Radar Cross Section Reduction and Shape Optimization using Adjoint Method and Automatic Differentiation

Ming Li, Junqiang Bai, and Feng Qu

School of Aeronautics
Northwestern Polytechnical University, Xi'an, 710072, China
2014200130@mail.nwpu.edu.cn, junqiang@nwpu.edu.cn, qufeng@nwpu.edu.cn

Abstract ─ An efficient Radar Cross Section (RCS) gradient evaluation method based on the adjoint method is presented. The Method of Moments is employed to solve the Combined Field Integral Equation (CFIE) and the corresponding derivatives computing routines are generated by the program transformation Automatic Differentiation (AD) technique. The differential code is developed using three kinds of AD mode: tangent mode, multidirectional tangent mode, and adjoint mode. The differential code in adjoint mode is modified and optimized by changing the "two-sweeps" architecture into the "inner-loop two-sweeps" architecture. Their efficiency and memory consumption are tested and the differential code using modified adjoint mode demonstrates the great advantages in both efficiency and memory consumption. A gradient-based shape optimization design method is established using the adjoint method and the mechanism of RCS reduction is studied. The results show that the sharp leading can avoid the specular back-scattering and the undulations of the surface could change the phases which result in a further RCS reduction.

Index Terms ─ Adjoint method, automatic differentiation, method of moments, sensitivity, shape optimization.

I. INTRODUCTION

During the last decades, the shape optimization design method has been widely applied to the aircraft design. The scattering response of an object depends on its geometry and materials, as well as the incoming wave characteristics (frequency and polarization). In general, a large number of design variables are required for the shape optimization design due to the fact that the shape of the aircraft is complicated [1]. Using the gradient-based optimization algorithms to deal with this optimization problem is a better choice since they usually converge quickly to a local optimum, regardless of the number of design variables. These algorithms require the gradient of the objective function, therefore the design sensitivity analysis is a vital step in the gradient-based shape optimization.

The adjoint method can obtain the sensitivities with respect to all design variables by solving the adjoint equation once. Due to this advantage, the adjoint method has been developed for Computational Electromagnetics (CEM) techniques, such as the Method of Moments (MoM) [2]-[5], the multilevel fast multipole algorithm (MLFMA) [6], the finite element method [7]-[8], the finite-difference time-domain (FDTD) method[9]-[10], and the transmission line method (TLM) [11]-[12]. Also, it's has been applied to the multidisciplinary optimization design, e.g., the aero-stealth coupled optimization design [1],[13].

The drawback of the adjoint method for the MoM is that the derivatives of the impedance matrix need to be differentiated which might be a complicated task. The derivatives of the impedance matrix can be computed analytically [14]-[16] or obtained by the finite difference method [2]. In [4], the Broyden update of the impedance matrix is used to estimate these derivatives. But in [17], the authors point out that in MoM discretization, the matrix elements can depend on the nodal coordinates of the mesh in a very complicated manner. With very few exceptions, straightforward analytical differentiation of the matrix elements may not be feasible. Moreover, it would require complete reprogramming of the existing codes, which is an insurmountable complication for most researchers and code developers.

An alternative way is computing the derivatives with the help of the automatic differentiation (AD) technique [18]-[19]. Using the AD tools to develop code is much more efficient and time-saving, and is suitable for dealing with these error-prone tasks. Toivanen et al. [18] demonstrate how sensitivity analysis can be incorporated into an existing in-house MoM solver with a relatively small amount of labor by using the automatic differentiation technique.

In this paper, an RCS gradient calculation approach based on the adjoint approach of Maxwell's integral equation is presented. The MoM solver is employed in the solution of the scattering problem. We adopt a parallel LU factorization driver routine of ScaLAPACK [20] to solve the Combined Field Integral Equation (CFIE). Both the current coefficient and the adjoint

Submitted On: October 28, 2020
Accepted On: February 1, 2021

https://doi.org/10.47037/2020.ACES.J.360312

coefficient would be obtained by factorizing the matrix once. The program transformation AD tool Tapenade [21] is applied to analyze the functions and subroutines of the MoM and generate the corresponding differential code. As for the derivatives of the impedance matrix computing routine, we develop the code in three different AD modes: tangent mode, multidirectional tangent mode, and adjoint mode. And then their accuracy, efficiency, and memory consumption are tested. After that, a gradient-based shape optimization design method is established by coupling the MoM, the adjoint method, the Free-Form Deformation approach (FFD)[22], and the Sequential Quadratic Programming algorithm (SQP) [23]. Finally, we apply this method to optimize an almond geometry and study the mechanism of RCS reduction.

The main objective, and the novel nontrivial contribution, of this paper, is that we modify and optimize the adjoint AD code and make it more efficient and less consumed by changing the "two-sweeps" architecture into an "inner-loop two-sweeps" architecture.

This paper is organized as follows. In Section II, the discrete adjoint equation of the integral form of the Maxwell equation based on the MoM is derived. In Section III, the derivative computing routines developed in three AD modes are presented, and the "inner-loop two-sweeps" architecture of the adjoint AD mode is discussed. Next, the flow chart of the gradient-based shape optimization design method and the numerical methods employed in the optimization framework are described in Section IV. After that, in Section V, the accuracy of the gradient is validated using the finite difference method. Besides, the CPU time and the memory consumption of three AD modes are tested. The benchmark geometry almond is optimized using the method described in this paper and the mechanism of RCS reduction is studied. Finally, the conclusions are summarized in Section VI.

II. ADJOINT METHOD

Consider a three-dimensional scattering problem where the Radar Cross Section (RCS) is defined by:

$$\sigma = 4\pi \lim_{R\to\infty} R^2 \frac{\left|\boldsymbol{E}_{\mathrm{s}}\right|^2}{\left|\boldsymbol{E}_{\mathrm{i}}\right|^2}, \tag{1}$$

where $\boldsymbol{E}_{\mathrm{s}}$ and $\boldsymbol{E}_{\mathrm{i}}$ are the scattered and incident electric field at the distance R. The scattered electric field is given by:

$$\boldsymbol{E}_{\mathrm{s}} = -\mathrm{j}k\eta \int_S \left[\boldsymbol{J}G + \frac{1}{k^2}\nabla\cdot\boldsymbol{J}\nabla G \right] dS, \tag{2}$$

where η refers to the wave impedance and k means the wavenumber. And G means the Green's function. Only the surface current $\boldsymbol{J}$ is unknown. The Rao–Wilton–Glisson (RWG) [24] basis function is adopted to discretize the surface current and then it could be expanded into a sum of N weighted basis function as shown in Eq. (3):

$$\boldsymbol{J} \cong \sum_{n=1}^{N} I_n \boldsymbol{f}_n(\boldsymbol{r}), \tag{3}$$

where $\boldsymbol{I}$ is the current coefficient and $\boldsymbol{f}_n(\boldsymbol{r})$ denotes the basis function. According to the Method of Moments, the solution of the CFIE with the Galerkin method leads to the solution of the linear system:

$$\boldsymbol{Z}\boldsymbol{I} = \boldsymbol{V}, \tag{4}$$

where $\boldsymbol{Z}$ is the impedance matrix, and $\boldsymbol{V}$ is the excitation vector. In this paper, the parallel LU factorization driver routine of ScaLAPACK[20] is applied to solve the linear system. When the parameters of the incident wave including frequency, direction, and polarization mode are given, the scattered electric field and RCS only depend on the target surface and the induced current. The derivative of the scattered electric field is given by:

$$\frac{d\boldsymbol{E}_{\mathrm{s}}}{d\boldsymbol{X}} = \frac{\partial \boldsymbol{E}_{\mathrm{s}}}{\partial \boldsymbol{X}} + \frac{\partial \boldsymbol{E}_{\mathrm{s}}}{\partial \boldsymbol{I}}\frac{d\boldsymbol{I}}{d\boldsymbol{X}}, \tag{5}$$

where $\boldsymbol{X}$ means the nodal coordinates of the target mesh. This derivative is also called the surface sensitivity, which represents the sensitivity of the scattered electric field to changes in the surface geometry. It would be of particular use for retrofitting the device on an existing object geometry where a whole new design is not feasible. Assume that the residual of the matrix equation is equal to zero:

$$\boldsymbol{R} = \boldsymbol{Z}\boldsymbol{I} - \boldsymbol{V} = 0. \tag{6}$$

The shape derivatives of the residual depend on the geometry surface and the surface current solution, that is:

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{X}} + \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{I}}\frac{d\boldsymbol{I}}{d\boldsymbol{X}} = 0. \tag{7}$$

It is intensely inefficient to calculate the term d$\boldsymbol{I}$/d$\boldsymbol{X}$ directly since that would require a mass of the MoM evaluations. Therefore we rewrite Eq. (7) as:

$$\frac{d\boldsymbol{I}}{d\boldsymbol{X}} = -\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{I}}\right]^{-1}\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{X}}. \tag{8}$$

Replacing the term d$\boldsymbol{I}$/d$\boldsymbol{X}$ in Eq. (5) and we obtain:

$$\frac{d\boldsymbol{E}_{\mathrm{s}}}{d\boldsymbol{X}} = \frac{\partial \boldsymbol{E}_{\mathrm{s}}}{\partial \boldsymbol{X}} - \frac{\partial \boldsymbol{E}_{\mathrm{s}}}{\partial \boldsymbol{I}}\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{I}}\right]^{-1}\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{X}}. \tag{9}$$

And the adjoint variable vector $\boldsymbol{\psi}$ is defined as:

$$\boldsymbol{\psi}^{\mathrm{T}} = \frac{\partial \boldsymbol{E}_{\mathrm{s}}}{\partial \boldsymbol{I}}\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{I}}\right]^{-1}. \tag{10}$$

According to Eq. (6), the derivative of residual with respect to the surface current coefficient is just the impedance matrix:

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{I}} = \frac{\partial(\boldsymbol{Z}\boldsymbol{I} - \boldsymbol{V})}{\partial \boldsymbol{I}} = \boldsymbol{Z}. \tag{11}$$

Thus, the adjoint equation is written as:

$$\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{\psi} = \left[\frac{\partial \boldsymbol{E}_{\mathrm{s}}}{\partial \boldsymbol{I}}\right]^{\mathrm{T}}. \tag{12}$$

We do not need to solve the adjoint equation

anymore since the matrix $\boldsymbol{Z}$ has been already factorized into upper and lower triangular matrices for the scattering problem. Finally, the surface sensitivity of RCS could be computed by:

$$\frac{d\sigma}{d\boldsymbol{X}} = \frac{d\sigma}{d\boldsymbol{E}_s}\frac{d\boldsymbol{E}_s}{d\boldsymbol{X}} = \frac{d\sigma}{d\boldsymbol{E}_s}\left(\frac{\partial \boldsymbol{E}_s}{\partial \boldsymbol{X}} - \boldsymbol{\psi}^T \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{X}}\right). \tag{13}$$

The Free-Form Deformation (FFD) approach [22] is adopted to parameterize the geometry and manipulate the mesh. It's more efficient to change the shape through the FFD volume than to modify the surface mesh directly. The surface mesh of the object is embedded inside the FFD volume, and all changes of the surface mesh are performed on the outer boundary of the FFD volume. Any modification of the boundary of FFD volume can be applied to indirectly modify the embedded surface mesh. The displacements of the FFD control points are selected as the design variables $\boldsymbol{x}$. And then the gradient required from the gradient-based optimization algorithm is easily obtained:

$$\frac{d\sigma}{d\boldsymbol{x}} = \frac{d\sigma}{d\boldsymbol{X}}\frac{d\boldsymbol{X}}{d\boldsymbol{x}}. \tag{14}$$

The code can be developed with $d\sigma / d\boldsymbol{X}$ (geometry nodal derivatives) or $d\sigma / d\boldsymbol{x}$ (control point derivatives). If the surface sensitivity analysis is required, the code must be developed with $d\sigma / d\boldsymbol{x}$. And then the gradient of the cost function is obtained by multiplying the surface sensitivity with $d\boldsymbol{X} / d\boldsymbol{x}$. When using the AD tools to differentiate the code in tangent mode, it's suitable to develop the code with $d\sigma / d\boldsymbol{x}$ directly:

$$\frac{d\sigma}{d\boldsymbol{x}} = \frac{d\sigma}{d\boldsymbol{E}_s}\left(\frac{\partial \boldsymbol{E}_s}{\partial \boldsymbol{x}} - \boldsymbol{\psi}^T \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}\right). \tag{15}$$

When the code is differentiated in adjoint mode, we can develop the code with either $d\sigma / d\boldsymbol{X}$ or $d\sigma / d\boldsymbol{x}$. The differences between the tangent mode and the adjoint mode of the AD technique would be discussed in the next section.

III. AUTOMATIC DIFFERENTIATION TECHNIQUE

Automatic Differentiation technique is developed to differentiate computer programs exactly without large user intervention. It's more efficient and time-saving to apply AD tools to obtain analytical derivatives of differentiable functions, in the case where these functions are provided in the form of a computer program. There are two principal ways to code the algorithm program, namely, *operator overloading* and *program transformation*. We choose the program transformation approach since it allows the tool to apply some global analysis on the program, such as the data-flow, to produce more efficient differentiated code. Tapenade [21] is an AD tool using the program transformation which, given a Fortran or C code that computes a function, creates a new code that computes its tangent or adjoint derivatives.

There are two basic modes of operation for program differentiation: tangent mode and adjoint mode. The tangent mode propagates the sensitivity at the same time as the solution is being computed. The derivative Jacobian is computed column by column, as shown in Fig. 1. Tapenade also provides an advanced tangent mode, called the multidirectional tangent mode. This mode calculates the derivative Jacobian multicolumn by multicolumn. On the contrary, the adjoint mode calculates the Jacobian row by row so that it is extremely efficient to compute the gradient of a function with respect to a large number of design variables.

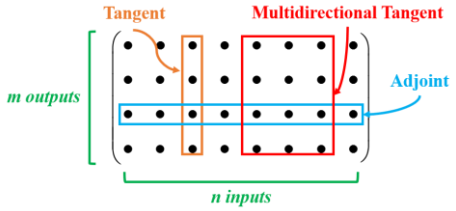


Fig. 1. Elements of the Jacobian computable by tangent mode and adjoint mode.

The crucial task of our work is the computation of residual derivative shown in Eq. (16):

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{X}} = \frac{\partial \boldsymbol{Z}\bar{\boldsymbol{I}}}{\partial \boldsymbol{X}} - \frac{\partial \boldsymbol{V}}{\partial \boldsymbol{X}}, \tag{16}$$

where $\bar{\boldsymbol{I}}$ refers to the current coefficient obtained from the scattering problem. Both of the vectors $\boldsymbol{X}$ and $\boldsymbol{R}$ have a large dimension so that it's extremely time-consuming to calculate the term $\partial \boldsymbol{R} / \partial \boldsymbol{X}$ no matter whether using the tangent mode or the adjoint mode. A feasible way to improve efficiency is computing the term $\partial \boldsymbol{R} / \partial \boldsymbol{x}$ directly instead of $\partial \boldsymbol{R} / \partial \boldsymbol{X}$ due to the fact that the number of design variables is much smaller than the number of coordinate points:

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{X}}\frac{d\boldsymbol{X}}{d\boldsymbol{x}} = \frac{\partial \boldsymbol{Z}\bar{\boldsymbol{I}}}{\partial \boldsymbol{x}} - \frac{\partial \boldsymbol{V}}{\partial \boldsymbol{x}}. \tag{17}$$

The finite difference method is a popular way to compute derivatives Jacobian since it requires a small amount of code modification. However, the step size has a great influence on the accuracy of the finite difference method. Large step size causes truncation error whereas too small a step size may lead to cancellation error. Also, we find that the appropriate step size changes with the geometric parameterization method, object shape, electromagnetic frequency, and incidence angle. In this section, we discuss the application of AD to a specific in-house MoM solver.

A. Parallel matrix filling algorithm

Before applying the AD tool to differentiate the

code, we should introduce the parallel matrix filling algorithm used in this paper. The pseudo-code of the serial matrix filling algorithm [25] is given in Fig. 5, where N_T denotes the number of triangles. First of all, the coordinates of the target surface are updated through the FFD approach according to the design variables. Next, the serial matrix filling algorithm loops over the field triangles and source triangles, and then performs the integral on each triangle pair, followed by the inner loops over the edges of the triangles. Some computational routines marked in gray show the route of information transfer from the design variables to the impedance matrix. The AD tool would analyze these routines and generate the corresponding derivative computational routines according to the chain rule.

Figure 6 depicts the pseudo-code scheme of the parallel matrix filling algorithm [25]. The modifications and improvements of the parallel matrix filling algorithm are marked in gray. In the parallel matrix filling algorithm, only a portion of the matrix is placed on each process after the computation thus the memory required for each process is reduced. The integral routine "interactions(*p*, *q*, ***X***)" (line 4 in Fig. 5), which compute the interactions between the triangle pair (*p*, *q*), is moved inside the innermost loop over the edge of a source patch (line 15 in Fig. 6). This modification avoids the redundant computations that all the processes calculate all the integrals between a pair of the source and the field triangles. To further reduce the redundancy, some computations of the intermediate data needed in the integration, such as the triangle area and the normal vector, are picked out and moved out of the innermost loop (line 9 in Fig. 6). In Fig. 6, ***v*** denotes the intermediate data. The entire procedure of the parallel matrix filling algorithm is now described.

The coordinates of the surface mesh are computed by the FFD approach, and then the code loops over all the field triangles, the source triangles, the edges on a certain field triangle, and the edges on a certain source triangle. In order to reduce the redundancy in the calculation of the integral, only the process that corresponds to the *m*th row and the *n*th column will calculate the integral over the surfaces of the triangle pair (line 7 to line 21 in Fig. 6). The corresponding processes that are about to calculate the *m*th row and the *n*th column of the matrix are picked up according to the two-dimensional block-cyclic decomposition [26] data distribution required by ScaLAPACK (line 7 and line 12 in Fig. 6). The 4 steps involved in using ScaLapack are now described.

Step 1. Create a Logical Process Grid.

Assume that the MoM solver is running on 6 processes with a 2 × 3 array of process grid layout shown in Fig. 2. The subscript of the symbol ***P*** represents the process number.

	0	1	2
0	$\boldsymbol{P_0}$	$\boldsymbol{P_1}$	$\boldsymbol{P_2}$
1	$\boldsymbol{P_3}$	$\boldsymbol{P_4}$	$\boldsymbol{P_5}$

Fig. 2. The 2 × 3 array of process grid layout.

We could use the routine "Cblacs_gridinit" to set up and initialize a process grid and run the routine "Cblacs_gridinfo" to obtain the process grid information of the current process.

Step 2. Distribute Matrices and Vectors on the Process Grid.

In this significant step, the matrics and vectors are distributed to each process according to the two-dimensional block-cyclic decomposition. The impedance matrix ***Z*** is partitioned into *MB* by *NB* blocks, and the recommended block sizes are 32 × 32 or 64 × 64. An illustration is shown in Fig. 3. The first 3 blocks in the top block row are mapped to the top row of the process grid in order, the next 3 blocks in the top row are also mapped to these same processes, and so on. Similarly, the second block row is mapped to the second grid row. When the 3rd row is reached, the mapping returns back to the first grid row. This mapping method leads to a two-dimensional block-cyclic decomposition shown in Fig. 4.

Z_{11}	Z_{12}	Z_{13}	Z_{14}	Z_{15}	Z_{16}	Z_{17}
Z_{21}	Z_{22}	Z_{23}	Z_{24}	Z_{25}	Z_{26}	Z_{27}
Z_{31}	Z_{32}	Z_{33}	Z_{34}	Z_{35}	Z_{36}	Z_{37}
Z_{41}	Z_{42}	Z_{43}	Z_{44}	Z_{45}	Z_{46}	Z_{47}
Z_{51}	Z_{52}	Z_{53}	Z_{54}	Z_{55}	Z_{56}	Z_{57}
Z_{61}	Z_{62}	Z_{63}	Z_{64}	Z_{65}	Z_{66}	Z_{67}
Z_{71}	Z_{72}	Z_{73}	Z_{74}	Z_{75}	Z_{76}	Z_{77}

Fig. 3. An example of the block matrix construction.

Each process holds a local matrix with several non-contiguous portions of the global matrix. For instance, the process $\boldsymbol{P_1}$ marked in yellow holds blocks from block rows 1,3,5,7 and block columns 2 and 5, while the process $\boldsymbol{P_3}$ marked in green holds blocks from block rows 2, 4, 6 and block columns 1, 4, and 7. We could call the ScaLAPACK routine "descinit" to create a descriptor for this block matrix. After completing the matrix distribution, we can proceed to the next step.

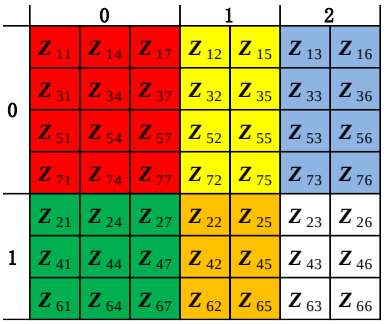


Fig. 4. An example of the two-dimensional block-cyclic decomposition.

Step 3. Call the LU Factorization Routine.

In this step, we call the ScaLAPACK routine "pzgesv" to solve the matrix equation. And then, the impedance matrix $\boldsymbol{Z}$ is replaced by the $\boldsymbol{LU}$ triangular matrix after the factorization, and the excitation vector $\boldsymbol{V}$ is replaced by the current coefficient vector.

Step 4. Release the Process Grid.

Two routines are used after finishing the calculation. A particular process grid is released with the routine "Cblacs_gridexit", and after all the computations are finished, the routine "Cblacs_exit" should be called.

B. Tangent AD

In general, the dimension of residual is far larger than the number of design variables so that it's advisable to code the program using tangent mode. Most of the CPU time is spent on computing the term $\partial \boldsymbol{Z}\bar{\boldsymbol{I}} / \partial \boldsymbol{x}$ since the impedance matrix is a large dimension dense matrix. The corresponding pseudo-code scheme of the derivatives matrix filling algorithm generated by Tapenade using tangent mode is shown in Fig. 7. Some additional routines added by Tapenade are marked in gray. It has an extra loop that loops over the design variables. In the ith cycle, the derivatives of all dependent variables with respect to the ith design variable are calculated. The variable with suffix "_d" represents the derivative of the corresponding variable with respect to the ith design variable and is calculated by the corresponding tangent routines which are suffixed with "_d". These routines are usually executed before the corresponding regular routines. Of particular note that it is unnecessary to compute the coordinates of triangles (line 4 in Fig. 7) in order to reduce the redundancy. We list it here just for the sake of program integrity. The vector $\boldsymbol{x\_d}$ has the same dimension as the design variables and the ith element is set to 1 while the others are set to 0. The vector $\boldsymbol{X\_d}$ means $d\boldsymbol{X}/dx_i$ and the matrix $\boldsymbol{Z\_d}$ refers to $\partial \boldsymbol{Z} / \partial x_i$.

C. Multidirectional tangent AD

Although the derivatives calculation accuracy of tangent AD is higher than that of the finite difference method, there is a shortcoming that reduces its efficiency. As can be seen from Fig. 7, there are masses of redundant calculations at each outermost loop, such as the computations of intermediate data $\boldsymbol{v}$. It should be noted that the subroutines, "temporary(p, q, $\boldsymbol{X}$)" and "interactions(p, q, $\boldsymbol{X}$, $\boldsymbol{v}$)", actually contain plenty of calculations and intermediate variables. Hence, it is unrealistic to store all of the intermediate variables in memory. One way to improve efficiency is by using the multidirectional tangent mode provided by Tapenade. The pseudo-code of the parallel derivative matrix filling algorithm using multidirectional tangent mode is shown in Fig. 8. Some improvements are explained as follows. The integer variable N_{out} means the number of outer loops and is given by:

$$N_{out} = \text{ceiling}(N_{DV} / n_{col}),\ n_{col} \in [1, N_{DV}], \tag{18}$$

where n_{col} indicates how many columns (see Fig. 1) are calculated in one AD multidirectional tangent calculation. The function "ceiling(x)" returns the least integer greater than or equal to x. In the AD multidirectional tangent mode, it loops over the N_{out} instead of the number of design variables N_{DV}. The larger n_{col} we set, the less redundant calculations it requires. If n_{col} equates to the N_{DV}, the Jacobian $\partial \boldsymbol{Z}\bar{\boldsymbol{I}} / \partial \boldsymbol{x}$ would be obtained by looping once. If n_{col} is set to 1, it would be the same as the ordinary tangent mode. Upon most occasions, the n_{col} might not be a factor of N_{DV} so that we define a new parameter n_{dv} which depicts the actual number of columns calculated for one particular run. This parameter is calculated by:

$$n_{dv} = \begin{cases} n_{col} & , \text{if } i \times n_{col} \le N_{DV} \\ N_{DV} - (i-1) \times n_{col} & , \text{if } i \times n_{col} > N_{DV} \end{cases}. \tag{19}$$

The code has to compute the starting index i_s and ending index i_e at the beginning of each outermost cycle (line 2 in Fig. 8). These two indexes indicate that from the i_sth to the i_eth columns of the Jacobian $\partial \boldsymbol{Z}\bar{\boldsymbol{I}} / \partial \boldsymbol{x}$ would be computed in this loop. These indexes are calculated through Eq. (20):

$$\begin{cases} i_s = (i-1) \times n_{col} + 1 \\ i_e = i \times n_{col} \end{cases}. \tag{20}$$

And then we could set some elements of $\boldsymbol{x\_dv}$ to 1 (line 3 in Fig. 8) according to the starting and ending indexes. Matrix $\boldsymbol{x\_dv}$ is an $N_{DV} \times n_{dv}$ matrix described in (21):

$$\boldsymbol{x\_dv} = \begin{bmatrix} 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \vdots & \vdots & \vdots \\ \underbrace{0 \quad 0 \quad 0}_{n_{dv}} \end{bmatrix} \begin{matrix} \\ \\ \rightarrow i_s \\ \\ \rightarrow i_e \\ \\ \rightarrow N_{DV} \end{matrix}. \tag{21}$$

The multidirectional variables with the suffix "***_dv***" (in Fig. 8) can be seen as a collection of the multiple corresponding variables with the suffix "***_d***" (in Fig. 7). For instance,

$$\boldsymbol{v\_dv} = \left[\boldsymbol{v\_d}^{(i_s)}, \boldsymbol{v\_d}^{(i_s+1)}, \ldots, \boldsymbol{v\_d}^{(i_e)} \right]. \tag{22}$$

The element $\boldsymbol{v\_d}^{(i_s)}$ is equal to the $\boldsymbol{v\_d}$ computed in the i_sth loop in Fig. 7. The data storage form of these variables depends on the user. Similarly,

$$\boldsymbol{X\_dv} = \left[\boldsymbol{X\_d}^{(i_s)}, \boldsymbol{X\_d}^{(i_s+1)}, \ldots, \boldsymbol{X\_d}^{(i_e)} \right], \tag{23}$$

$$\boldsymbol{Z\_dv} = \left[\boldsymbol{Z\_d}^{(i_s)}, \boldsymbol{Z\_d}^{(i_s+1)}, \ldots, \boldsymbol{Z\_d}^{(i_e)} \right]. \tag{24}$$

In every outermost cycle of the AD multidirectional tangent mode, the matrix ***Z_dv*** needs to be filled and stored, whose memory requirement is n_{dv} times larger than the impedance matrix. Unfortunately, we can't avoid storing the whole matrix ***Z_dv*** before the matrix-vector product due to the fact that the matrix is filled by looping triangle-to-triangle, rather than edge-to-edge. In brief, the multidirectional tangent mode is faster than the tangent mode by setting a large value of n_{col}, but it requires more memory space.

D. Adjoint AD

Both the tangent mode and multidirectional tangent mode are dependent on the number of design variables. They are inefficient if there are hundreds or thousands of design variables. On the contrary, the adjoint mode is independent of the number of design variables. However, it's impractical to compute $\partial \boldsymbol{Z}\bar{\boldsymbol{I}} / \partial \boldsymbol{x}$ directly using the adjoint mode. The Jacobian matrix $\partial \boldsymbol{Z}\bar{\boldsymbol{I}} / \partial \boldsymbol{x}$ has a large number of rows but the adjoint mode could only obtain a single row for one particular run. In order to deal with this problem, we rewrite the sensitivity of RCS Eq. (13) as:

$$\frac{d\sigma}{d\boldsymbol{X}} = \frac{d\sigma}{d\boldsymbol{E}_s}\frac{\partial \boldsymbol{E}_s}{\partial \boldsymbol{X}} - \left(\frac{d\sigma}{d\boldsymbol{E}_s}\boldsymbol{\psi}^{\mathrm{T}}\frac{\partial \boldsymbol{Z}\bar{\boldsymbol{I}}}{\partial \boldsymbol{X}} - \frac{d\sigma}{d\boldsymbol{E}_s}\boldsymbol{\psi}^{\mathrm{T}}\frac{\partial \boldsymbol{V}}{\partial \boldsymbol{X}} \right). \tag{25}$$

Instead of computing $\partial \boldsymbol{Z}\bar{\boldsymbol{I}} / \partial \boldsymbol{X}$ alone, we compute the term $\left[d\sigma / d\boldsymbol{E}_s \right] \boldsymbol{\psi}^{\mathrm{T}} \left[\partial \left(\boldsymbol{Z}\bar{\boldsymbol{I}} \right) / \partial \boldsymbol{X} \right]$ together. Thus, we define a new function:

$$F(\boldsymbol{Z}, \boldsymbol{X}) = \frac{d\sigma}{d\boldsymbol{E}_s}\boldsymbol{\psi}^{\mathrm{T}}\boldsymbol{Z}\bar{\boldsymbol{I}} . \tag{26}$$

The surface current coefficient is a constant vector, and the adjoint field does not depend directly on the design variables. Although the term $d\sigma / d\boldsymbol{E}_s$ depends on $\boldsymbol{X}$ actually, we assume that it has been computed and is regarded as a constant as well. Thus, the major calculation of this function is the matrix filling. Take the partial derivatives of F and we obtain:

$$\frac{\partial F}{\partial \boldsymbol{X}} = \frac{d\sigma}{d\boldsymbol{E}_s}\boldsymbol{\psi}^{\mathrm{T}}\frac{\partial \boldsymbol{Z}\bar{\boldsymbol{I}}}{\partial \boldsymbol{X}} , \tag{27}$$

$$\frac{\partial F}{\partial \boldsymbol{Z}} = \frac{d\sigma}{d\boldsymbol{E}_s}\boldsymbol{\psi}^{\mathrm{T}}\bar{\boldsymbol{I}} . \tag{28}$$

And then the routine that computing this function $F(\boldsymbol{Z},\boldsymbol{X})$ is analyzed by Tapenade using the adjoint mode. The mesh coordinates $\boldsymbol{X}$ are set as the inputs while F is set as the outputs when differentiating the code using Tapenade. As shown in Fig. 9, the pseudo-code depicts the procedure of derivatives evaluation and the vector $\partial F / \partial \boldsymbol{X}$ would be obtained for a single adjoint AD run. The last term of Eq. (25) is also computed in the same way.

The Tapenade adopts the *store-all* strategy [21] to differentiate the code when using the adjoint mode. In this strategy, the intermediate values are saved just before a statement, which leads to a "two-sweeps" architecture for the control-flow reversal. As shown in Fig. 9, these two sweeps are separated by a dotted line. The first sweep is called the forward sweep and is basically a copy of matrix filling (shown in Fig. 7), augmented with a recording of the control. This recorded control would be used by the second sweep, called the backward sweep, to orchestrate control-flow reversal. The intermediate data ($\boldsymbol{v_1}$ and $\boldsymbol{v_2}$) that would be used by the backward sweep to evaluate the elements of the derivatives Jacobian is also recorded. The natural way to record is to use a stack that grows during the forward sweep and shrinks during the backward sweep. Of particular note that the subroutine "interactions'(p, q, $\boldsymbol{X}$, $\boldsymbol{v_1}$)" (line 19 in Fig. 9) is the simplification of the original integral subroutine "interactions(p, q, $\boldsymbol{X}$, $\boldsymbol{v_1}$)" (line 15 in Fig. 6). The simplified subroutine only calculates the intermediate data used for the integral on the triangle, not the matrix elements.

The Tapenade uses the PUSH and POP primitives for stack manipulations and applies the global data-flow analysis To-Be-Recorded (TBR) [21] to reduce significantly the number of intermediate values that need to be stored on that tape. In Fig. 9, the variables with suffix "_b" represent the derivatives of the F with respect to the corresponding variables. For instance, ***X_b*** means $\partial F / \partial \boldsymbol{X}$. The corresponding backward sweep subroutines are suffixed by "_b". The PUSH/POP subroutines provided by Tapenade are used to record the intermediate values whereas the PUSHCONTROL and POPCONTROL subroutines are called for the control-flow recording. These PUSH and POP primitives are marked in gray in Fig. 9. The vector $\partial F / \partial \boldsymbol{X}$ could be obtained by running these two sweeps once which shows the great merit of the adjoint mode. However, there is still a serious problem that affects its application. As we mentioned above, the forward sweep is basically a copy of the matrix filling augmented with the data recording. Assume that the number of unknowns is N, the regular matrix filling routine only needs to store an $N \times N$ complex matrix in total. But for the forward sweep, if computing each element of Jacobian requires to store n intermediate values, it will push at least $n \times N \times N$ data into the stack. The memory cost is unacceptable even though it would not compute and store the elements of the

impedance matrix.

In order to address this problem, we propose an "inner-loop two-sweeps" architecture for the adjoint mode. Note that the computation of each element of the impedance matrix or Jacobian is independent. Therefore the backward sweep routine could be executed straight after the corresponding forward sweep is done in an inner cycle. In other words, we could change the "two-sweeps" architecture into the "inner-loop two-sweeps" architecture. The modified adjoint AD pseudo-code using the "inner-loop two-sweeps" architecture is depicted in Fig. 10. These modifications can only be done by hand. The manual programming work depends on the architecture of the existing codes. If the framework of the existing codes is clear and modularized, the complete reprogramming work could be done within a couple of days.

```
1.  X = FFD(x)                                  ! compute the coordinates of triangles
2.  Do p = 1, N_T                               ! loop over the field (testing) triangles
3.    Do q = 1, N_T                             ! loop over the source triangles
4.      dZ = interactions(p, q, X)              ! calculate integral on the triangle pair (p, q)
5.      Do ii = 1, 3                            ! loop over edges of the field triangle
6.        mm = edge_num(p, ii)                  ! compute the global index of the iith edge of the pth field triangle
7.        If (mm .NE. 0) then                   ! the mmth edge is a valid common edge
8.          Do jj = 1, 3                        ! loop over edges of the source triangle
9.            nn = edge_num(q, jj)              ! compute the global index of the jjth edge of the qth source triangle
10.           If (nn .NE. 0) then               ! the nnth edge is a valid common edge
11.             Z(mm, nn) += dZ(mm, nn)         ! add into the impedance matrix
12.           Endif
13.         Enddo                               ! end loop over edges of the source triangle
14.       Endif
15.     Enddo                                   ! end loop over edges of the field triangle
16.   Enddo                                     ! end loop over the source triangles
17. Enddo                                       ! end loop over the field (testing) triangles
```

Fig. 5. The serial matrix filling algorithm.

```
1.  X = FFD(x)                                  ! compute the coordinates of triangles
2.  Do p = 1, N_T                               ! loop over the field (testing) triangles
3.    Do q = 1, N_T                             ! loop over the source triangles
4.      flag = 0                                ! initialize the flag of whether do integration
5.      Do ii = 1, 3                            ! loop over edges of the field triangle
6.        m = edge_num(p, ii)                   ! compute the global index of the iith edge of the pth field triangle
7.        If (m .NE. 0 .and. m is on this process) then   ! the mth edge is valid and on this process
8.          mm = local_index(m)                 ! get the local index of the global index m
9.          v = temporary(p, q, X)              ! compute the intermediate data needed in the integration
10.         Do jj = 1, 3                        ! loop over edges of the source triangle
11.           n = edge_num(q, jj)               ! compute the global index of the jjth edge of the qth source triangle
12.           If (n .NE. 0 .and. n is on this process) then   ! the nth edge is valid and on this process
13.             nn = local_index(n)             ! get the local index of the global index n
14.             If (flag == 0) then
15.               dZ = interactions(p, q, X, v) ! calculate integral on the triangle pair (p, q)
16.               flag = 1                      ! set flag that the integration has been done
17.             Endif
18.             Z(mm, nn) += dZ(mm, nn)         ! add into the impedance matrix
19.           Endif
20.         Enddo                               ! end loop over edges of the source triangle
21.       Endif
22.     Enddo                                   ! end loop over edges of the field triangle
23.   Enddo                                     ! end loop over the source triangles
24. Enddo                                       ! end loop over the field (testing) triangles
```

Fig. 6. The parallel matrix filling algorithm.

```
1.  Do i = 1, N_DV                                          ! loop over the design variables
2.    x_d = [0,..,0,1,0,..,0]^T                             ! set the ith element to 1 and the others to 0
3.    X_d = FFD_d(x, x_d)                                   ! compute dX/dx_i
4.    X = FFD(x)                                            ! compute the coordinates of triangles
5.    Do p = 1, N_T                                         ! loop over the field (testing) triangles
6.      Do q = 1, N_T                                       ! loop over the source triangles
7.        flag = 0                                          ! initialize the flag of whether do integration
8.        Do ii = 1,3                                       ! loop over edges of the field triangle
9.          m = edge_num(p, ii)                             ! compute the global index of the iith edge of the pth field triangle
10.         If (m .NE.0 .and. m is on this process) then    ! the mth edge is valid and on this process
11.           mm = local_index(m)                           ! get the local index of the global index m
12.           v_d = temporary_d(p, q, X, X_d)               ! compute the derivatives of the intermediate data
13.           v = temporary(p, q, X)                        ! compute the intermediate data needed in the integration
14.           Do jj = 1, 3                                  ! loop over edges of the source triangle
15.             n = edge_num(q, jj)                         ! compute the global index of the jjth edge of the qth source triangle
16.             If (n .NE. 0 .and. n is on this process) then   ! the nth edge is valid and on this process
17.               nn = local_index(n)                       ! get the local index of the global index n
18.               If (flag == 0) then
19.                 dZ_d = interactions_d(p,q,X, X_d, v, v_d)   ! calculate the derivatives matrix elements
20.                 flag = 1                                ! set flag that the integration has been done
21.               Endif
22.               Z_d(mm, nn) += dZ_d(mm, nn)               ! add into the derivatives matrix
23.             Endif
24.           Enddo                                         ! end loop over edges of the source triangle
25.         Endif
26.       Enddo                                             ! end loop over edges of the field triangle
27.     Enddo                                               ! end loop over the source triangles
28.   Enddo                                                 ! end loop over the field (testing) triangles
29.   ZI_d(:, i) = multiplications(Z_d, I)                  !compute ∂ZI / ∂x_i and release the memory of Z_d
30. Enddo                                                   !end loop over design variables
```

Fig. 7. The parallel derivative matrix filling algorithm using the tangent mode.

```
1.  Do i = 1, N_out                                         ! loop over the outermost loops
2.    compute_index(i, i_s, i_e, n_dv)                      ! compute the starting and ending indexes
3.    x_dv = set_ones(i_s, i_e)                             ! set some elements to 1 and others to 0
4.    X_dv = FFD_dv(x, x_dv, n_dv)                          ! compute the derivatives of the coordinates
5.    X = FFD(x)                                            ! compute the coordinates of triangles
6.    Do p = 1, N_T                                         ! loop over the field (testing) triangles
7.      Do q = 1, N_T                                       ! loop over the source triangles
8.        flag = 0                                          ! initialize the flag of whether do integration
9.        Do ii = 1, 3                                      ! loop over edges of the field triangle
10.         m = edge_num(p, ii)                             ! compute the global index of the iith edge of the pth field triangle
11.         If (m .NE. 0 .and. m is on this process) then   ! the mth edge is valid and on this process
12.           mm = local_index(m)                           ! get the local index of the global index m
13.           v_dv = temporary_dv(p, q, X, X_dv, n_dv)      ! compute the derivatives of the intermediate data
14.           v = temporary(p, q, X)                        ! compute the intermediate data needed in the integration
15.           Do jj = 1, 3                                  ! loop over edges of the source triangle
16.             n = edge_num(q, jj)                         ! compute the global index of the jjth edge of the qth source triangle
17.             If (n .NE. 0 .and. n is on this process) then   ! the nth edge is valid and on this process
18.               nn = local_index(n)                       ! get the local index of the global index n
19.               If (flag == 0) then
20.                 dZ_dv = interactions_dv(p, q, X, X_dv, v, v_dv, n_dv)   ! calculate the derivatives matrix elements
21.                 flag = 1                                ! set flag that the integration has been done
22.               Endif
23.               Z_dv(mm, nn, :) += dZ_dv(mm, nn, :)       ! add into the derivatives matrix
24.             Endif
25.           Enddo                                         ! end loop over edges of the source triangle
26.         Endif
27.       Enddo                                             ! end loop over edges of the field triangle
28.     Enddo                                               ! end loop over the source triangles
29.   Enddo                                                 ! end loop over the field (testing) triangles
30.   ZI_d(:, (i-1) × n_dv + 1: i × n_dv) = Multiplications(Z_dv, I)   ! matrix multiplications and release memory
31. Enddo                                                   ! end loop over design variables
```

Fig. 8. The parallel derivative matrix filling algorithm using the multidirectional tangent mode.

```
1.  X = FFD(x)                                          ! compute the coordinates of triangles
2.  Do p = 1, N_T                                       ! loop over the field (testing) triangles
3.    Do q = 1, N_T                                     ! loop over the source triangles
4.      flag = 0                                        ! initialize the flag of whether do integration
5.      Do ii = 1, 3                                    ! loop over edges of the field triangle
6.        m = edge_num(p, ii)                           ! compute the global index of the iith edge of the pth field triangle
7.        If (m .NE.0 .and. m is on this process) then  ! the mth edge is valid and on this process
8.          PUSH(mm)                                    ! push data into the stack
9.          mm = local_index(m)                         ! get the local index of the global index m
10.         PUSH(v_1)                                   ! push data into the stack
11.         v_1 = temporary(p, q, X)                    ! compute the  intermediate data
12.         Do jj = 1, 3                                ! loop over edges of the source triangle
13.           n = edge_num(q, jj)                       ! compute the global index of the jjth edge of the qth source triangle
14.           If (n .NE. 0 .and. n is on this process) then ! the nth edge is valid and on this process
15.             PUSH(nn)                                ! push data into the stack
16.             nn = local_index(n)                     ! get the local index of the global index n
17.             If (flag == 0) then
18.               PUSH(v_1, v_2)                        ! push data into the stack
19.               v_2 = interactions'(p, q, X, v_1)     ! calculate the  intermediate data
20.               flag = 1                              ! set flag that the integration has been done
21.               PUSHCONTROL(1)                        ! push control parameter into the stack
22.             Else
23.               PUSHCONTROL(0)                        ! push control parameter into the stack
24.             Endif
25.             PUSHCONTROL(1)                          ! push control parameter into the stack
26.           Else
27.             PUSHCONTROL(0)                          ! push control parameter into the stack
28.           Endif
29.         Enddo                                       ! end loop over edges of the source triangle
30.         PUSHCONTROL(1)                              ! push control parameter into the stack
31.       Else
32.         PUSHCONTROL(0)                              ! push control parameter into the stack
33.       Endif
34.     Enddo                                           ! end loop over edges of the field triangle
35.   Enddo                                             ! end loop over the source triangles
36. Enddo                                               ! end loop over the field (testing) triangles
-------------------------------------------------------------------------------------------------------
37. Z_b = ∂F / ∂Z                                       ! set the input derivatives
38. Do p = N_T,1,-1                                     ! loop over the field (testing) triangles
39.   Do q = N_T,1,-1                                   ! loop over the source triangles
40.     Do ii=3,1,-1                                    ! loop over edges of the field triangle
41.       POPCONTROL(branch)                            ! pop the control parameter from stack
42.       If (branch) then
43.         Do jj=3,1,-1                                ! loop over edges of the source triangle
44.           POPCONTROL(branch)                        ! pop the control parameter from stack
45.           If (branch) then
46.             dZ_b(mm, nn) += Z_b(mm, nn)             ! set the derivatives matrix elements
47.             POPCONTROL(branch)                      ! pop the control parameter from stack
48.             If (branch) then
49.               POP(v_1, v_2)                         ! pop the data from stack
50.               interactions_b(p, q, X, X_b, v_1, v_1_b, v_2, v_2_b, dZ_b) ! calculate the derivatives of the  intermediate data
51.             Endif
52.             POP(nn)                                 ! pop the data from the stack
53.           Endif
54.         Enddo                                       ! end loop over edges of the source triangle
55.         POP(v_1)                                    ! pop the data from stack
56.         temporary_b(p, q, X, X_b, v_1, v_1_b)       ! calculate the derivatives ∂F / ∂X
57.         POP(mm)                                     ! pop the data from the stack
58.       Endif
59.     Enddo                                           ! end loop over edges of the field triangle
60.   Enddo                                             ! end loop over the source triangles
61. Enddo                                               ! end loop over the field (testing) triangles
62. x_b = FFD_b(x, X, X_b)                              ! compute ∂F / ∂x
```

Fig. 9. The parallel derivative matrix filling algorithm generated by the Tapenade using adjoint mode.

```
1.  Z_b = ∂F / ∂Z                                   ! set the input derivatives
2.  X = FFD(x)                                      ! compute the coordinates of triangles
3.  Do p = 1, N_T                                   ! loop over the field (testing) triangles
4.    Do q = 1, N_T                                 ! loop over the source triangles
5.      flag = 0                                    ! initialize the flag of whether do integration
6.      Do ii = 1, 3                                ! loop over edges of the field triangle
7.        m = edge_num(p, ii)                       ! compute the global index of the iith edge of the pth field triangle
8.        If (m .NE.0 .and. m is on this process) then   ! the mth edge is valid and on this process
9.          mm = local_index(m)                     ! get the local index of the global index m
10.         PUSH(v1')                               ! push data into the stack
11.         v1' = temporary(p, q, X)                ! compute the intermediate data
12.         Do jj = 1, 3                            ! loop over edges of the source triangle
13.           n = edge_num(q, jj)                   ! compute the global index of the jjth edge of the qth source triangle
14.           If (n .NE. 0 .and. n is on this process) then   ! the nth edge is valid and on this process
15.             nn = local_index(n)                 ! get the local index of the global index n
16.             If (flag == 0) then
17.               PUSH(v1', v2')                    ! push data into the stack
18.               v2' = interactions'(p, q, X, v1')   ! calculate the intermediate data
19.               flag = 1                          ! set flag that the integration has been done
20.               PUSHCONTROL(1)                    ! push control parameter into the stack
21.             Else
22.               PUSHCONTROL(0)                    ! push control parameter into the stack
23.             Endif
-------------------------------------------------------------------------------------------------------------------------
24.             dZ_b(mm, nn) += Z_b(mm, nn)         ! set the derivatives matrix elements
25.             POPCONTROL(branch)                  ! pop the control parameter from stack
26.             If (branch) then
27.               POP(v1', v2')                     ! pop the data from the stack
28.               interactions_b(p, q, X, X_b, v1', v1_b', v2', v2_b', dZ_b)   ! calculate the derivatives of the intermediate data
29.             Endif
30.           Endif
31.         Enddo                                   ! end loop over edges of the source triangle
32.         POP(v1')                                ! pop the data from the stack
33.         temporary_b(p, q, X, X_b, v1', v1_b')   ! calculate the derivatives ∂F / ∂X
34.       Endif
35.     Enddo                                       ! end loop over edges of the field triangle
36.   Enddo                                         ! end loop over the source triangles
37. Enddo                                           ! end loop over the field (testing) triangles
38. x_b = FFD_b(x, X, X_b)                          ! compute ∂F / ∂x
```

Fig. 10. The parallel derivative matrix filling algorithm using the "inner-loop two-sweeps" architecture.

As for the "two-sweeps" architecture, the forward sweep loop over the field triangles, the source triangles, the edges on a certain field triangle, and the edges on a certain source triangle in order to compute and store the intermediate data ($\boldsymbol{v}_1$ and $\boldsymbol{v}_2$). This intermediate data is used for the backward subroutine (such as "interactions_b" and "temporary_b") to calculate the derivatives. The intermediate data will change in each loop, therefore all of the intermediate data needs to be pushed into the stack (lines 10 and 18 in Fig. 9) before starting the backward sweep. The intermediate data will be popped out of the stack when it is needed for the backward subroutines (lines 49 and 55 in Fig. 9).

As for the "inner-loop two-sweeps" architecture, all of the backward sweep routines are moved into the loops of the forward sweep. The backward subroutine "interactions_b" is in the same loop as the subroutine "interactions' ", the intermediate data calculated by the subroutine "interactions' " can be sent into the backward subroutine "interactions_b" directly. Similarly, the backward subroutine "temporary_b" is in the same loop as the subroutine "temporary". The memory consumption is much lower than before since it is hardly inevitable to record plenty of intermediate data values and control parameters. Even though there are still some small amounts of data values ($\boldsymbol{v}_1'$ and $\boldsymbol{v}_2'$) that need to be stored, they would be pulled out of the stack before the end of the current loop.

IV. OPTIMIZATION FRAMEWORK

The flow chart of the gradient-based shape optimization design method is presented in Fig. 11. The numerical methods applied in the optimization process consist of the geometric parameterization, the MoM solver, the adjoint-based gradient evaluations, and the SQP algorithm. The operational process and relationships among the methods mentioned above are described below.

First of all, the mesh of the baseline geometry is parameterized through the FFD approach. And then the parameterize method updates the mesh and transfers it to the MoM solver. After solving the scattering problem, the gradient could be obtained through the adjoint method. Next, the RCS, gradient, and some geometrical

parameters (e.g., thickness) are sent to the optimizer to search the optimized direction and step. The SQP algorithm will generate new design variables and start the next iteration process until the convergence tolerance reaches the required accuracy. Of particular note is that the gradient calculation is not required at every iteration, it is determined by the SQP algorithm. In this paper, the implementation of the SQP algorithm is SNOPT [27], which is useful for solving large-scale constrained problems with smooth objective functions and constraints. SNOPT is a sparse nonlinear optimizer that uses a smooth augmented Lagrangian merit function while making explicit provision for infeasibility in the original problem and in the quadratic programming subproblems. The Hessian of the Lagrangian is approximated through a limited-memory quasi-Newton method, and a reduced-Hessian algorithm is used for solving the quadratic programming subproblems [27].

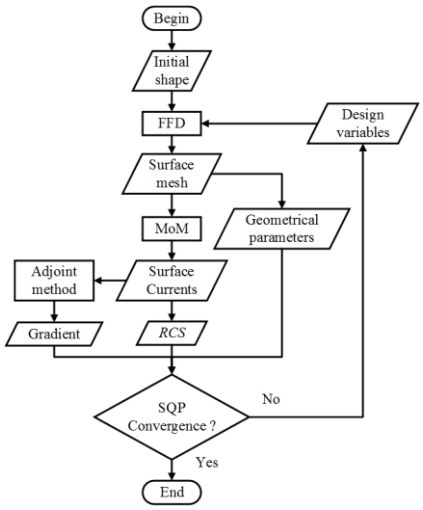


Fig. 11. The flow chart of the gradient-based shape optimization design method.

V. NUMERICAL EXAMPLES

In this section, we present some numerical examples. Firstly, we verify the accuracy of the gradient computed by the adjoint method. Then we study the CPU time and memory consumption of adjoint AD compared to the tangent AD and multidirectional tangent AD. Finally, we apply the presented method to a shape optimization problem.

A. Verification

To gain confidence in the effectiveness of the gradient for use in the optimization design, the adjoint method is compared with the traditional forward finite difference method. The metallic almond [28] model is applied for the electromagnetic analysis. The frequency of the incident wave is 7 GHz and the polarization mode is horizontal polarization. Figure 12 shows the mesh, the FFD control frame, and the incident direction. Each edge length of the triangle facet is less than the 1/10 wavelength and the amount of unknowns is 12618. The almond model is parameterized by the FFD approach with 56 control points in total. The displacements at the z-direction of the FFD control points are selected as the design variables.

The gradients computed by the adjoint method and the finite difference method are shown in Fig. 13. Notice that several step sizes are tested to find the appropriate step size before using the finite difference method to compute the gradient. The gradients obtained by the adjoint method are in good agreement with those computed by the finite difference method. The absolute error and relative error between these two results are shown in Fig. 14. The relative error is given by $\varepsilon_r = \left|F' - F'_{\text{ref}}\right| / \left|F'_{\text{ref}}\right|$, where $F'_n = \mathrm{d}\sigma / \mathrm{d}x_n$, $n = 1,2,\ldots,56$. The values calculated by the finite difference method are selected as the reference values. From this figure, it can be seen that both the absolute error and relative error are less than 10^{-2}. The gradient computed by the adjoint method has acceptable accuracy for the gradient-based shape optimization design.

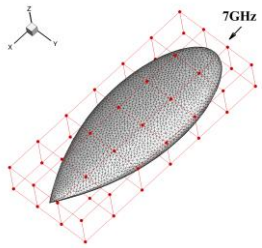


Fig. 12. The mesh of almond and the FFD control frame.

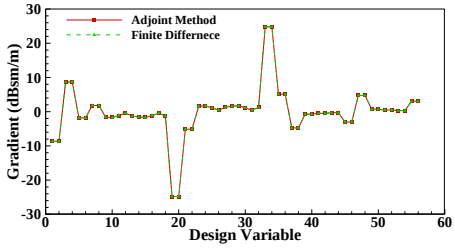


Fig. 13. Comparison of the gradients.

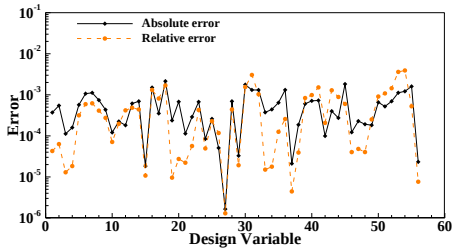


Fig. 14. Absolute error and relative error.

B. Computation time and memory consumption

In this section, the comparisons of efficiency and consumption between (multidirectional) tangent AD and adjoint AD are studied. These simulations are run on a cluster with 28 CPU cores. The total CPU time and the memory consumption are tested through a different number of design variables and the results are shown in Fig. 15 and Fig. 16. The total CPU time is the sum of the CPU time of all cores. Notice that solving a regular scattering problem is about 1.28 hours total CPU time and requires 3.08 GB memory space.

As for the adjoint AD using "inner-loop two-sweeps" architecture, it requires minimal time and memory space, which are a little bit large than the requirements of solving a regular scattering problem. The total CPU time and the memory consumption are consistent when increasing the number of design variables.

The tangent AD (red broken line) requires maximum running time but less memory consumption. The total CPU time grows linearly with the number of design variables while the memory consumption keeps stable.

The multidirectional tangent AD is tested using different n_{col} values (16 in blue dotted line and 32 in green dash-dot line). The larger n_{col} we set, the less CPU time it spends. But the multidirectional tangent with larger n_{col} requires more memory space, especially when dealing with the problem with a large number of design variables.

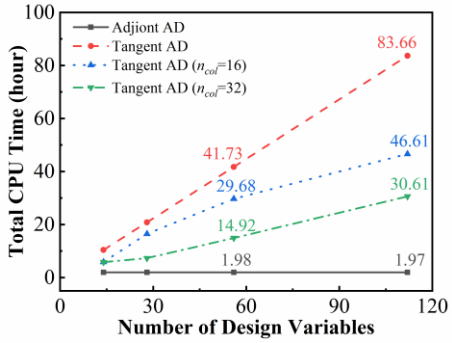


Fig. 15. Total CPU time consumption.

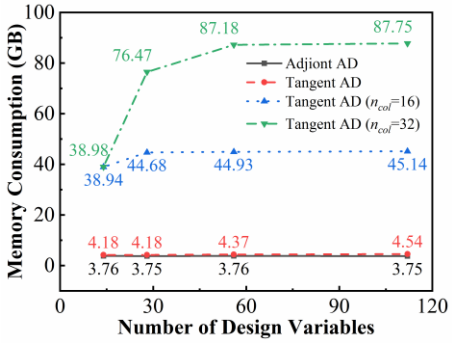


Fig. 16. Memory consumption.

In order to show the evidence for the advantages of our proposed method, we perform a comparison between our code and the commercial software HFSS-IE (HFSS Integral Equation) [29]. The cube model shown in Fig. 17 is used for the electromagnetic analysis.

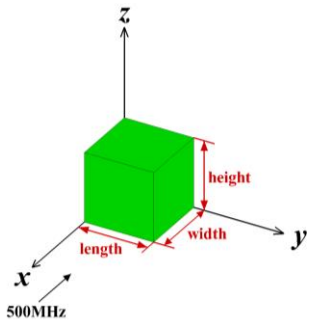


Fig. 17. Cube model and incident wave.

The side length of the cube is 1 meter, and the frequency of the incident wave is 500MHz. The length, width, and height of this cube are selected as the design variables. Thus, there are 3 design variables in total. Firstly, we compute the gradients of RCS using our code in adjoint AD mode. The unknowns of the matrix equation is 8118 and the total CPU time and the memory consumption are listed in Table 1. And then, we do the same simulation using the HFSS-IE. The Adaptive Cross Approximation (ACA) [29] technique provided by the HFSS-IE is applied to solve the integral equation, and the maximum residual error is set to 0.004. The HFSS-IE employs the central finite difference approximation to calculate the RCS derivatives with respect to the design variables. The maximum number of iterations is set to 6 and the approximate error in master is set to 0.001 when running the sensitivity analysis. Both the electromagnetic simulations are running on a workstation with 16 CPU cores.

These comparison results are listed in Table 1. It has been found that the total CPU time required by the proposed method is less than that required by the HFSS-IE, while the memory consumption of the proposed method is only a little higher than that of the HFSS-IE.

Table 1: Comparison of the total CPU time and the memory consumption

	Total CPU Time	Memory Consumption
Proposed method	48.5 min	2010 MB
HFSS-IE	451.7 min	1720 MB

In short, the adjoint AD shows great advantages in both efficiency and memory consumption, and the tangent AD is inefficient whereas the multidirectional tangent AD requires large memory consumption.

C. Application to the shape optimization

We study the shape optimization of the almond and the requirements for a low observable shape. The optimization problem is described in Eq. (29):

$$\begin{aligned} &\min: \ \sigma(\boldsymbol{x}) \\ &\text{s.t.}: \ t^{(n)}(\boldsymbol{x}) \geq 0.1 t_0^{(n)}, \ n = 1,2,...,130, \\ &\qquad \forall \boldsymbol{x} \in [x_1, x_2, ..., x_{56}]^{\mathrm{T}} \end{aligned} \tag{29}$$

where t denotes the thickness at a certain point of optimized shape whereas t_0 is the initial thickness. The thickness constraints are imposed at points on the surface of the object to avoid unrealistic designs. As shown in Fig. 18, there are 130 segments inside the almond and their length represents the local thickness. They should not less than 10% of the initial shape to prevent the thickness from being so thin. The objective function is the RCS and the design variables are the displacements at the z-direction of the FFD control points. The frequency of the incident wave is 7GHz and the polarization mode is vertical polarization.

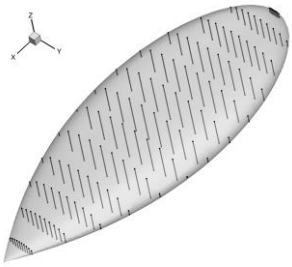


Fig. 18. Thickness constraints.

The optimizer arrives after 4 iterations, and 11 evaluations of RCS, to a local optimum σ_* = -61.629 dBsm (decibel square meter, $\sigma_{\mathrm{dBsm}} = 10\log(\sigma_{m^2})$). Figure 19 shows the convergence history of the objective function. We refine the mesh of the optimized shape since large deformation would lead to distorted mesh elements. The final RCS of the optimized shape after mesh refinement is -53.194 dBsm. Figure 20 presents the profiles (Coordinate Y=0m) of the almond and optimized shape. The optimized shape has undulations and a sharp leading edge. The sharp leading edge could change the specular scattering into the edge diffraction and lead to a large RCS reduction. Figure 21 shows the distribution of the surface current density and the optimized shape has a lower current magnitude around the leading edge when compared to the almond. As for the instantaneous magnitude of the surface current, it's a periodic distribution from the leading edge to the end. The interval is approximately equal to a wavelength.

In order to study the RCS reduction mechanism of the undulations, we divide the model shape into several parts according to the length of a quarter wavelength. As shown in Fig. 22, two parts (part5 and part9) are picked out to study their scattering field contribution. These two parts are located on either side of the maximum thickness and are one wavelength apart. The scattering electric field contribution of each part is computed by integrating the surface current on this part individually. Figure 23 shows the instantaneous scattering electric field along the reflection direction (negative direction of x-axis) from the origin of the coordinate system and Fig. 24 depicts the phase of the scattering electric field.

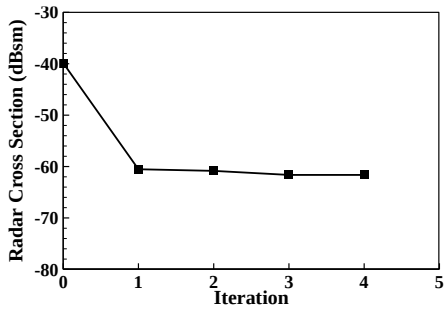


Fig. 19. The convergence history of the objective function.

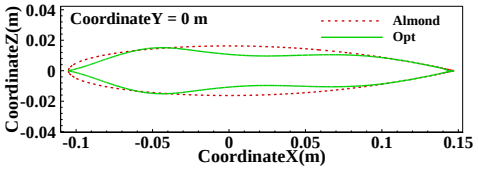
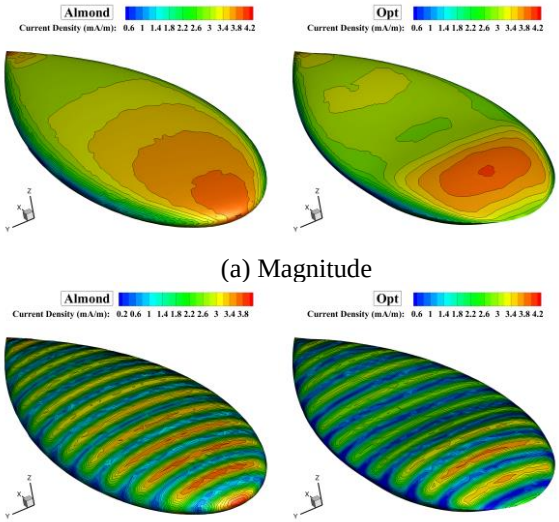


Fig. 20. Comparison of the profiles.



(a) Magnitude

(b) Instantaneous magnitude

Fig. 21. Comparison of the surface current density.

As for the almond, the scattering electric fields generated by these two parts are in the same phase. The

total amplitude increases after the superimposition of these two scattering electric fields.

On the contrary, the undulations of the optimized shape change the phase of the scattering electric fields. The phase difference between them is approximately 180 degrees which leads to a cancellation of the total amplitude. As for the other parts, the undulations would enlarge the phase difference of each pair and weaken the superimposed effect.

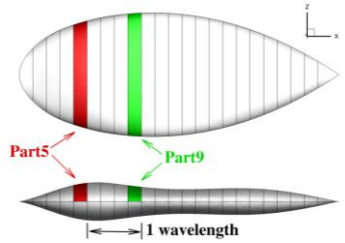
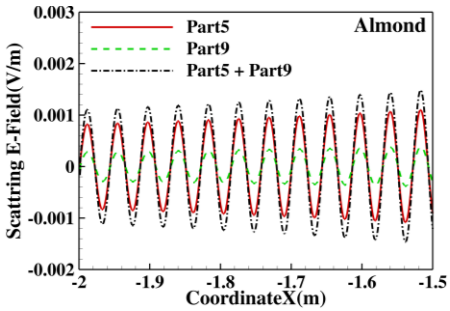
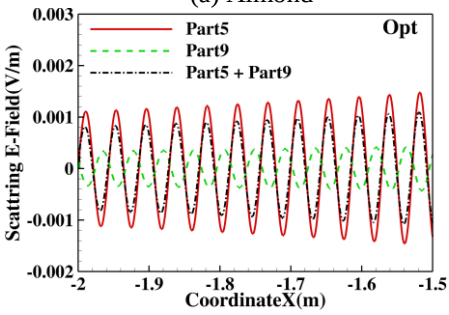


Fig. 22. Schematic of the part division.

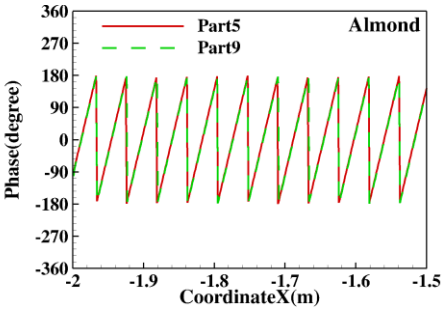


(a) Almond

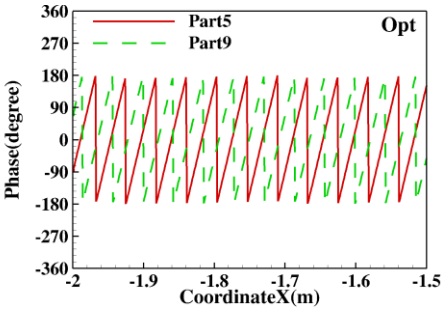


(b) Opt

Fig. 23. Scattering electric field from each part.



(a) Almond



(b) Opt

Fig. 24. The phase of the instantaneous scattering electric field.

VI. CONCLUSION

In this work, the MoM is applied to solve the CFIE and evaluate the RCS of the object. The adjoint equation based on MoM is derived to compute the gradient of RCS efficiently. The LU factorization routine of ScaLAPACK is called to solve the large scale complex dense matrix equation so that the adjoint equation no longer needs to be solved.

The most difficult task for the gradient evaluation is the computation of the derivatives of the impedance matrix. The program transformation AD tool Tapenade is applied to generate the derivatives computation routines. We develop the code in three AD modes and test their efficiency and memory consumption. One of the bright spots of our work is that the subroutine which computes the derivatives of impedance matrix using adjoint AD mode is optimized by changing the "two-sweeps" architecture into the "inner-loop two- sweeps" architecture. This modification makes it far faster than the codes generated by tangent and multidirectional tangent modes. In addition, the memory consumption of this architecture is friendly.

The gradient calculated through the adjoint method is compared with those computed by the finite difference method. The results show that the accuracy is satisfactory. Both of the absolute errors and relative errors are in an acceptable region. The accuracy of the gradient meets the requirement of gradient-based shape optimization.

A gradient-based shape optimization design method is developed by coupling the MoM, the adjoint method, the FFD approach, and the SQP algorithm. The almond geometry is optimized through this design method and the SQP reaches a local minimum within 10 iterations. The optimized shape has undulations and sharp leading edges. Further studies show that the sharp leading edge could reduce the surface current magnitude and avoid the specular back-scattering, resulting in a large RCS reduction. The undulations on the upper surface and lower surface could change the phases which leads to a further RCS reduction.

REFERENCES

[1] M. Li, J. Bai, L. Li, X. Meng, Q. Liu, and B. Chen, "A gradient-based aero-stealth optimization design method for flying wing aircraft," *Aerosp. Sci. Technol.*, vol. 92, pp. 156-169, June 2019.

[2] N. Georgieva, S. Glavic, M. Bakr, and J. Bandler, "Feasible adjoint sensitivity technique for EM design optimization," *IEEE Trans. Microw. Theory Techn*., vol. 50, no. 12, pp. 2751-2758, Dec. 2002.

[3] A. Bondeson, Y. Yang, and P. Weinerfelt, "Optimization of radar cross section by a gradient method," *IEEE Trans. Magn.*, vol. 40, pp. 1260-1263, Mar. 2004.

[4] N. K. Nikolova, R. Safian, E. A. Soliman, M. H. Bakr, and J. W. Bandler, "Accelerated gradient based optimization using adjoint sensitivities," *IEEE Trans. Antennas Propag*., vol. 52, pp. 2147-2157, Aug. 2004.

[5] Y. Zhang, R. K. Nikolova, and R. H. Bakr, "Input impedance sensitivity analysis of patch antenna with discrete perturbations on method-of-moment grids," *Applied Computational Electromagnetics Society Journal*, vol. 10, no. 25, pp. 867-876. Oct. 2010.

[6] J. Kataja, S. J. Rvenp, J. I. Toivanen, R. M. Kinen, and P. Y. Oijala, "Shape sensitivity analysis and gradient-based optimization of large structures using MLFMA," *IEEE Trans. Antennas Propag.*, vol. 62, Nov. 2014.

[7] X. Zhang, J. C. I. Newman, W. Lin, and W. K. Anderson, "Time-dependent adjoint formulation for metamaterial optimization using Petrov-Galerkin methods," *Applied Computational Electromagnetics Society Journal*, vol. 32, no. 2, pp. 236-239. Feb. 2018.

[8] H. Igarashi and K. Watanabe, "Complex adjoint variable method for finite-element analysis of eddy current problems," *IEEE Trans. Magn*., vol. 46, no. 8, pp. 2739-2742, Aug. 2010.

[9] M. M. T. Maghrabi, M. H. Bakr, S. Kumar, A. Z. Elsherbeni, and V. Demir, "FDTD-based adjoint sensitivity analysis of high-frequency nonlinear structures," *IEEE Trans. Antennas Propag.*, vol. 68, pp. 4727-4737, June 2020.

[10] Y. Song, N. K. Nikolova, and M. H. Bakr, "Efficient time-domain sensitivity analysis using coarse grids," *Applied Computational Electromagnetics Society Journal*, vol. 23, no. 1, pp. 5-15, Mar. 2008.

[11] S. M. Ali, N. K. Nikolova, and M. H. Bakr, "Semi-analytical approach to sensitivity analysis of lossy inhomogeneous structures," *Applied Computational Electromagnetics Society Journal*, vol. 22, no. 2, pp. 219-227, July 2007.

[12] L. S. Kalantari and M. H. Bakr, "Optical cloak design exploiting efficient anisotropic adjoint sensitivity analysis," *Applied Computational Electromagnetics Society Journal*, vol. 32, no. 5, pp. 449-454, May 2017.

[13] L. Zhou, J. Huang, Z. Gao, and W. Zhang, "Three-dimensional aerodynamic/stealth optimization based on adjoint sensitivity analysis for scattering problem," *AIAA Journal*, pp. 1-14, Mar. 2020.

[14] J. Kataja and J. I. Toivanen, "On shape differentiation of discretized electric field integral equation," *Eng. Anal. Bound. Elem.*, vol. 37, pp. 1197-1203, May 2013.

[15] M. S. Dadash, N. K. Nikolova, and J. W. Bandler, "Analytical adjoint sensitivity formula for the scattering parameters of metallic structures," *IEEE Trans. Microw. Theory Tech.*, vol. 60, pp. 2713-2722, Sep. 2012.

[16] J. Kataja, A. G. Polimeridis, J. R. Mosig, and P. Yla-Oijala, "Analytical shape derivatives of the MFIE system matrix discretized with RWG functions," *IEEE Trans. Antennas Propag.*, vol. 61, pp. 985-988, Feb. 2013.

[17] N. K. Nikolova, J. W. Bandler, and M. H. Bakr, "Adjoint techniques for sensitivity analysis in high-frequency structure CAD," *IEEE Trans. Microw. Theory Tech*, vol. 52, pp. 403-419, Jan. 2004.

[18] J. I. Toivanen, R. A. E. Mäkinen, S. Järvenpää, P. Ylä-Oijala, and J. Rahola, "Electromagnetic sensitivity analysis and shape optimization using method of moments and automatic differentiation," *IEEE Trans. Antennas Propag.*, vol. 57, pp. 168-175, Jan. 2009.

[19] V. Fischer, L. Gerbaud, and F. Wurtz, "Using automatic code differentiation for optimization," *IEEE Trans. Magn*., vol. 41, pp. 1812-1815, May 2005.

[20] J. Dongarra and L. S. Blackford, "ScaLAPACK tutorial," in *Proceedings of the Third International Workshop on Applied Parallel Computing*, Industrial Computation and Optimization, 1997.

[21] L. Hascoet and V. Pascual, "The tapenade automatic differentiation tool: principles, model, and specification," *ACM Transactions on Mathematical Software*, vol. 39, pp. 1-43, Apr. 2013.

[22] T. W. Sederberg and S. R. Parry, "Free-form deformation of solid geometric models," in *ACM SIGGRAPH Computer Graphics*, pp. 151-160, Aug. 1986.

[23] A. Barclay, "SQP methods for large-scale optimization," *Dissertation Abstracts International*, vol. 60-06, Section B, p. 2730, Chair: Philip Gill, 1999.

[24] S. Rao, D. Wilton, and A. Glisson, "Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Trans. Antennas Propag*., vol. 30, pp. 409-418, May 1982.

[25] Y. Zhang, T. K. Sarkar, D. D. Oro, H. Moon, and R. Geijn, "A parallel mom code using RWG basis functions and ScaLAPACK-based in-core and out-of-core solvers," in *Parallel Solution of Integral*

Equation-Based EM Problems in the Frequency Domain: John Wiley & Sons, pp. 81-84, 2009.
[26] V. Eijkhout, J. Langou, and J. Dongarra, "Parallel Linear Algebra Software," *Netlib Repository at UTK and ORNL*, 2006.
[27] P. E. Fill, W. Murray, and M. A. Saunders, "SNOPT: An SQP algorithm for large-scale constrained optimization," *SIAM Journal on Optimization*, vol. 12, no. 4, pp. 979-1006, Apr. 2002.
[28] A. C. Woo, H. Wang, M. J. Schuh, and M. L. Sanders, "Benchmark radar targets for the validation of computational electromagnetics programs," *IEEE Antennas Propag. Mag.*, vol. 35, pp. 84-89, Feb. 1993.
[29] HFSS help manual (ANSYS ED).

Ming Li was born in Liuzhou, Guangxi, China, in 1992. He received the B.S. degree in Aircraft Design and Engineering from Beijing Institute of Technology, Beijing, in 2014, and the M.S. degree in Aircraft Design from Northwestern Polytechnical University, Xi'an, in 2017, where he is currently pursuing the Ph.D. degree in Aircraft Design with the School of Aeronautics.

His current research interests include computational electromagnetics, aerodynamics, and multidisciplinary optimization design of flight vehicles.

Junqiang Bai was born in Xinxiang, Hernan, China, in 1971. He received the B.S. in Aerodynamics from the National University of Defense Technology, Changsha, in 1991, the M.S. degree in Aerodynamics from Northwestern Polytechnical University, Xi'an, in 1994, and the Ph.D. degree in Aircraft Design from Northwestern Polytechnical University, Xi'an, in 1999.

From 1999 to 2004, he was an Associate Professor with the Department of Aircraft Design Engineering, Northwestern Polytechnical University, Xi'an. He was a Visiting Scholar with the Institute of Aerodynamic and Flow Technology, German Aerospace Center (DLR), Braunschweig, Germany, in 2006. Since 2004, he has been a Professor with the Department of Aircraft Design Engineering, Northwestern Polytechnical University, Xi'an. Since 2017, he has been a Deputy Dean with the Unmanned System Research Institute, Northwestern Polytechnical University, Xi'an, China. He has authored or co-authored over 100 papers and he holds or has applied for 5 Chinese patents. His current research interests include the conceptual design of aircraft, the aircraft aerodynamic shape design, and the aircraft multidisciplinary optimization.

Feng Qu was born in Pizhou, Jiangsu, China, in 1988. He received the B.Eng. degree in Mathematics and Applied Mathematics from China University of Petroleum, Beijing, in 2010, and the Ph.D. degree in Fluid Mechanics from Beihang University, Beijing, in 2015.

From 2015 to 2017, he was an Engineer with the Institute of Manned Space System Engineering, China. Since 2018, he has been an Associate Professor with the Northwestern Polytechnical University, Xi'an, and he is currently a Director of fluid mechanics in National Natural Science Foundation of China. His current research interests involve the flux schemes for all speeds, the high-order schemes, and the turbulence modeling. He is authoring a structured CFD software which is capable of simulating configures of all speeds in both laminar flow and turbulent flow.