs**() is a built-in function in MATLAB and Octave. This means that the operations required to compute the magnitude have been compiled by the authors of the MATLAB and Octave leading to fast execution. By comparison, the computation of the squared magnitude requires that five built-in functions (**real** (), **imag**(), .^2 twice and +) be performed and their results combined by the interpreter – a far slower process. This example thus demonstrates that the use of built-in functions is preferable in MATLAB and Octave unless the required function is not available and no similar functions exist.

While the above discussion is of limited value to MATLAB and Octave, it is far more significant when other programming languages such as C and C++ are considered. In languages such as these, all code is compiled, so the code implemented by a developer will be executed in the same way as library code. The implementation of a custom squared-magnitude function is thus expected to lead to significantly faster execution due to the lack of the square-root computation.

E. Determining the Start of the Sidelobe Region

1) Description: The goal of many optimisation algorithms is to minimise the SLL of the array factor of an antenna array (e.g. [1]–[6]). The challenge with optimising the SLL is that the start of the sidelobe region must be determined. Two approaches to achieving this are considered below.

The first option is simply to define the edge of the main beam as being at some angle and then to compute the SLL as the highest value outside this region. This approach is extremely efficient from a computational perspective as the start of the sidelobe region does not need to be determined. However, this approach can lead to undesirable results.

The array factors of three 50-element thinned arrays synthesised using the genetic algorithm described in [10] are shown in Fig. 5 to illustrate the three important cases which arise.

When the correct main-lobe region is used, the optimum SLL is obtained as shown in Fig. 5(a). When the main-lobe region is too small, the array factor is primarily determined by the width of the main beam as shown in Fig. 5(b), and while this does lead to a narrower main beam, it also results in a substantially worse SLL. Lastly, using a main-beam region which is too large can allow sidelobes to be considered as part of the main beam as shown in Fig. 5(c). This means that the relevant sidelobe is not included in the SLL computation leading to an incorrect SLL value which is far better than the true SLL (-23.00 dB versus -15.57 dB in Fig. 5(c)).

The main implication of Fig. 5 is that the selection of the main-lobe width is crucial to obtaining the correct SLL results. Extreme care should thus be exercised when specifying the main-beam width, and in most cases, automatic determination of the start of the sidelobe region is required.

Arguably the most reliable way to determine the extent of the main beam is to use the position of the first array-factor null. But while potentially accurate, this approach means that the roots of a polynomial need to be computed to determine the main-beam width. Such computations are extremely time-consuming, with MATLAB determining roots by computing the eigenvectors of the $n \times n$ companion matrix of an n^{th}-order polynomial [21], for example. Computing the eigenvalues of a 199×199 matrix to determine the start of the sidelobe region of a 200-element array will be extremely time-consuming.

A far more computationally efficient approach is simply to determine when the array-factor magnitude begins to increase. The only drawback of this approach is that main-beam distortion can occur as shown in Figure 6. The position of the first root from broadside is indicated and can be seen to cause a distortion of the main beam. This would not have occurred if the position of the root was used to determine the width of the main beam. However, situations like this are rare, and to the best of the author's knowledge, only occur in the case of thinned arrays where only a small proportion of the antenna elements are active and the end elements are forced to be active.

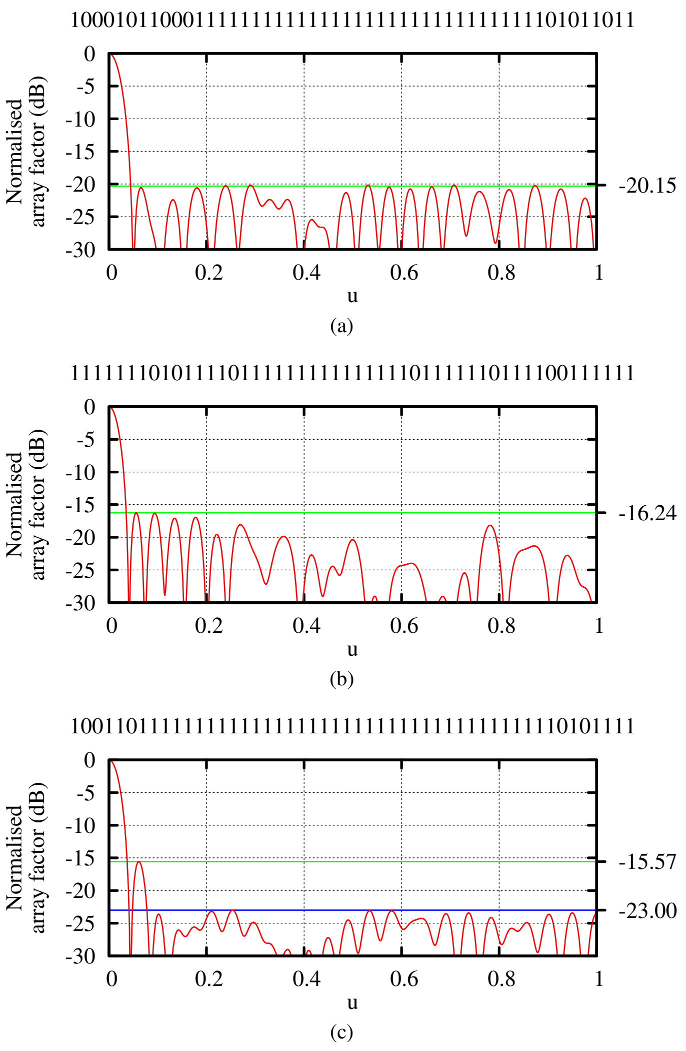


Fig. 5: The array factors for cases where the main-beam region is (a) correct, (b) too small and (c) too large. The numbers above each graph show the excitation with a 0 and 1 representing inactive and active elements, respectively.

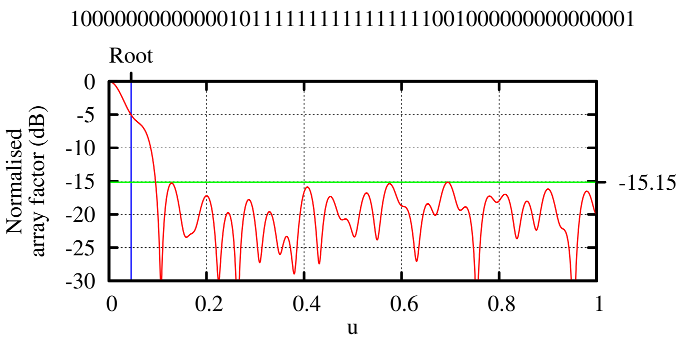


Fig. 6: Beam distortion which can occur when using the increase in the array factor to determine the main-beam region.

While it is possible to determine whether the array-factor magnitude increases for all points at which the array factor was computed, this is inefficient. As long as the excitations are purely real, the array-factor magnitude is symmetrical around its centre point [22], so only half of the points need to be considered. But more importantly, the main beam constitutes only a small portion of the array factor, so only a small number of points outside the main beam need to be considered. Even a rough estimate of the width of the main beam can thus significantly reduce the number of points which need to be considered to find the start of the sidelobe region.

The roots of an equally-excited array are equally spaced around the unit circle with the exception of the positive real axis. The index of the first array-factor null (the edge of the main beam) in this case is thus obtained by dividing the FFT length by the number of elements in the array. The first time the pattern increases now only needs to be computed around the position of this null to find the sidelobe region in the majority of cases. There will inevitably be cases where the main beam is wider than this estimated value, but these special cases are easily dealt with.

The main issue in implementing the determination of the width of the main beam is that the **find**() function in MATLAB and Octave does not allow independent operations to be performed on each column of a matrix. Unfortunately, means that the operation cannot be vectorised, making it necessary to use a loop to compute the start of the sidelobe region for a number of arrays.

2) Code: The code to determine whether the array factor magnitude is increasing is in Lines 13 to 16 of Figure 7, and the start of the sidelobe region is determined by Lines 18 to 27.

Line 14 of Figure 7 estimates the edge of the main beam as outlined above. Three times this estimated beamwidth is then used in Line 16 as the range over which to determine whether the array-factor magnitude increases. If the array-factor magnitude does not increase over the relevant range, Line 19 will not produce a result, and the condition in Line 21 will be true. The array-factor magnitude over slightly more than half the pattern is then checked (hence the addition of 2 in Line 12) to determine where the array-factor magnitude first increases. Slightly more than half the number of points are used because cases with a single null will only increase in the second half of the pattern as the null at the edge of the main beam is at $u = 1$. If Line 22 also fails to produce a result, it means that only a single element is active and the pattern has no nulls. In this case, Lines 23 to 26 designate the first point as the start of the sidelobe region to ensure that the correct of SLL of 0 dB is returned. This last case is unlikely, so Lines 23 to 26 could safely be removed in the majority of cases (though this could lead to errors during the synthesis of thinned arrays, for example).

The alternative to using the main-beam region estimate is to replace **min**(**ceil**(3∗beam_end), half_end) by half_end in Line 16 of Figure 7, and to remove Lines 13 and 14, 20 to 22 and 27. This approach would simplify the code at the cost have having to perform the computation in Line 16 on more points. While this change would be expected to significantly slow the algorithm down, the nature of vectorised computations and conditional branches in MATLAB and Octave means that this is not necessarily the case.

3) Results: The results obtained when computing start of the sidelobe region using the estimated first-null position and using half the points are shown in Table VIII as "Beam" and "Half" respectively. Three different cases with random excitations, random excitations with 10% of the arrays having wide beamwidths (the case considered by Lines 21 and 22 of Fig. 7), and random excitations with 10% of the arrays having no pattern nulls (the case considered by Lines 24 to 26) denoted by "Rand.," "Wide" and "Flat" in Table VIII respectively. The two non-random cases are included to demonstrate the effect of the conditional branches in Lines 21 to 27. Finally, arrays of 50, 100 and 200 elements are considered because shorter arrays have broader main beams, so the penalty associated with using half the available angular points is reduced for smaller arrays.

TABLE VIII: A comparison of the different approaches to determining the start of the sidelobe region.

Arrays		Case	MATLAB			Octave		
			C_1 (ms)	C_2 (ms)	C_3 (ms)	C_1 (ms)	C_2 (ms)	C_3 (ms)
Rand.	50	Beam	174	113	71	408	358	239
		Half	187	129	79	419	370	248
	100	Beam	304	182	95	655	612	377
		Half	338	213	114	684	652	396
	200	Beam	525	251	123	1 185	1 135	664
		Half	595	340	171	1 253	1 211	703
Wide	50	Beam	173	113	73	419	368	251
		Half	185	128	81	415	367	246
	100	Beam	305	182	95	661	625	390
		Half	333	213	112	676	637	392
	200	Beam	522	255	125	1 186	1 142	674
		Half	588	335	171	1 236	1 198	697
Flat	50	Beam	172	115	77	416	364	253
		Half	185	129	77	412	363	244
	100	Beam	303	183	100	656	625	391
		Half	331	213	115	670	637	393
	200	Beam	518	256	125	1 174	1 125	671
		Half	584	336	171	1 224	1 182	697

Table VIII shows that limiting the range of angles which are checked to determine the start of the sidelobe region is faster is the majority of the cases considered. The only exceptions are for some of the arrays with 50 elements when special cases comprise 10% of the arrays. However, these cases are likely to be extremely rare and are expected to comprise significantly less than 10% of the arrays considered. Furthermore, the penalty associated with these exceptions is small.

The performance improvement associated with estimating the width of the main beam is surprisingly small considering the dramatic reduction in the number of points which are checked for increases in the array factor magnitude (62 versus 2 050 points for 200 elements). This is a result of the fact that software like MATLAB and Octave executes built-in functions like **diff**() extremely quickly, while conditional branches and loops are far slower as outlined in Section IV-D.

V. Final Algorithm

The final function taking all the issues evaluated in Section IV into consideration is shown in Fig. 7. A description of the function will be followed by a brief analysis of its performance.

A. Code

The inputs to the function **thinned_sll**() are the array excitations and the number of points at which the array factors should be computed. The input array excitations are in the columns of the variable A, while the number of points at which the array factor should be computed is in the variable n_points. As outlined in Section III-B, 4 096 points gives only a small SLL error for 200-element arrays, and this value can be scaled by the number of array elements (e.g. 1 024 and 2 048 points for 50- and 100-element arrays respectively).

No error checking has been included in the code in Fig. 7 both to keep the listing brief and to avoid slowing the code down. However, this does mean that care should be exercised in ensuring that the inputs are valid.

Line 6 of Fig. 7 computes the magnitude of the array factor, and Line 8 normalises the result to its maximum value. The use

```
1  function [ SLL, AF_abs ] =
       array_sll_af(A, n_points)
2  % Compute the SLL and array factor of linear
3  %  arrays at the specified number of points.
4
5  % Compute the array factor magnitude.
6  AF_abs = abs(ifft(A, n_points));
7  % Normalise the array factor magnitude.
8  AF_abs =
       bsxfun(@rdivide, AF_abs, max(AF_abs));
9  % Allocate a variable for SLL results.
10 SLL = zeros(1, size(A, 2));
11 % The end of the unique half of the pattern.
12 half_end = ceil(n_points/2) + 2;
13 % Estimate the extent of the main beam.
14 beam_end = n_points/size(A, 1);
15 % Precompute the pattern differences.
16 AF_inc = diff(AF_abs(1:
       min(ceil(3*beam_end), half_end), :)) > 0;
17 for i_A = 1:size(A, 2)
18   % Find first pattern magnitude increase.
19   SLL_start = find(AF_inc(:, i_A), 1);
20   % If main beam is too broad.
21   if(numel(SLL_start) == 0)
22     SLL_start = find(diff(
           AF_abs(1:half_end, i_A)) > 0, 1);
23     % If pattern has no nulls - unlikely.
24     if(numel(SLL_start) == 0)
25       SLL_start = 1;
26     end
27   end
28   % Compute the SLL.
29   SLL(i_A) =
         max(AF_abs(SLL_start:half_end, i_A));
30 end % for i_A = 1:size(A, 2)
```

Fig. 7: Final algorithm for array pattern magnitude and SLL computation in MATLAB and Octave.

of **bsxfun**() vectorises the process of dividing all the values in each column by the maximum value of that column. While recent versions of Octave can vectorise such operations without the use of **bsxfun**(), its inclusion helps to highlight that vectorisation has been used and to maintain compatibility with MATLAB. The normalised array factors for each of the arrays in the input variable A are returned as the columns of the second variable returned by the function (AF_abs).

Line 10 of Fig. 7 allocates the memory where the computed SLL values are returned. Lines 13 to 27 have already been evaluated in Section IV-E. Finally, Line 29 determines the SLL by finding the maximum value of the array factor in the sidelobe region. This value is the correct SLL as a result of the fact that the normalised array factor is used for the computation. The SLL of each of the arrays in the input variable A is returned in the columns of the first variable returned by the function (SLL).

B. Results

The execution time of the final algorithm as a function of the number of CPUs used is tabulated in Table IX and plotted in Fig. 8.

As in Section IV, MATLAB outperforms Octave in all cases. While MATLAB's performance advantage over Octave is apparent for any number of cores, it is also noticeable that MATLAB is better able to leverage the benefit of additional cores. This is seen by the fact that the MATLAB execution times decrease more rapidly than for Octave as the number of cores increases.

TABLE IX: The execution time for the final algorithm.

CPUs	MATLAB C_1 (ms)	MATLAB C_2 (ms)	MATLAB C_3 (ms)	Octave C_1 (ms)	Octave C_2 (ms)	Octave C_3 (ms)
2	525	489	248	1 185	1 099	620
4		351	160		1 036	606
6		272	141		1 141	650
8		251	147		1 135	660
12			123			664

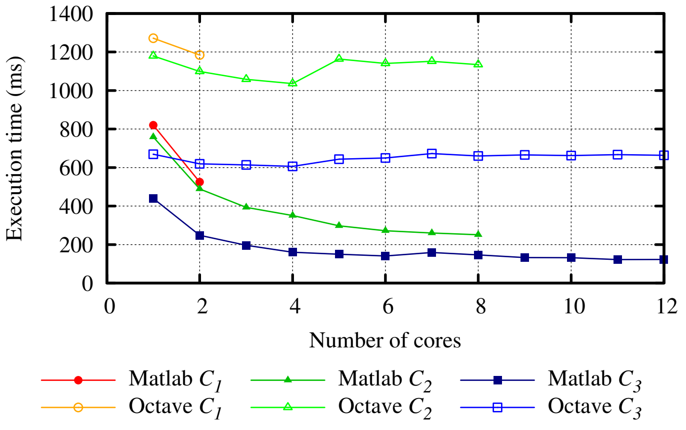


Fig. 8: The time required to execute the final algorithm.

The Octave results display a significant performance decrease when more than four cores are used. A similar observation was noted in Section IV-A where only the FFT computation was considered, so this result is most likely due to some aspect of the FFT algorithm used. The fact that pre-compiled binary files are used by the Linux distribution on the test machines suggests that the FFT library and/or Octave were optimised for four cores when they were compiled.

At this point, it is worth noting that MATLAB is a commercial software package, while both Octave and the Octave-Forge extensions are FOSS. This means that Octave can be freely used, shared and modified, and there is no limitation on the number of instances which can be run at once. Furthermore, the performance of Octave is strongly dependent on the libraries it uses, so it may be possible to improve performance by using faster libraries or better optimising the underlying libraries for the machine on which Octave is run. However, the ease with which improved performance can be achieved by MATLAB may well be sufficiently large to justify the licence cost on this basis alone.

A final point which is worth highlighting is that MATLAB and Octave use the function **log**(x) to denote the natural logarithm $\ln(x) = \log_e(x)$. Decibel values computed with (29) and (30) should thus use the function **log10**(x) corresponding to $\log_{10}(x)$ which is sometimes written $\log(x)$.

VI. Conclusion

The efficient computation of the array-factor magnitude and SLL of linear arrays in MATLAB and Octave has been studied. The final result is an efficient function which can be used to implement linear array-synthesis algorithms.

The development started with a discussion of how an FFT can be used to compute the array factor of a linear array, and an evaluation of the number of points required to obtain accurate results. The extremely efficient implementation of modern FFT algorithms was shown to outperform other approaches to computing the array factor. The determination of the start of the sidelobe region of the array factor is necessary to compute the SLL, and the use of an estimate of the width of the main beam along with error checking was shown to give good results.

A number of aspects of coding in MATLAB and Octave were considered and included in the final function including vectorisation, memory allocation, and the use of built-in functions.

References

[1] A. Trucco and V. Murino, "Stochastic optimization of linear sparse arrays," *IEEE J. Oceanic Eng.*, vol. 24, no. 3, pp. 291–299, July 1999.

[2] R. L. Haupt, "Thinned arrays using genetic algorithms," *IEEE Trans. Antennas Propag.*, vol. 42, no. 7, pp. 993–999, July 1994.

[3] O. Quevedo-Teruel and E. Rajo-Iglesias, "Ant colony optimization in thinned array synthesis with minimum sidelobe level," *IEEE Antennas Wirel. Propag. Lett.*, vol. 5, pp. 349–352, Dec. 2006.

[4] N. Jin and Y. Rahmat-Samii, "Advances in particle swarm optimization for antenna designs: real-number, binary, single-objective and multiobjective implementations," *IEEE Trans. Antennas Propag.*, vol. 55, no. 3, pp. 556–567, March 2007.

[5] M. Gregory, Z. Bayraktar, and D. Werner, "Fast optimization of electromagnetic design problems using the covariance matrix adaptation evolutionary strategy," *IEEE Trans. Antennas Propag.*, vol. 59, no. 4, pp. 1275–1285, April 2011.

[6] W. P. du Plessis, "SLL, active elements and available elements of uniformly-excited linear thinned arrays," in *Int. Conf. Electromagn. Adv. Appl. (ICEAA)*, Sept. 2012, pp. 558–561.

[7] Y. T. Lo, "Aperiodic arrays," in *Antenna Handbook*, Y. T. Lo and S. W. Lee, Eds. Van Nostrand Reinhold, 1993, vol. II – Antenna Theory, ch. 14.

[8] M. I. Skolnik, J. W. Sherman, III, and F. C. Ogg, Jr, "Statistically designed density-tapered arrays," *IEEE Trans. Antennas Propag.*, vol. 12, no. 4, pp. 408–417, July 1964.

[9] W. P. du Plessis, "Efficient synthesis of large-scale thinned arrays using a density-taper initialized genetic algorithm," in *Int. Conf. Electromagn. Adv. Appl. (ICEAA)*, Sept. 2011, pp. 363–366.

[10] W. P. du Plessis and A. bin Ghannam, "Improved seeding schemes for interleaved thinned array synthesis," *IEEE Trans. Antennas Propag.*, vol. 62, no. 11, pp. 5906–5910, Nov. 2014.

[11] (2016, Feb.) MATLAB – the language of technical computing. [Online]. Available: http://www.mathworks.com/products/matlab/

[12] (2016, Feb.) GNU Octave. [Online]. Available: https://www.gnu.org/software/octave/

[13] (2016, Feb.) Octave-Forge. [Online]. Available: http://octave.sourceforge.net/

[14] S. J. Orfanidis, *Electromagnetic Waves and Antennas*. Piscataway, NJ: Rutgers University, 2 July 2014. [Online]. Available: www.ece.rutgers.edu/~orfanidi/ewa

[15] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[16] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[17] T. T. Taylor, "Design of line-source antennas for narrow beamwidth and low side lobes," *IRE Transactions on Antennas and Propagation*, vol. 3, no. 1, pp. 16–28, Jan. 1955.

[18] (2016, Feb.) Fast Fourier transform – MATLAB fft. [Online]. Available: http://www.mathworks.com/help/matlab/ref/fft.html

[19] (2016, Feb.) GNU Octave: Signal processing. [Online]. Available: https://www.gnu.org/software/octave/doc/interpreter/Signal-Processing.html

[20] (2016, Feb.) FFTW home page. [Online]. Available: http://www.fftw.org/

[21] (2016, March) Polynomial roots – MATLAB roots. [Online]. Available: http://www.mathworks.com/help/matlab/ref/roots.html

[22] J. Tsui, *Digital Techniques for Wideband Receivers*, 2nd ed. SciTech Publishing, 2004, Available electronically from the UP library via the Knovel database.

Warren du Plessis (wduplessis@ieee.org) received the B.Eng. (Electronic) and M.Eng. (Electronic) and Ph.D. (Engineering) degrees from the University of Pretoria in 1998, 2003 and 2010 respectively, winning numerous academic awards including the prestigious Vice-Chancellor and Principal's Medal. He spent two years as a lecturer at the University of Pretoria, and then joined Grintek Antennas as a design engineer for almost four years, followed by six years at the Council for Scientific and Industrial Research (CSIR). He is currently an Associate Professor at the University of Pretoria, and his primary research interests are cross-eye jamming and thinned antenna arrays. He is a Senior Member of the IEEE.