Chapter 9

Adaptive Cross Approximation

In previous chapters we discussed the difficulties encountered when applying the Method of Moments to large problems. Foremost among these are the system RAM required to store the MoM matrix and the $O(N^3)$ computational cost of a direct LU factorization. If the number of right-hand sides is small, iterative techniques (Section 4.2) may require less runtime than LU factorization. Radar cross section problems, however, often have thousands of right-hand sides. Additionally, the MoM system matrix is often poorly conditioned, and so preconditioning is typically needed. However, preconditioners carry with them their own memory and CPU overhead, and even then the number of iterations per right-hand side may be very high or the solution may not converge at all. If the system matrix does not fit in RAM, there have historically been few options other than modifying the geometry to reduce the number of unknowns or using virtual memory (disk) with an out-of-core solver. The latter option, however, may make the problem intractable due to the far slower read/write speed of hard disks. With the increasing availability of multicore CPUs and parallel processing, the overall computational burden has been lessened, but the difficulties with memory requirements remain. These issues motivate direct-solution techniques that reduce memory requirements while retaining the advantages of a direct solver.

In this chapter we investigate the Adaptive Cross Approximation (ACA), a method by which a significant portion of the MoM system matrix is compressed and stored in memory directly. A direct LU factorization is then performed, optimized using a CPU- or GPU-accelerated Level 3 BLAS. The ACA is particularly attractive for radar cross section problems, as it permits solution of many right-hand sides simultaneously. As the ACA is a purely matrix-based method, requiring no knowledge about the underlying problem, our discussion here focuses primarily on the required linear algebra and on the changes needed to add ACA capability to an existing full-matrix 3D MoM solver. In Chapter 10, we will then extend the ACA further to a Multi-Level Adaptive Cross Approximation (MLACA), which improves compressibility further at the expense of additional runtime. In Chapter 11, we will then examine the Fast Multipole Method (FMM), which shares some fundamental concepts with the ACA and MLACA but compresses and solves the MoM matrix system in a different way.

9.1 Rank Deficiency

The matrix system that results from MoM discretization and testing has full rank. However, it is well known that if the basis functions are partitioned into spatially localized groups, the matrix becomes a block structure in which the off-diagonal blocks are *rank-deficient* [1]. To begin, let us consider a matrix block $\mathbf{Z}_{m\times n}$ having m rows and n columns, which can be decomposed using the singular-value decomposition (SVD) as

$$\mathbf{Z}_{m\times n} = \mathbf{U}_{m\times m}\mathbf{\Sigma}_{m\times n}\mathbf{V}^*_{n\times n} \,, \tag{9.1}$$

where the diagonals of $\mathbf{\Sigma}$ comprise the singular values, and $\mathbf{U}$ and $\mathbf{V}^*$ contain the left- and right-singular vectors, respectively. As $\mathbf{Z}$ is rank-deficient, the singular values in $\mathbf{\Sigma}$ decrease quickly below machine precision. To illustrate, consider the two roughly spherical groups of basis functions in Figure 9.1, having diameter a and centers separated by distance D.

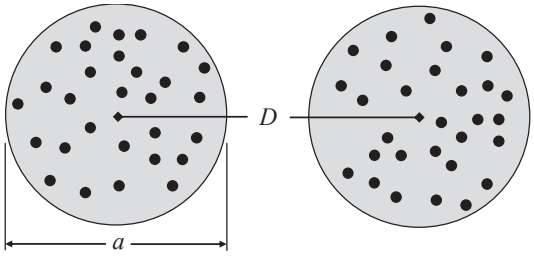


Figure 9.1: Groups of Basis Functions

We now define the matrix elements Z_{mn} as the 3D Green's function

$$Z_{mn} = G(\mathbf{r},\mathbf{r}') = \frac{e^{-jkR_{mn}}}{R_{mn}} \,, \tag{9.2}$$

where $R_{mn} = |\mathbf{r}_m - \mathbf{r}_n|$. We now compute $\mathbf{Z}_{m\times n}$ for spherical regions of diameter $a = 1\lambda$ having 256 randomly distributed points, where the separation distance D is assigned values of 0, 2λ and 4λ. This corresponds to overlapping groups, groups whose bounding spheres just touch, and well-separated groups. In Figure 9.2 are plotted the normalized singular values of $\mathbf{Z}$ for each case, using a minimum threshold of 10^{-6}. This is very close to machine precision when using 32-bit floating point variables. We see that when the groups overlap, the singular values fall off very slowly. However, as the separation increases, they fall off much faster, and when $D = 4\lambda$, only the first 28 singular values are above the threshold. This is an intuitive result, given that as the separation between groups becomes large, there is less variation in R_{mn} and $\mathbf{Z}$ begins to take on redundant information. This example shows that, for well-separated groups, $\mathbf{Z}_{m\times n}$ could be approximated as

$$\mathbf{Z}_{m\times n} \approx \mathbf{U}_{m\times k}\mathbf{\Sigma}_{k\times k}\mathbf{V}^*_{k\times n} = \mathbf{Z}_u^{m\times k}\mathbf{Z}_v^{k\times n} \,, \tag{9.3}$$

where the *effective* rank $k \ll m,n$ is chosen based on a threshold τ of largest to smallest singular values. Only $\mathbf{Z}_u$ and $\mathbf{Z}_v$ would be stored explicitly.

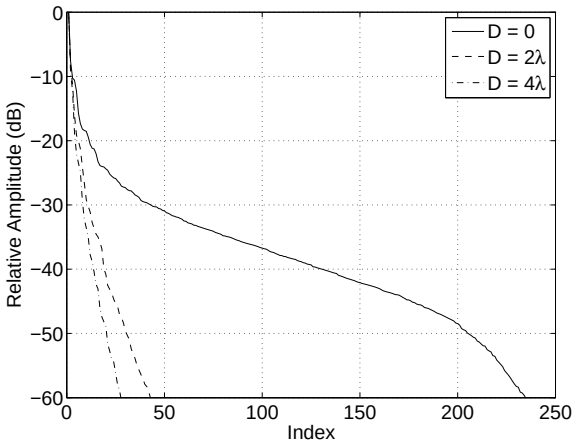


Figure 9.2: Singular Values versus Group Separation D

9.1.1 Limitations of Using SVD For Compression

Assuming that a block structure for the MoM matrix exists, a naive approach for compressing the off-diagonal blocks would be to simply fill and compress them using the SVD. However, as matrix filling is a time consuming operation and the SVD itself scales as $O(N^3)$, this would be computationally infeasible for larger problems. The ACA, introduced in Section 9.2, alleviates these issues for large problem sizes. In the following chapters, we will revisit the SVD in our discussion of MLACA and FMM, as for smaller group sizes it is often easier and more efficient to apply it directly.

9.2 Adaptive Cross Approximation

The Adaptive Cross Approximation, first presented by Bebendorf in [2], efficiently computes a compressed representation of a rank-deficient matrix block $\mathbf{Z}_{m\times n}$ that satisfies the condition

$$\|\mathbf{R}_{m\times n}\| = \|\mathbf{Z}_{m\times n} - \tilde{\mathbf{Z}}_{m\times n}\| \leq \epsilon\|\mathbf{Z}_{m\times n}\| \,, \tag{9.4}$$

where $\mathbf{R}$ is the error matrix and $\|\mathbf{Z}\|$ is the matrix Frobenius norm. It is a simple, method-agnostic, fast rank-revealing algorithm which iteratively constructs the product matrices $\mathbf{Z}_u$ and $\mathbf{Z}_v$ in (9.3) by selecting only k rows and columns from $\mathbf{Z}$, until the residual norm falls below a chosen threshold τ_{ACA}. Thus for small k, only a small fraction of $\mathbf{Z}$ needs to be computed, with final storage requirements for $\mathbf{Z}_u$ and $\mathbf{Z}_v$ comprising only $k(m+n)$, versus mn for $\mathbf{Z}$.

The ACA was first applied to electromagnetic problems in [3], and the procedure from that paper is summarized in Algorithm 6 with some modifications. In this algorithm, $\mathbf{I} = [I_1, \ldots, I_m]$ and $\mathbf{J} = [J_1, \ldots, J_n]$ are arrays containing the indexes of selected rows and columns of $\mathbf{Z}_{m\times n}$. The vector $\mathbf{u}_k$ is the k^{th} column of the matrix $\mathbf{Z}_u$ and $\mathbf{v}_k$ is the k^{th} row of the matrix $\mathbf{Z}_v$. $\tilde{\mathbf{R}}(I_1, :)$ and $\tilde{\mathbf{R}}(:, J_1)$ indicate the ${I_1}^{\text{th}}$ row and ${J_1}^{\text{th}}$ column of $\tilde{\mathbf{R}}$, respectively. $\tilde{\mathbf{Z}}^{(k)}$ comprises the matrix $\tilde{\mathbf{Z}}$ at the k^{th} iteration. Step 12 as originally presented in [3] had an error, and has been corrected following [4].

We note that the convergence criteria in step 13 is based on an *estimate* of the norm and is not completely accurate. Though this author has found these criteria to work well in practice, these criteria were studied further in [5] and [6], where some alternatives are proposed.

There are two immediately obvious advantages to the ACA:

1. The ACA iteratively assembles $\mathbf{Z}_u$ and $\mathbf{Z}_v$ by adding more rows and columns until an acceptable error threshold is reached. The ACA itself has relatively low computational overhead.

2. Complete knowledge of $\mathbf{Z}$ is not necessary. Only k rows and columns must be computed.

9.2.1 Modifications

Based on our experience with the ACA in practice, we have made several modifications to the algorithm as presented originally.

9.2.1.1 Initialization

When originally initializing the ACA in [3], the first row of $\mathbf{Z}$ was used. However, this approach fails if that row is zero or the entire block $\mathbf{Z}$ is zero, which is possible for problems having multiple dielectric regions or certain geometric configurations. During matrix filling, we will have *a priori* knowledge of some of these zero blocks, and these can be zeroed explicitly without doing any work (see Section 9.6.1.1). However, during the LU factorization some blocks will no longer be zero due to fill-in, and each row in those blocks must be tested explicitly.

Algorithm 6 Adaptive Cross Approximation

1: **procedure** INITIALIZATION
2: Set $k = 1$, $\tilde{\mathbf{Z}} = 0$
3: Find a nonzero row $\mathbf{Z}(I_1, :)$ of $\mathbf{Z}$. If none are found, ACA is done.
4: **end procedure**
5: **procedure** ITERATION
6: Update the $I_k{}^{\text{th}}$ row of the error matrix: $\tilde{\mathbf{R}}(I_k, :) = \mathbf{Z}(I_k, :) - \sum_{l=1}^{k-1} (\mathbf{u}_l)_{I_k} \mathbf{v}_l$
7: If all $\|\tilde{\mathbf{R}}(I_k, j)\| = 0, j \neq J_1, \ldots, J_{k-1}$, ACA is done.
8: Find the k^{th} column index $J_k : |\tilde{\mathbf{R}}(I_k, J_k)| = \max_j(|\tilde{\mathbf{R}}(I_k, j)|), j \neq J_1, \ldots, J_{k-1}$
9: $\mathbf{v}_k = \tilde{\mathbf{R}}(I_k, :)/\tilde{\mathbf{R}}(I_k, J_k)$
10: Initialize the $J_k{}^{\text{th}}$ column of the error matrix: $\tilde{\mathbf{R}}(:, J_k) = \mathbf{Z}(:, J_k) - \sum_{l=1}^{k-1} (\mathbf{v}_l)_{J_k} \mathbf{u}_l$
11: $\mathbf{u}_k = \tilde{\mathbf{R}}(:, J_k)$
12: $\|\tilde{\mathbf{Z}}^{(k)}\|^2 = \|\tilde{\mathbf{Z}}^{(k-1)}\|^2 + 2\text{Re}\{ \sum_{j=1}^{k-1} (\mathbf{u}_j^H \mathbf{u}_k)(\mathbf{v}_j^H \mathbf{v}_k) \} + \|\mathbf{u}_k\|^2 \|\mathbf{v}_k\|^2$
13: Convergence check: If $\|\mathbf{u}_k\| \|\mathbf{v}_k\| \leq \tau_{ACA} \|\tilde{\mathbf{Z}}^{(k)}\|$, ACA is done.
14: Find the next row index $I_{k+1} : |\tilde{\mathbf{R}}(I_{k+1}, J_k)| = \max_i(|\tilde{\mathbf{R}}(i, J_k)|), i \neq I_1, \ldots, I_k$
15: $k = k + 1$
16: **end procedure**

9.2.1.2 Early Termination

A second modification was added in step 7. During the ACA iteration, the values in $\tilde{\mathbf{R}}(I_k, :)$ computed in step 6 can become very small. As the search in step 8 is done from an ever decreasing pool of unused columns, it is possible that the largest available $\tilde{\mathbf{R}}(I_k, J_k) \approx 0$ and the operation in step 9 will fail. Thus, if this condition is reached, the ACA algorithm is terminated.

9.2.1.3 Pathological Failure Case

A case where the ACA fails was identified when computing an off-diagonal matrix block for a dielectric cube. This occurred when the source and testing groups comprised co-planar groups of facets on a cube face. For this block, (3.187) takes on the form

$$\mathbf{Z} = \begin{bmatrix} Z^{EJ} & Z^{EM} \\ Z^{HJ} & Z^{HM} \end{bmatrix} = \begin{bmatrix} Z^{EJ} & 0 \\ 0 & Z^{HM} \end{bmatrix}, \tag{9.5}$$

and (3.189) and (3.190) evaluate to zero due to the coplanar geometry. The failure occurs because the ACA does not know which rows are zero. During initialization, the first-row selection in step 3 will simply choose the first row of $\mathbf{Z}$, whose nonzero portion comes from sub-block Z^{EJ}. As a consequence, the column search in step 8 will only select columns from this nonzero portion, whose nonzero content is again from Z^{EJ} only. Thus, the ACA will *never* select rows and columns with elements

from sub-block Z^{HM}. A similar condition would occur if the ACA was initialized with a row from the bottom part of $\mathbf{Z}$, where elements from sub-block Z^{EJ} would now be excluded.

Our solution is to *force* the ACA to select content from both Z^{EJ} and Z^{HM}. We first rearrange $\mathbf{Z}$ so it has the block structure in (9.5). This is done during the clustering and basis mapping step (Section 9.3) by ordering all electric basis functions before magnetic basis functions in the mapping for each block. We then pass a hint to the ACA driver routine, containing the number of magnetic basis functions in the source and testing groups. If these are nonzero, this driver will alternate between two instances of the ACA, which collectively add to $\mathbf{Z}_u$ and $\mathbf{Z}_v$ but are restricted to choosing only rows from the upper or lower parts of $\mathbf{Z}$. As the number of basis functions of each type may differ, as in the case of composite geometry, it is possible that one instance will run out of rows before the other. In this case, the remaining instance continues normally until convergence is reached.

9.2.2 QR/SVD Recompression

The columns and rows of the ACA factors $\mathbf{Z}_u$ and $\mathbf{Z}_v$ are not strictly orthonormal. Thus, significant additional compression can be obtained by applying the QR/SVD recompression method prescribed in [7], which requires QR decompositions of $\mathbf{Z}_u$ and $\mathbf{Z}_v$ and an SVD of size $k \times k$. This method begins by computing

$$\mathbf{Z}_u^{m\times k} = \mathbf{Q}_u^{m\times k}\mathbf{R}_u^{k\times k} \tag{9.6}$$

and

$$\mathbf{Z}_v^{H,n\times k} = \mathbf{Q}_v^{n\times k}\mathbf{R}_v^{k\times k}. \tag{9.7}$$

We then compute the SVD of the product

$$\mathbf{R}_u^{k\times k} \times \mathbf{R}_v^{H,k\times k} = \mathbf{U}^{k\times k}\mathbf{\Sigma}^{k\times k}\mathbf{V}^{*,k\times k}\,, \tag{9.8}$$

and we then determine a new effective rank $r \leq k$ by retaining all singular values in $\mathbf{\Sigma}$ where

$$diag(\mathbf{\Sigma}) \geq \tau_{SVD}\mathbf{\Sigma}(1,1)\,. \tag{9.9}$$

Thus, 9.8 becomes

$$\mathbf{R}_u^{k\times k} \times \mathbf{R}_v^{H,k\times k} = \mathbf{U}^{k\times r}\mathbf{\Sigma}^{r\times r}\mathbf{V}^{*,r\times k}\,. \tag{9.10}$$

We then compute the recompressed versions of $\mathbf{Z}_u$ and $\mathbf{Z}_v$ as

$$\mathbf{Z}_u^{m\times r} = \mathbf{Q}_u^{m\times k}\mathbf{U}^{k\times r}\,, \tag{9.11}$$

and

$$\mathbf{Z}_v^{r\times n} = \mathbf{\Sigma}^{r\times r}\left[\mathbf{Q}_v^{n\times k}\mathbf{R}_v^{H,k\times r}\right]^H. \tag{9.12}$$

Note that the matrix $\mathbf{\Sigma}$ may be multiplied with either $\mathbf{U}$ or $\mathbf{V}$ to yield $\mathbf{Z}_u$ or $\mathbf{Z}_v$. For the regular ACA, either form may be used, however both will be required for the MLACA described later in Section 10.1.1.

After the ACA algorithm has generated an initial estimate of $\mathbf{Z}_u$ and $\mathbf{Z}_v$, this recompression step is invoked immediately to compress them further. Extensive testing has shown that good results can be achieved with a tolerance $\tau_{SVD} \approx 10\ \tau_{ACA}$, and for practical problems, the QR/SVD recompression typically yields additional compression of 20 to 30 percent over the ACA alone. Though this step does add more computational overhead, this is amortized later during the block LU factorization as the off-diagonal block matrices are smaller and their products require fewer total operations.

9.3 Clustering Techniques

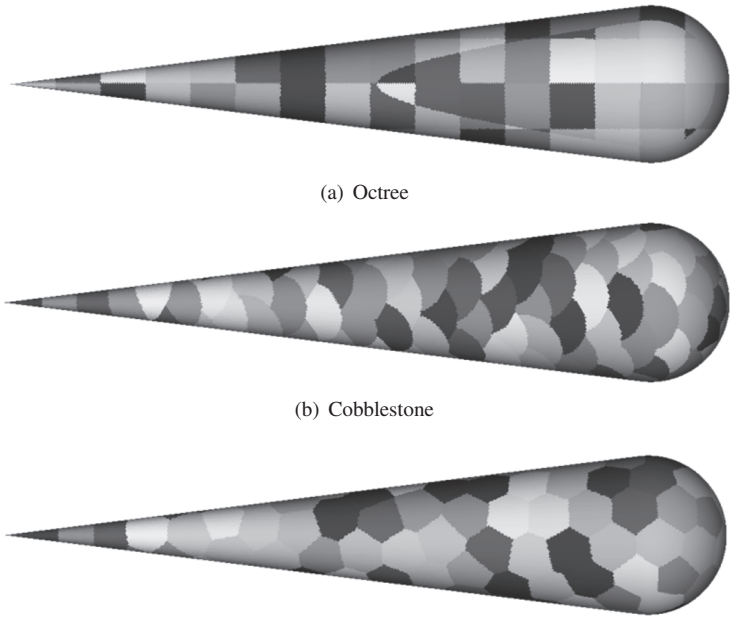
In the present treatment we limit the discussion to RWG basis functions, which are assigned to triangle edges. Following the full matrix approach from Chapter 8, where after all triangles have been analyzed, junctions found and basis functions assigned, we are left with a single, large group of basis functions. We now wish to partition these basis functions into M spatially localized groups, and there are several potential clustering algorithms to choose from. The balanced binary tree and octree use axis-aligned bounding boxes to recursively divide the target's bounding volume, and a cobblestone algorithm was used in [8] to cluster groups for the ACA. However, this author's experience has shown that a K-means clustering algorithm yields groups with the most consistent size, and better overall compression and performance.

Given N points $(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$, K-means attempts to partition these points into M groups $(S_1, S_2, \ldots, S_M)$ by minimizing the inter-cluster sum of squares, given by

$$\sum_{i=1}^{M} \sum_{\mathbf{r}_j \in S_i} \|\mathbf{r}_j - \boldsymbol{\mu}_i\|^2, \tag{9.13}$$

where $\boldsymbol{\mu}_i$ is the mean of group S_i. Finding the global optimum is NP-hard, however it can be implemented in practice as an iterative algorithm. Given a total number of basis functions N and a number of groups M, the algorithm is initialized by choosing at random the midpoints of M surface edges as the initial centroid for each group. Alternatively, a target group size N_g could be specified, and the number of groups chosen as approximately $M = N/N_g$. At each iteration, edges are assigned to groups based on the minimum squared distance of their midpoint from each group's centroid in (9.13). The centroids of each group are then recomputed using the newly assigned points, and the process repeated. For most practical problems, it was observed that the groups do not change much past 30 or so iterations. Though the amount of work grows significantly per iteration with the number of basis functions, the assignment step is easily parallelized.

Figure 9.3 shows the results of the octree, cobblestone, and K-means algorithms applied to the EMCC cone-sphere target described in Section 7.9.2.3. The surface mesh has 79496 facets and 119244 edges, and each algorithm has a target group size of 2000 edges. The octree and cobblestone clustering in Figures 9.3a and 9.3b clearly yield groups having a greater overall variation in size than the K-means clustering in Figure 9.3c.



(a) Octree

(b) Cobblestone

(c) K-means

Figure 9.3: Different Clustering Techniques

Though K-means clustering was found to perform the best of the alternatives for this use case, in some cases it still yields groups having a number of basis functions much larger or smaller than the initial target and final average. However, it is the size of the largest group N_{max} that is of most concern, as the largest diagonal block will be of size $N_{max} \times N_{max}$. This quantity also affects the size of many temporary workspaces, which must not grow too large. This is of particular concern when operating on GPUs, which often have a much more limited memory space than the host system. Thus, for a group whose size exceeds the original target by more than 50 percent, it is split into two new groups via K-means. Finally, groups are sorted into ascending order by size, which allows for a more efficient use of workspace memory as the LU factorization moves along the diagonal. We will discuss CPU and GPU workspace management in more detail in Sections 9.6.2.1 and 9.6.2.2.

Once the groups have been assembled, a global basis function remapping is performed. This is a simple loop over all groups, where the electric basis functions in a group are added to the map first, followed by any magnetic basis functions (see Section 9.2.1.3). Once finished, the basis functions in each group will have global linear indexing, and the new map is used to reorder the original indexing assignments made elsewhere.

9.3.1 Target Group Size For ACA

Proper selection of the target group size N_g for the ACA is based on several factors. As the diagonal blocks are not compressed in ACA and are stored in full form, N_g cannot be made too large or the memory requirements for the diagonals will be too large. Similarly, the performance of the Level 3 BLAS matrix-matrix product `cgemm` in optimized BLAS libraries such as Intel MKL and NVIDIA cuBLAS grows with increasing block size, but will eventually plateau.

Conversely, we cannot make N_g too small, or the compressibility of the off-diagonal blocks will suffer, and we will not exploit the full potential of the optimized BLAS. Thus, our experience has shown the most favorable N_g for the ACA on current hardware lies between 2000 and 2500 unknowns. This balances the compression of off-diagonal blocks, the storage requirement of diagonal blocks, and the efficiency of block operations. We will consider the benefits of larger groups later in Section 10.4.2.1, where we compare the ACA to the MLACA.

9.4 LU Factorization of ACA-Compressed Matrix

After the block MoM matrix has been filled, with off-diagonal blocks stored in compressed form, the matrix system must be solved. In [3] the system was iteratively solved using a preconditioned GMRES algorithm. It was then shown in [8] that if a block LU factorization is applied to the compressed system matrix, the off-diagonal blocks of the **LU** matrix are also compressible. This approach, referred to herein as the "single-level ACA", is particularly suited to efficient computer implementation, as block updates in the LU factorization (and RHS solutions) are performed in rank-reduced form using a compressed matrix representation. This approach is described in detail in this section.

9.4.1 ACA-Compressed Block LU Factorization

Consider an ACA-compressed system matrix, written as

$$\mathbf{Z} = \begin{bmatrix} \mathbf{Z}_{11} & [\mathbf{Z}_u\mathbf{Z}_v]_{12} & [\mathbf{Z}_u\mathbf{Z}_v]_{13} \\ [\mathbf{Z}_u\mathbf{Z}_v]_{21} & \mathbf{Z}_{22} & [\mathbf{Z}_u\mathbf{Z}_v]_{23} \\ [\mathbf{Z}_u\mathbf{Z}_v]_{31} & [\mathbf{Z}_u\mathbf{Z}_v]_{32} & \mathbf{Z}_{33} \end{bmatrix}, \tag{9.14}$$

where a 3×3 block matrix is used for illustrative purposes. In practice, the matrix may have hundreds of block rows and columns. Assuming the off-diagonal blocks of the **LU** matrix are also compressible, it can be written similarly as

$$[\mathbf{LU}] = \begin{bmatrix} [\mathbf{LU}]_{11} & [\mathbf{U}_u\mathbf{U}_v]_{12} & [\mathbf{U}_u\mathbf{U}_v]_{13} \\ [\mathbf{L}_u\mathbf{L}_v]_{21} & [\mathbf{LU}]_{22} & [\mathbf{U}_u\mathbf{U}_v]_{23} \\ [\mathbf{L}_u\mathbf{L}_v]_{31} & [\mathbf{L}_u\mathbf{L}_v]_{32} & [\mathbf{LU}]_{33} \end{bmatrix}. \tag{9.15}$$

In Section 4.1.3, a right-looking block LU factorization for a general block matrix was discussed. One of the core elements in this algorithm was that as it progresses down the diagonal to the right, blocks in the trailing sub-matrix are continuously updated. However, any update operation made to a *compressed* block first requires that block to be expanded (uncompressed) back to full size, the update performed, and the updated block recompressed. This would be computationally inefficient, and approximation errors would be compounded by repeated recompression of blocks. However, a simple reordering of operations will make this much more efficient for a compressed matrix. Instead of iteratively updating the trailing sub-matrix, at each step we perform a total update of the diagonal and its trailing row and column blocks, using completed blocks from previous rows and columns. In this way, the off-diagonal blocks are recompressed only once during the factorization.

The modified approach is summarized in Algorithm 7, where the matrix products with $\mathbf{U}_{bb}^{-1}$ and $\mathbf{L}_{bb}^{-1}$ comprise a triangular back-substitution. Note that in steps 2, 5 and 7, the updates are made to full-size matrix blocks. In step 2, the diagonal $\mathbf{Z}_{bb}$ is already full, and in steps 5 and 7, $\mathbf{Z}_{sb}$ and $\mathbf{Z}_{bs}$ are first expanded to full size. However, the block updates in steps 2, 5 and 7 are very fast, as the matrix products are performed in rank-reduced form using the Level-3 BLAS. To illustrate, consider the product of ACA-compressed matrices

$$\mathbf{A}^{m_1\times n_1}\mathbf{B}^{m_2\times n_2} = \mathbf{A}_u^{m_1\times k_1}\mathbf{A}_v^{k_1\times n_1}\mathbf{B}_u^{m_2\times k_2}\mathbf{B}_v^{k_2\times n_2}, \tag{9.16}$$

Algorithm 7 ACA-Compressed Block LU Factorization

1: **for** $b = 1$ to M **do**
2: $\quad \mathbf{A} = \mathbf{Z}_{bb} - \sum_{s=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{bs} [\mathbf{U}_u \mathbf{U}_v]_{sb}$
3: $\quad [\mathbf{LU}]_{bb} = \mathrm{LU}(\mathbf{A})$ (scalar LU factorization)
4: $\quad$ **for** $s = b + 1$ to M **do**
5: $\quad\quad \mathbf{L}_{sb} = \left[[\mathbf{Z}_u \mathbf{Z}_v]_{sb} - \sum_{p=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{sp} [\mathbf{U}_u \mathbf{U}_v]_{pb} \right] \mathbf{U}_{bb}^{-1}$
6: $\quad\quad$ Compress $\mathbf{L}_{sb}$ via ACA and store $[\mathbf{L}_u \mathbf{L}_v]_{sb}$
7: $\quad\quad \mathbf{U}_{bs} = \mathbf{L}_{bb}^{-1} \left[[\mathbf{Z}_u \mathbf{Z}_v]_{bs} - \sum_{p=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{bp} [\mathbf{U}_u \mathbf{U}_v]_{ps} \right]$
8: $\quad\quad$ Compress $\mathbf{U}_{bs}$ via ACA and store $[\mathbf{U}_u \mathbf{U}_v]_{bs}$
9: $\quad$ **end for**
10: **end for**

where $n_1 = m_2$. The innermost product is performed first, which is

$$\mathbf{T}_1^{k_1 \times k_2} = \mathbf{A}_v^{k_1 \times n_1} \mathbf{B}_u^{m_2 \times k_2} \ , \tag{9.17}$$

and so

$$\mathbf{A}^{m_1 \times n_1} \mathbf{B}^{m_2 \times n_2} = \mathbf{A}_u^{m_1 \times k_1} \mathbf{T}_1^{k_1 \times k_2} \mathbf{B}_v^{k_2 \times n_2}. \tag{9.18}$$

Which of the remaining products in (9.18) is performed next depends on $\min(k_1, k_2)$. If $k_1 < k_2$, we compute

$$\mathbf{T}_2^{k_1 \times n_2} = \mathbf{T}_1^{k_1 \times k_2} \mathbf{B}_v^{k_2 \times n_2} \tag{9.19}$$

and then

$$\mathbf{A}^{m_1 \times n_1} \mathbf{B}^{m_2 \times n_2} = \mathbf{A}_u^{m_1 \times k_1} \mathbf{T}_2^{k_1 \times n_2}. \tag{9.20}$$

Otherwise, we compute

$$\mathbf{T}_3^{m_1 \times k_2} = \mathbf{A}_u^{m_1 \times k_1} \mathbf{T}_1^{k_1 \times k_2} \tag{9.21}$$

and then

$$\mathbf{A}^{m_1 \times n_1} \mathbf{B}^{m_2 \times n_2} = \mathbf{T}_3^{m_1 \times k_2} \mathbf{B}_v^{k_2 \times n_2}. \tag{9.22}$$

Note that in practice, $\mathbf{L}_{sb}$ and $\mathbf{U}_{bs}$ are temporary workspaces whose contents are discarded after being compressed by the ACA driver in steps 6 and 8.

9.4.1.1 Compressibility of the LU Matrix

Though the **LU** matrix blocks are compressible, in general they are less compressible than the blocks of the MoM matrix **Z** for the same τ_{ACA}. Thus, during factorization the size of the **LU** matrix will typically increase after each row/column update, requiring extra attention to memory management. It is not uncommon for the final size of the **LU** matrix to be more than double that of the MoM matrix. Experience has also shown that this size increase is much greater for problems involving dielectrics, versus those with only conductors.

9.5 Solution of the ACA-Compressed Matrix System

Monostatic scattering problems often have thousands of incident angles, and solving for each RHS individually can be very time consuming. Let us instead group all the RHS and solution vectors into matrices $\mathbf{B}_{N\times N_{RHS}}$ and $\mathbf{X}_{N\times N_{RHS}}$ where N is the number of unknowns and N_{RHS} is the number of incident angles. Given the 3×3 matrix in (9.15), the linear matrix system has the form

$$\begin{bmatrix} [\mathbf{LU}]_{11} & [\mathbf{U}_u\mathbf{U}_v]_{12} & [\mathbf{U}_u\mathbf{U}_v]_{13} \\ [\mathbf{L}_u\mathbf{L}_v]_{21} & [\mathbf{LU}]_{22} & [\mathbf{U}_u\mathbf{U}_v]_{23} \\ [\mathbf{L}_u\mathbf{L}_v]_{31} & [\mathbf{L}_u\mathbf{L}_v]_{32} & [\mathbf{LU}]_{33} \end{bmatrix} \begin{bmatrix} [\mathbf{X}_u\mathbf{X}_v]_1 \\ [\mathbf{X}_u\mathbf{X}_v]_2 \\ [\mathbf{X}_u\mathbf{X}_v]_3 \end{bmatrix} = \begin{bmatrix} [\mathbf{B}_u\mathbf{B}_v]_1 \\ [\mathbf{B}_u\mathbf{B}_v]_2 \\ [\mathbf{B}_u\mathbf{B}_v]_3 \end{bmatrix}. \tag{9.23}$$

Note that the block structure of the system matrix, resulting from the ACA clustering, also defines the block structures of $\mathbf{X}$ and $\mathbf{B}$. It was shown in [8] that when there are many closely spaced incident angles, $\mathbf{X}$ and $\mathbf{B}$ are also highly compressible. Thus, we can solve all right-hand sides simultaneously using the block back-substitution in Algorithm 8. As in the LU factorization, all matrix products are in rank-reduced form, and $\tilde{\mathbf{B}}_b$ and $\mathbf{X}_b$ each comprise a temporary workspace that is discarded after compression.

Algorithm 8 ACA-Compressed Block-RHS Solution

1: Given $\mathbf{LU}\,\mathbf{X} = \mathbf{B}$, first solve $\mathbf{UX} = \mathbf{L}^{-1}\mathbf{B} = \tilde{\mathbf{B}}$:
2: **for** $b = 1$ to M **do**
3: $\quad\tilde{\mathbf{B}}_b = \mathbf{L}_{bb}^{-1}\left[[\mathbf{B}_u\mathbf{B}_v]_b - \sum_{s=1}^{b-1}[\mathbf{L}_u\mathbf{L}_v]_{bs}[\mathbf{B}_u\mathbf{B}_v]_s\right]$
4: $\quad$Compress $\tilde{\mathbf{B}}_b$ via ACA and store updated $[\tilde{\mathbf{B}}_u\tilde{\mathbf{B}}_v]_b$
5: **end for**
6: Then solve $\mathbf{X} = \mathbf{U}^{-1}\left[\mathbf{L}^{-1}\mathbf{B}\right]$
7: **for** $b = M$ to 1 **do**
8: $\quad\mathbf{X}_b = \mathbf{U}_{bb}^{-1}\left[[\tilde{\mathbf{B}}_u\tilde{\mathbf{B}}_v]_b - \sum_{s=b+1}^{M}[\mathbf{U}_u\mathbf{U}_v]_{bs}[\mathbf{X}_u\mathbf{X}_v]_s\right]$
9: $\quad$Compress $\mathbf{X}_b$ via ACA and store $[\mathbf{X}_u\mathbf{X}_v]_b$
10: **end for**

As with the $\mathbf{LU}$ matrix, the blocks of the solution matrix $\mathbf{X}$ are generally less compressible than those of the RHS matrix $\mathbf{B}$. However, as the overall size of $\mathbf{X}$ is typically far less than the $\mathbf{LU}$ matrix, this is not usually a practical limitation.

If there is only a single right-hand side, the RHS blocks in (9.23) will have a single column. Thus, the matrix system reduces to

$$\begin{bmatrix} [\mathbf{LU}]_{11} & [\mathbf{U}_u\mathbf{U}_v]_{12} & [\mathbf{U}_u\mathbf{U}_v]_{13} \\ [\mathbf{L}_u\mathbf{L}_v]_{21} & [\mathbf{LU}]_{22} & [\mathbf{U}_u\mathbf{U}_v]_{23} \\ [\mathbf{L}_u\mathbf{L}_v]_{31} & [\mathbf{L}_u\mathbf{L}_v]_{32} & [\mathbf{LU}]_{33} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{bmatrix}, \tag{9.24}$$

and the back-substitution is now performed as in Algorithm 9. Note that when performing the matrix-vector products in steps 3 and 4, it is more efficient to form the product with the $\mathbf{V}$ matrix first. That is, for the product $\mathbf{A}_u\mathbf{A}_v\mathbf{b}$, first compute $\mathbf{c} = \mathbf{A}_v\mathbf{b}$ and then $\mathbf{A}_u\mathbf{c}$.

Algorithm 9 Single RHS Solution

Given $\mathbf{LU}\ \mathbf{x} = \mathbf{b}$, first solve $\mathbf{Ux} = \mathbf{L}^{-1}\mathbf{b} = \tilde{\mathbf{b}}$:
for $b = 1$ to M **do**
    $\tilde{\mathbf{b}}_b = \mathbf{L}_{bb}^{-1}\left[\mathbf{b}_b - \sum_{s=1}^{b-1}[\mathbf{L}_u\mathbf{L}_v]_{bs}\ \tilde{\mathbf{b}}_s\right]$
end for
Then solve $\mathbf{x} = \mathbf{U}^{-1}\left[\mathbf{L}^{-1}\mathbf{b}\right]$
for $b = M$ to 1 **do**
    $\mathbf{x}_b = \mathbf{U}_{bb}^{-1}\left[\tilde{\mathbf{b}}_b - \sum_{s=b+1}^{M}[\mathbf{U}_u\mathbf{U}_v]_{bs}\ \mathbf{x}_s\right]$
end for

9.6 Software Implementation Notes

In this section we discuss practical software implementation of the ACA. In particular, we will describe several elements of how this was done in our *Serenity* MoM solver. Note that the material presented here comprises only programming suggestions, based on the author's own implementation.

9.6.1 Software Class Support

In this section we will discuss some of the software classes implemented in our *Serenity* solver in support of ACA operations. As *Serenity* is written in the C++ programming language, these classes make use of that language's object-oriented features, such as inheritance and polymorphism. In particular, the use of virtual functions has enabled functionality that would have been difficult or impossible to implement in other languages, while reducing the total codebase size.

9.6.1.1 Element Engine Class

The `ElementEngine` class is responsible for computing matrix elements in the MoM and RHS sub-blocks. At a minimum, it must be capable of computing a full diagonal block, and selected rows and columns of any off-diagonal matrix block or RHS block. For testing and debugging purposes, our implementation supports these operations for all matrix blocks. This was critical in identifying and correcting the issue identified in Section 9.2.1.3.

During initialization, the element engine performs several key bookkeeping operations that facilitate these capabilities. For each basis function group, it determines the dielectric regions in which those basis functions have support, and maps each basis function to its corresponding triangle pair. A list of those triangles is then constructed for each region, where each entry references a triangle and edges having basis functions that exist *only in that region* and *only in that group*.

Filling full matrix blocks in their entirety is carried out in a straightforward manner, following very closely the treatment in Chapter 8 and in particular Section 8.7. A loop over source and testing triangle pairs is still performed, using only those triangles and corresponding edges that were added to the lists for the source and testing group.

Filling of individual rows and columns comprises a slightly different operation. When filling row r, basis function r is a testing function and the two testing triangles are obtained from that function's triangle pair mapping. An innermost loop is then performed over all triangles in the source group. When filling column c, basis function c is a source function and the source triangles are similarly obtained from that function's triangle mapping. An outermost loop is then performed over all triangles in the testing group.

Because the element engine maps a group's basis functions to triangles and triangles with edges having basis functions on a *per region* basis, this information is used to determine whether a matrix block is zero before any computation is performed. In this case, the block would be explicitly set to zero without needing to invoke the ACA driver.

A significant amount of code from our original full matrix *Serenity* solver was reused in the element engine. Its routine for computing diagonal matrix blocks, for example, is almost identical to the original full matrix fill routine.

9.6.1.2 Matrix Classes

In prior chapters, little attention was paid to a software class structure for working with the MoM matrix system. This is because one is not really needed for smaller problems that do not have a block structure. In this case, the programmer can simply allocate a single, linear memory buffer for the matrix and address that array directly to perform all matrix operations. However, with a compressed block matrix system this is no longer possible, and a more elaborate matrix class system is needed.

9.6.1.2.1 Abstract Matrix Class `AbstractMatrix` is an abstract class that defines the virtual functions (interface) used by the ACA driver when compressing a matrix block $\mathbf{Z}_{m\times n}$. These functions are re-implemented by all classes that derive from `AbstractMatrix`. These functions are:

1. `getNumRows()`: Returns the number of rows m in $\mathbf{Z}$.
2. `getNumColumns()`: Returns the number of columns n in $\mathbf{Z}$.
3. `getRow(`r`)`: Returns row r of $\mathbf{Z}$.
4. `getColumn(`c`)`: Returns column c of $\mathbf{Z}$.
5. `getNumStored()`: Returns the total number of data elements stored.

This is the most critical class in our software implementation. The ACA driver requires only a reference to an `AbstractMatrix` object, which provides the selected rows and columns of $\mathbf{Z}$ via the abstract interface.

9.6.1.2.2 Full Matrix Class The `FullMatrix` class is used to store a regular, uncompressed matrix block of size $m \times n$, and `getNumStored()` returns mn. It returns rows and columns by simply copying them from its internal storage.

9.6.1.2.3 ACA Matrix Class The `ACAMatrix` class stores the $\mathbf{U}$ and $\mathbf{V}$ components of a compressed matrix block, which are stacked together into a single array in memory. `getNumStored()` returns $(m + n)k$. The ACA driver routine, discussed in Section 9.2, constructs matrix blocks of this type. Note that this class does not necessarily need to implement `getRow` or `getColumn`, however doing so can be very useful for debugging purposes.

9.6.1.2.4 BlockMatrix Class The `BlockMatrix` class is the master, high-level class that stores and manages M block sub-rows and N block sub-columns. It provides a function `getBlock(`sr, sc`)` which returns a pointer to the matrix block on sub-row sr and sub-column sc, which are of type `FullMatrix` or `ACAMatrix`. These

pointers are simply stored in a flat array of size $M \times N$, stored in row-major format. For the `BlockMatrix` class, `getNumStored()` returns the sum of that function for all stored matrix blocks. It also provides a `getPercentCompressed()` function which compares the result of `getNumStored()` against mn.

9.6.1.2.5 ACA Translator Class As mentioned previously, the ACA driver and underlying ACA routines are only provided as a reference to an `AbstractMatrix` class, which is queried to obtain the matrix dimensions and selected rows and columns. However, the source of these rows and columns differs depending on which step of the solution is being performed. During LU factorization, the source comprises a `FullMatrix` class which simply copies the data from internal storage. However, when filling the system matrix or a right-hand side matrix, the rows and columns are computed on the fly by the element engine, which needs additional information regarding the source and testing groups, and the incident angles to be considered. The `ACAMatrixTranslator` and `ACARHSTranslator` classes interface directly with the element engine, providing all necessary extra information, while presenting the same `AbstractMatrix` interface to the ACA driver.

9.6.2 Shared Memory Processing

On shared memory systems, *Serenity* performs all operations in parallel using POSIX Threads. *Serenity* implements two thread classes: one for operation on the CPU, and one for operation on the GPU. These classes have many similarities, and are discussed in more detail in this section.

9.6.2.1 ACA CPU Thread Class

The `ACACPUThread` class is used on systems where no GPU is available, and performs all calculations on the CPU. For matrix operations, *Serenity* uses the CPU-optimized BLAS from the Intel MKL.

9.6.2.1.1 Workspace Management As modern computer systems have multi-core CPUs supporting a large number of threads, the scratch space used by each thread must be managed carefully. Specifically, an attempt should be made to keep these spaces as small as possible, and to avoid reallocations which lead to memory fragmentation over time. Our strategy is to assign to each thread resizable work arrays, which are initially empty. Prior to using an array, it is reallocated to a larger size if necessary, based on the space currently needed. As a result, resizing of a workspace occurs only a few times until it reaches a certain size, beyond which resizing is infrequent and memory fragmentation minimized.

9.6.2.1.2 Matrix Filling When filling the system matrix, each thread computes matrix blocks asynchronously from other threads, choosing from blocks that are not yet filled. This step requires only a mutex-protected counter; no other coordination between threads is needed. For approximating off-diagonal blocks, the ACA driver is

provided workspaces to store the rows and columns computed by the element engine, accumulate **U** and **V**, and perform the QR/SVD recompression once the ACA is completed. The size of these arrays is increased on demand as needed.

9.6.2.1.3 LU Factorization The LU factorization requires more coordination between threads as the factorization moves along the diagonal in Algorithm 7. A single thread performs the diagonal update and the scalar LU factorization in step 3. As this updated diagonal is required for the row and column updates, a barrier is used to restrict progress to step 4 until this thread is finished. Threads are now free to perform updates of the trailing row and column blocks (steps 4 to 9), doing so asynchronously by choosing from blocks that are not yet updated. Another barrier is used at the end of this loop so that all rows and columns are fully updated before moving to the next diagonal. During the update, each thread will need workspaces for storing $\mathbf{L}_{sb}$ and $\mathbf{U}_{bs}$, as well as the intermediate products $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_3$ from Section 9.4.1. As before, these arrays are resized as needed.

9.6.2.1.4 Block RHS Solution Filling of the RHS blocks is also done in parallel, with each thread choosing from blocks that are not yet filled. The solution of each RHS in Algorithm 8 is performed by a single thread. For radar cross section problems this comprises two threads, one for each of the vertical and horizontal polarizations. Each thread will need workspaces for storing $\mathbf{B}_b$ and $\mathbf{X}_b$, as well as the intermediate products $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_3$ previously mentioned in Section 9.4.1.

9.6.2.2 ACA GPU Thread Class

The `ACAGPUThread` class derives from the `ACACPUThread` class, and shares much of the same code, but performs a majority of the matrix filling, factorization and RHS solution on the GPU. Our implementation is designed to operate with NVIDIA GPUs using the NVIDIA CUDA programming language and the GPU-optimized cuBLAS library. Operating on the GPU requires additional programming considerations, as it has its own on-board memory space. Data to be processed must first be uploaded to the GPU (device) memory, any processing done, and the results downloaded back to the CPU (host) memory. Most GPUs are connected to the host system through an expansion interface, most often a PCI-E expansion slot. As these have a much lower bandwidth than the memory bus, the programmer must carefully manage data transfer to and from the GPU so that it remains busy and not stalled, waiting for data. Fortunately, NVIDIA GPUs support simultaneous upload and download of data, and overlap of different computations on the device via independent compute "streams". *Serenity* leverages this capability, with GPU threads operating independently from the others, and a minimal amount of coordination between steps. On systems with multiple GPUs, *Serenity* schedules an equal number of GPU threads on the host for each device. For systems supporting a total of N_t host threads and N_{GPU} GPUs, this is N_t/N_{GPU} threads per device.

9.6.2.2.1 GPU Workspace Management Each `ACAGPUThread` needs a workspace in device memory. This memory differs from host memory in that it is typically of limited size, so it must be managed carefully to avoid depletion. Our strategy is to allocate one large, contiguous block of device memory during initialization, and then assign portions of this block to each thread as needed. As there is no reallocation, workspace assignments must remain fixed during certain operations. Additionally, as each thread operates asynchronously, there is no *a priori* knowledge regarding which portions of an operation will be completed by a thread. Thus, the workspaces assigned to each thread must be sized based on the maximum possible space that *may* be needed. As a consequence, available workspace on the device may not be sufficient to support all threads, and some may go idle. Our approach makes all possible effort to minimize idle threads.

9.6.2.2.2 Matrix Filling NVIDIA GPUs have Streaming Multiprocessor (SM) units, each of which contains many CUDA processing cores having floating point and integer processors. These CUDA cores are optimized to perform simple, parallel mathematical operations and do not perform instruction prefetching or speculative execution like a general-purpose CPU. Therefore, they are most efficient when executing the same operations in parallel, and complex code with excessive branching and divergence should be avoided. Our matrix fill implementation computes all far matrix elements on the GPU, as that code comprises loops over source and testing triangles and has very few conditional logic statements. As the routines for computing near and self terms have much more branching logic, they are computed by the host thread on the CPU. When computing a row, column or diagonal block, both operations are overlapped, with the GPU and CPU operating simultaneously. When the CPU is finished, it synchronizes with the GPU and downloads the results, which are then combined with the host's results. Each thread requires a device workspace of size $N_{max} \times N_{max}$ for matrix filling, sufficient for storing of all possible rows and columns and diagonal blocks.

Our CUDA code is a simplified version of the original CPU far-matrix element code. Information about triangle and edge geometry, as well as triangle-to-basis and basis-to-triangle mapping, are assembled by the element engine and uploaded to the device at initialization. On the device, each CUDA thread assembles a single matrix element via edge pairs instead of face pairs. Though this results in some redundant calculation, there is no inter-thread communication which simplifies the GPU code. For a more in-depth explanation at the code-level, the reader is referred to [9] and [10], where similar approaches to matrix filling were taken. For more detailed information on CUDA programming, the reader should consult the NVIDIA CUDA Programmer's Guide [11].

9.6.2.2.3 LU Factorization The LU factorization on the GPU follows the same steps as on the CPU, however for GPU processing, blocks must be sent to the GPU prior to any operations. Workspaces for these blocks, as well as the intermediate products (see Section 9.4.1) are all located in device memory and must be sized properly.

Diagonal blocks are full, and a single shared workspace of size $N_b \times N_{max}$ is sufficient for storing the diagonal on each device. Each diagonal is updated and LU factored

on both the host and one device by a single thread, following the approach in [12]. On multi-GPU systems, the factored diagonal is then copied to all other devices and remains until the trailing row and column update is complete. We note also that during the update of the diagonal, and trailing rows and columns, threads will repeatedly access the **U** blocks directly above the diagonal and the **L** blocks to its left. Staging these blocks on each device prior to the update avoids having to upload them every time they are needed. Fortunately, the size of these blocks is known beforehand, so the shared storage needed on each device can be sized exactly. However, staging blocks in this way reduces the device workspace left for GPU threads, and the space needed increases as the factorization moves to the right. Thus, if device workspace starts being depleted, staging is disabled.

As the host threads operate asynchronously, it is not known ahead of time which trailing block each thread will update. Therefore, each thread is assigned a block of workspace sufficient for handling any blocks it may encounter. If workspace is not available for all threads, those without will remain idle. However, as the groups were sorted into ascending order by size, workspaces are smallest at the beginning, and grow larger as the factorization progresses down the diagonal. Thus, if any threads are idled, it will occur later in the factorization, rather than at the start.

For each host thread, workspace is needed to store the full-size $\mathbf{L}_{sb}$ and $\mathbf{U}_{bs}$ blocks. As groups are sorted by size, the maximum possible size of these blocks is $N_b \times N_{max}$, increasing to $N_{max} \times N_{max}$ on the last diagonal. Workspace is also needed for both of the product matrices to be multiplied (or one, if the other was staged), and the intermediate products $\mathbf{T}_1$, $\mathbf{T}_2$ and $\mathbf{T}_3$. These can be sized by searching all matrix blocks above and to the right of the diagonal, and all blocks to the left and below the diagonal, and finding the maximum compressed block size C_{max} and the maximum rank k_{max}. Two spaces of size C_{max} are reserved (or one, if staging), as well as a space of size $k_{max} \times k_{max}$ for $\mathbf{T}_1$ and a space of size $N_{max} \times k_{max}$ for $\mathbf{T}_2$ and $\mathbf{T}_3$.

All matrix operations in Algorithm 7 are carried out asynchronously on the GPU using CUDA streams and the cuBLAS library. The host threads are responsible for uploading any matrix blocks not already staged to the device and making the appropriate calls to cuBLAS. They then wait for the block update to complete before downloading the finished block to host memory. ACA compression of the updated blocks (steps 6 and 8) is then performed on the CPU.

9.6.2.2.4 Block RHS Solution The RHS solution on the GPU follows the same steps as on the CPU, with one thread per RHS, and all matrix operations performed on the device. Workspace for each thread must be sufficient to handle products involving the largest **LU** matrix block as well as RHS matrix block. A workspace of size $N_{max} \times N_{max}$ is needed to store the diagonal, and workspace of size $N_{max} \times N_{RHS}$ for storing the full size $\mathbf{B}_b$ and $\mathbf{X}_b$. Let us define C^{max}_{RHS} as the maximum compressed block size and k^{max}_{RHS} the maximum rank of all RHS matrix blocks. Spaces of size C_{max} and C^{max}_{RHS} are needed for storing product matrices, a space of size $k_{max} \times k^{max}_{RHS}$ for $\mathbf{T}_1$ and a space of size $\max(N_{max}, N_{RHS}) \times \max(k_{max}, k^{max}_{RHS})$ for $\mathbf{T}_2$ and $\mathbf{T}_3$.

9.6.3 Distributed Memory Processing

Efficient operation on a distributed memory system requires knowledge about the CPU and network configuration of that system. The implementation discussed in this section was designed for a cluster-type system having multiple compute nodes, each with multi-core CPUs and a large memory pool, connected via a fast network fabric. Communication between compute nodes is done using the OpenMPI message passing library [13], a high-performance implementation of the Message Passing Interface (MPI) standard [14].

Compared to a shared memory system, a distributed memory platform has the benefit of a much larger potential memory pool as well as many more processors. For us to effectively utilize the memory space, we must break up the problem into smaller chunks and distribute them relatively equally across all nodes. However, this means that during processing, each node will require data stored on another node, which must be transmitted across the network. As the network is many times slower than direct memory access on each node, the programmer must analyze and modify core algorithms to minimize data transfer when possible. To effectively utilize the multiple cores on each node, these algorithms must also have the appropriate parallelization strategy, so that threads remain busy and not idle waiting for data. We will next consider modifications to the block LU factorization, block RHS solution, and shared memory implementation that allow for efficient operation on distributed memory systems.

9.6.3.1 Parallelization Strategy

When using MPI, the application is launched simultaneously on all compute nodes, all of which have multiple processing cores. In this context, each process is analogous to a thread in the shared memory implementation. Thus, to make effective use of the cores on each node, there are two valid paths that can be taken in designing the application:

1. A single-threaded program launched multiple times on each node. Parallelization occurs only at the MPI level.

2. A multi-threaded program launched once on each node. Parallelization occurs at the MPI level *and* the process level. This is called a *hybrid* approach.

During development, we experimented with both approaches, and the hybrid approach was found to have the best overall performance. At the process level, all high-level ACA operations are performed by a single thread. Matrix filling is done by a separate pool of work threads, and BLAS functions done in parallel by the Intel MKL. This approach eliminates costly inter-process MPI communication and the same data being stored into memory multiple times on each node. We have also adapted Algorithms 7 and 8 to better suit distributed memory operation. These modifications are discussed in Sections 9.6.3.2 and 9.6.3.3.

9.6.3.2 Block LU Factorization Using MPI

We note that in Algorithm 7, updates of a **U** block to the right of the diagonal require access to all previous **U** blocks in that column. Similarly, updates of an **L** block below

the diagonal require access to all previous **L** blocks in that row. Thus, our strategy is to assign each diagonal and all **U** blocks above that diagonal to individual nodes in a round-robin fashion. In a similar fashion, we assign all **L** blocks to the left of each diagonal to individual nodes in a reverse round-robin scheme. As a result, the only non-local blocks needed during a row/column update are the diagonal and those located above and to the left of the diagonal. During matrix filling, each node fills only local blocks, and the local block matrix structure is sparse. As an example, consider a 7×7 block matrix and 5 MPI processes. The block-to-process map in this case comprises the matrix **M** given by

$$\mathbf{M} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 0 & 1 \\ 3 & 1 & 2 & 3 & 4 & 0 & 1 \\ 2 & 2 & 2 & 3 & 4 & 0 & 1 \\ 1 & 1 & 1 & 3 & 4 & 0 & 1 \\ 0 & 0 & 0 & 0 & 4 & 0 & 1 \\ 4 & 4 & 4 & 4 & 4 & 0 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 1 \end{bmatrix}. \tag{9.25}$$

A modified version of the compressed block LU factorization is presented in Algorithm 10 for MPI processing. For each iteration of the outer loop, all off-diagonal **U** blocks directly above the diagonal and **L** blocks to its left are disseminated to all nodes via broadcasts. The diagonal $\mathbf{Z}_{bb}$ is then updated on the node where it is local, and the updated diagonal is then broadcast to all nodes. The remainder of the update operations follow Algorithm 7, where only local blocks are updated by each node.

9.6.3.3 Block-RHS Solution Using MPI

For the block-RHS solution, RHS blocks local to each node are reconstructed up front and then iteratively updated. To minimize communication and simplify programming, a new mapping is generated to rearrange the **LU** matrix post-factorization. These assign each row to a single node, in a round-robin fashion. Following the example from Section 9.6.3.2, these new mappings would comprise matrices $\mathbf{M}_{LU}$ and $\mathbf{M}_{RHS}$ given by

$$\mathbf{M}_{LU} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \qquad \mathbf{M}_{RHS} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 1 \end{bmatrix}. \tag{9.26}$$

Application of $\mathbf{M}_{LU}$ to the completed **LU** matrix comprises a shuffling step where nodes exchange blocks as necessary. An updated version of the ACA-compressed block-RHS solution is now presented in Algorithm 11 for MPI. We note that the only communications required here are the broadcasts in steps 8 and 23, comprising the most recently completed RHS block.

Algorithm 10 ACA-Compressed Block LU Factorization Using MPI

for $b = 1$ to M **do**
  for $k = 1$ to $b - 1$ **do**
    Broadcast compressed $\mathbf{U}_{kb}$
    Broadcast compressed $\mathbf{L}_{bk}$
  end for
  if $\mathbf{Z}_{bb}$ is local **then**
    $\mathbf{A} = \mathbf{Z}_{bb} - \sum_{s=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{bs} [\mathbf{U}_u \mathbf{U}_v]_{sb}$
    $[\mathbf{LU}]_{bb} = \mathrm{LU}(\mathbf{A})$ (scalar LU factorization)
  end if
  Broadcast $[\mathbf{LU}]_{bb}$
  for $s = b + 1$ to M **do**
    if $\mathbf{L}_{sb}$ is local **then**
      $\mathbf{L}_{sb} = \left[[\mathbf{Z}_u \mathbf{Z}_v]_{sb} - \sum_{p=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{sp} [\mathbf{U}_u \mathbf{U}_v]_{pb} \right] \mathbf{U}_{bb}^{-1}$
      Compress $\mathbf{L}_{sb}$ via ACA and store $[\mathbf{L}_u \mathbf{L}_v]_{sb}$
    end if
    if $\mathbf{U}_{bs}$ is local **then**
      $\mathbf{U}_{bs} = \mathbf{L}_{bb}^{-1} \left[[\mathbf{Z}_u \mathbf{Z}_v]_{bs} - \sum_{p=1}^{b-1} [\mathbf{L}_u \mathbf{L}_v]_{bp} [\mathbf{U}_u \mathbf{U}_v]_{ps} \right]$
      Compress $\mathbf{U}_{bs}$ via ACA and store $[\mathbf{U}_u \mathbf{U}_v]_{bs}$
    end if
  end for
end for

Algorithm 11 ACA-Compressed Block-RHS Solution Using MPI

Given $\mathbf{LU}\,\mathbf{X} = \mathbf{B}$, first solve $\mathbf{UX} = \mathbf{L}^{-1}\mathbf{B}$:
for $b = 1$ to M **do**
  if $\mathbf{B}_b$ is local **then**
    $\mathbf{B}_b = \mathbf{L}_{bb}^{-1}\mathbf{B}_b$
    Compress $\mathbf{B}_b$ via ACA and store updated $[\mathbf{B}_u\mathbf{B}_v]_b$
  end if
  if $b < M$ **then**
    Broadcast compressed $\mathbf{B}_b$
    for $s = b + 1$ to M **do**
      if $\mathbf{B}_s$ is local **then**
        $\mathbf{B}_s = \mathbf{B}_s - [\mathbf{L}_u\mathbf{L}_v]_{sb}[\mathbf{B}_u\mathbf{B}_v]_b$
      end if
    end for
  end if
end for
Next solve $\mathbf{X} = \mathbf{U}^{-1}\left[\mathbf{L}^{-1}\mathbf{B}\right]$
for $b = M$ to 1 **do**
  if $\mathbf{B}_b$ is local **then**
    $\mathbf{X}_b = \mathbf{U}_{bb}^{-1}\mathbf{B}_b$
    Compress $\mathbf{X}_b$ via ACA and store updated $[\mathbf{X}_u\mathbf{X}_v]_b$
  end if
  if $b > 1$ **then**
    Broadcast compressed $\mathbf{X}_b$
    for $s = b - 1$ to 1 **do**
      if $\mathbf{B}_s$ is local **then**
        $\mathbf{B}_s = \mathbf{B}_s - [\mathbf{U}_u\mathbf{U}_v]_{sb}[\mathbf{X}_u\mathbf{X}_v]_b$
      end if
    end for
  end if
end for

9.7 Numerical Examples

In this section we compute the radar cross section of several test articles using the *Serenity* solver code, which implements the ACA algorithm outlined in this chapter. We will consider conducting, dielectric, and coated spheres, several benchmark targets with measured data, and the monoconic RV with dielectric nose. The electrical size will be much larger than those in prior chapters. Triangle meshes are constructed with at least 10 unknowns per wavelength in each dielectric region. The CFIE is applied on closed conductors with $\alpha = 0.5$, and PMCHWT applied to dielectric interfaces. For the ACA, the target group size is $N_g = 2500$ basis functions, and $\tau_{ACA} = 10^{-5}$.

9.7.1 Compute Platform

Computations in this section were carried out on the Dell workstation described previously in Section 8.8.2. This workstation is equipped with dual NVIDIA GTX 1080 Ti GPUs, each having 11 GB of device memory. For matrix filling, near and self matrix elements are computed on the CPU, and far matrix elements on the GPUs. Block updates in the LU factorization, as well as the RHS solution, are all performed on the GPUs.

9.7.2 Adaptive ACA Tolerance

When compressing off-diagonal blocks, the most straightforward approach is to use a single τ_{ACA} for all blocks. However, we know that the amplitudes of interactions between groups decreases with separation, and so it is reasonable to assume that the importance (and the accuracy requirement) of the resulting blocks also decreases with distance.

Consider group A with bounding sphere of diameter d_A centered at $\mathbf{r}_A$, and group B with bounding sphere of diameter d_B centered at $\mathbf{r}_B$. We define the separation D between these groups as

$$D = |\mathbf{r}_A - \mathbf{r}_B| - d_A - d_B \; . \tag{9.27}$$

When compressing the off-diagonal blocks resulting from the group pair A and B, *Serenity* will compute a new tolerance τ'_{ACA} based on the following heuristics:

$$\tau'_{ACA} = \tau_{ACA} \qquad D < 1\lambda \; , \tag{9.28}$$

$$\tau'_{ACA} = 2\,\tau_{ACA} \qquad 2\lambda > D \geq 1\lambda \; , \tag{9.29}$$

$$\tau'_{ACA} = 5\,\tau_{ACA} \qquad 5\lambda > D \geq 2\lambda \; , \tag{9.30}$$

$$\tau'_{ACA} = 10\,\tau_{ACA} \qquad 10\lambda > D \geq 5\lambda \; , \tag{9.31}$$

and

$$\tau'_{ACA} = 20\,\tau_{ACA} \qquad D \geq 10\lambda \; . \tag{9.32}$$

Based on numerical experimentation, these choices reduce memory overhead significantly without noticeably affecting the solution accuracy.

9.7.3 Spheres

In this section we will consider the bistatic RCS of conducting, fully dielectric, and coated spheres. These spheres have the same dimensions as the spheres in Section 8.8.3, but a much larger electrical size.

9.7.3.1 Conducting Sphere

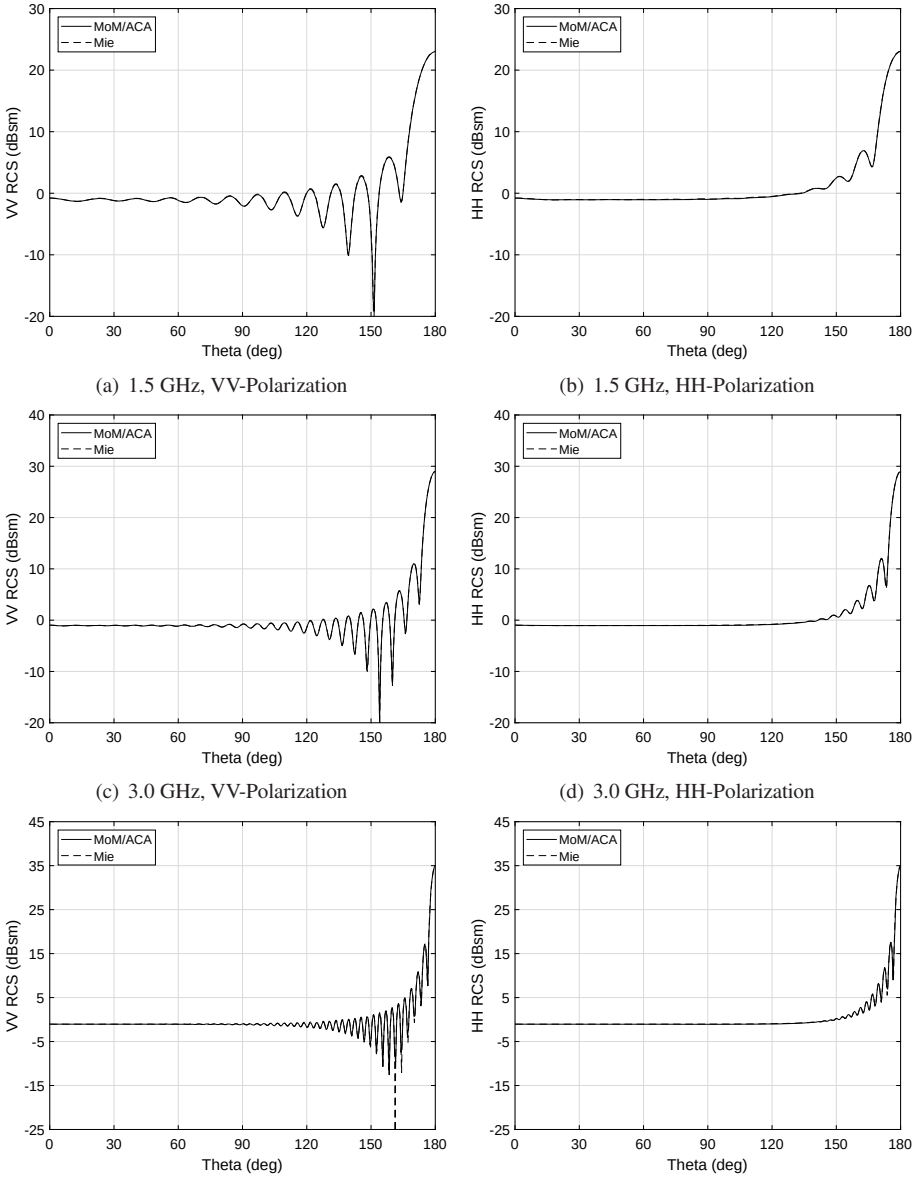
We first consider a conducting sphere with a radius of 0.5 meters at frequencies of 1.5, 3.0, and 6.0 GHz. For the MoM, we use models comprising 20480, 81920 and 327680 triangles, resulting in 30720, 122880 and 491520 unknowns, respectively. The bistatic RCS computed by the ACA is compared to the Mie series in Figures 9.4a–9.4f. The agreement is excellent.

9.7.3.2 Dielectric Sphere

We next consider a dielectric sphere with a radius of 0.5 meters at frequencies of 1.5, and 3.0 GHz. For the MoM simulation, we reuse the models from Section 9.7.3.1, however there are now an equal number of magnetic basis functions. This results in 61440 at 1.5 GHz and 245760 unknowns at 3.0 GHz. In Figures 9.5a–9.5d is compared the bistatic RCS computed by the ACA to the Mie series for a lossless sphere ($\epsilon_r = 2.56$) at 1.5 and 3.0 GHz. The agreement is excellent. A similar comparison is made for a lossy sphere ($\epsilon_r = 2.56 - j0.102$) at 3.0 GHz in Figures 9.6a–9.6b, where the agreement is again very good.

9.7.3.3 Coated Sphere

We next consider the coated sphere from Section 8.8.3.3 at 1.5 and 3.0 GHz. At 1.5 GHz, the surface meshes for the conducting core and the coating comprise 5120 and 20480 triangles, resulting in 38400 electric and 30720 magnetic basis functions, respectively, for a total of 69120 unknowns. At 3.0 GHz, they comprise 20480 and 81920 triangles, resulting in 153600 electric and 122880 magnetic basis functions, respectively, for a total of 276480 unknowns. In Figures 9.7a–9.7d is compared the bistatic RCS computed by the ACA to the Mie series for a lossless coating ($\epsilon_r = 2.56$) at 1.5 and 3.0 GHz. The agreement is excellent. A similar comparison is shown in Figures 9.8a–9.8b for a lossy coating ($\epsilon_r = 2.56 - j0.102$) at 3.0 GHz, where the agreement is again very good.



(a) 1.5 GHz, VV-Polarization

(b) 1.5 GHz, HH-Polarization

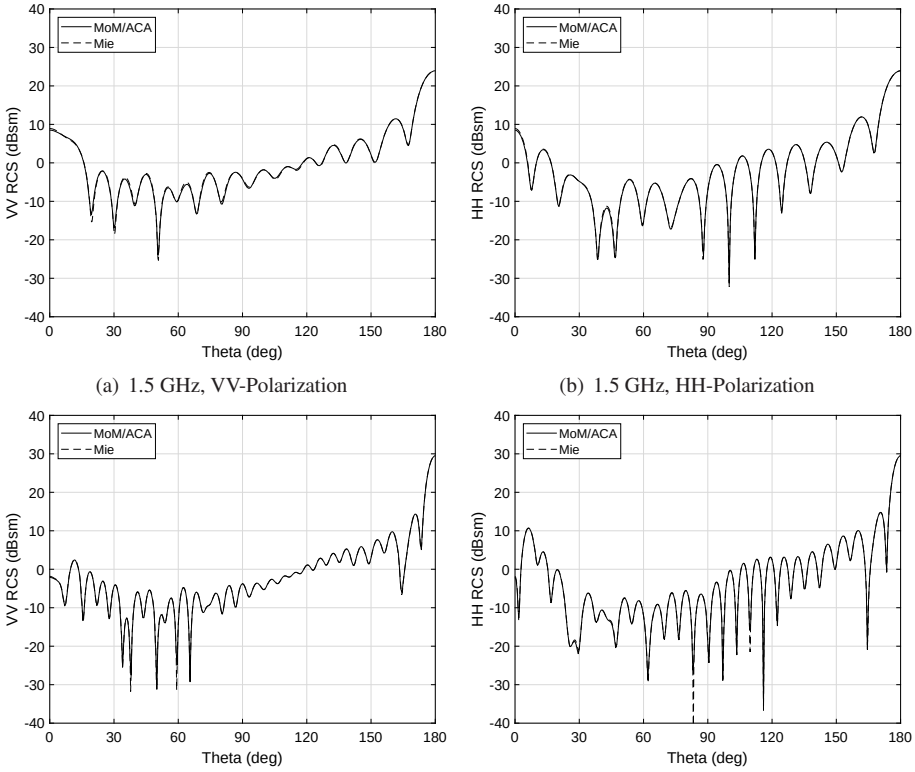
(c) 3.0 GHz, VV-Polarization

(d) 3.0 GHz, HH-Polarization

(e) 6.0 GHz, VV-Polarization

(f) 6.0 GHz, HH-Polarization

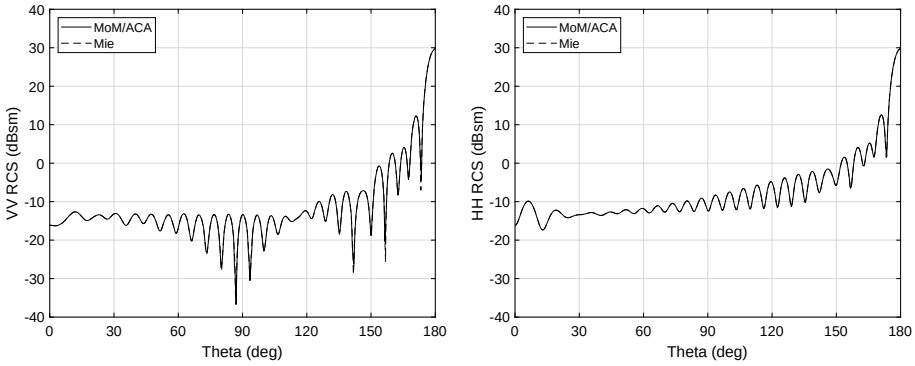
Figure 9.4: Conducting Sphere: Bistatic RCS



(a) 1.5 GHz, VV-Polarization (b) 1.5 GHz, HH-Polarization

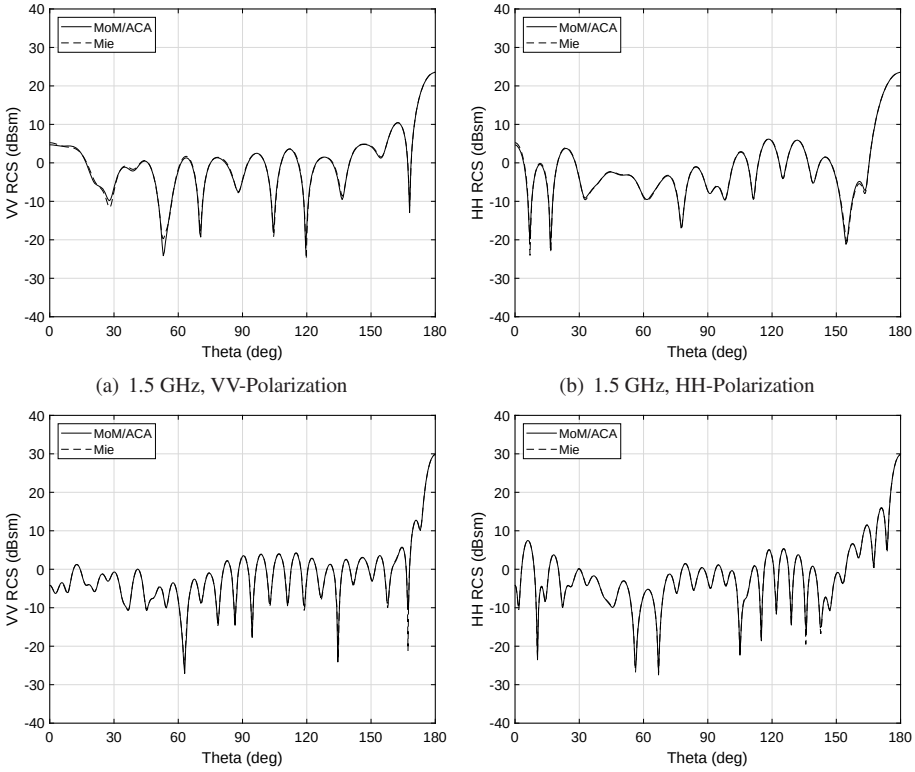
(c) 3.0 GHz, VV-Polarization (d) 3.0 GHz, HH-Polarization

Figure 9.5: Lossless Dielectric Sphere: Bistatic RCS



(a) 3.0 GHz, VV-Polarization (b) 3.0 GHz, HH-Polarization

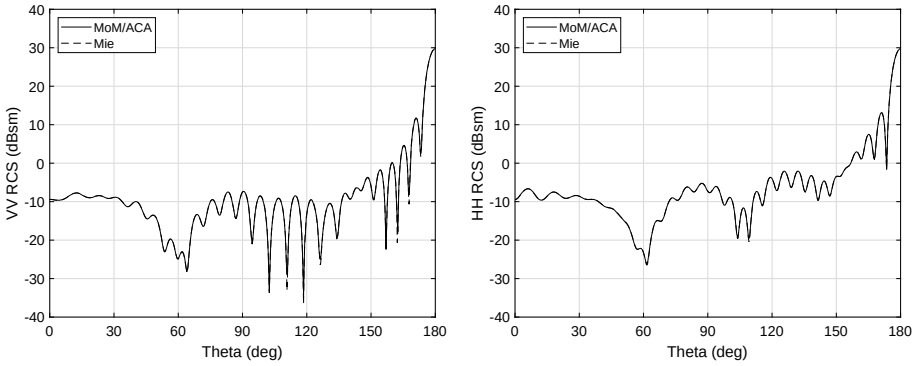
Figure 9.6: Lossy Dielectric Sphere: Bistatic RCS



(a) 1.5 GHz, VV-Polarization (b) 1.5 GHz, HH-Polarization

(c) 3.0 GHz, VV-Polarization (d) 3.0 GHz, HH-Polarization

Figure 9.7: Lossless Coated Sphere: Bistatic RCS



(a) 3.0 GHz, VV-Polarization (b) 3.0 GHz, HH-Polarization

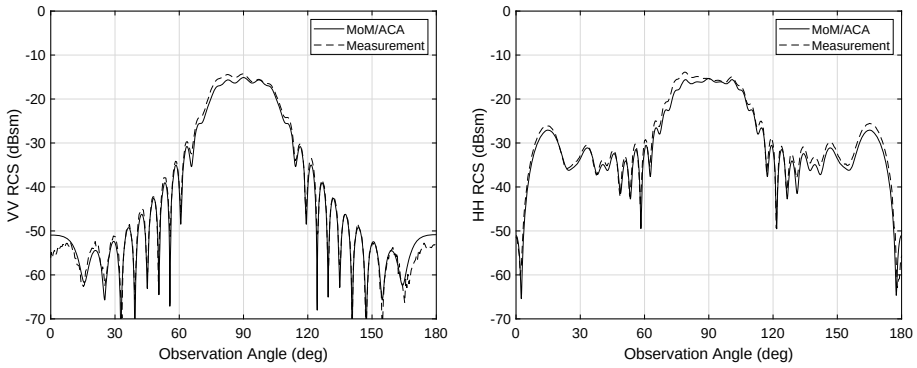
Figure 9.8: Lossy Coated Sphere: Bistatic RCS

9.7.4 EMCC Benchmark Targets

In this section, we again consider the EMCC benchmark radar targets previously discussed in Section 7.9.2, as well as the NASA almond from [15] and the cube and prism from [16].

9.7.4.1 EMCC Ogive

The EMCC ogive was previously described in Section 7.9.2.1, and the facet model used in this example comprises 21760 triangles, resulting in 32640 unknowns. The results from the ACA are compared to the EMCC measurements at 9.0 GHz in Figures 9.9a and 9.9b for vertical and horizontal polarizations, respectively. The agreement is very good. Surface current maps for $\theta^i = \pi/4$ are shown in Figures 9.16a and 9.16b for vertical and horizontal polarizations, respectively.



(a) VV-Polarization (b) HH-Polarization

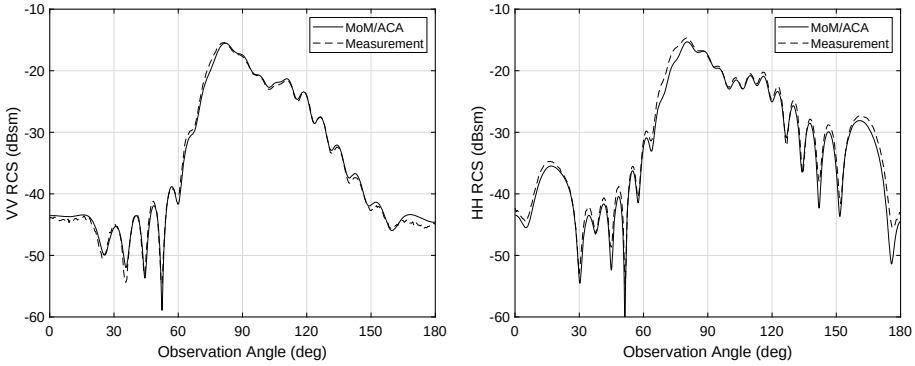
Figure 9.9: EMCC Ogive: MoM/ACA vs. Measurement at 9.0 GHz

9.7.4.2 EMCC Double Ogive

The EMCC double ogive was previously described in Section 7.9.2.2, and the facet model used in this example comprises 36240 triangles, resulting in 54360 unknowns. The results from the ACA are compared to the EMCC measurements at 9.0 GHz in Figures 9.10a and 9.10b for vertical and horizontal polarizations, respectively. The agreement is again very good. Surface current maps for $\theta^i = \pi/4$ are shown in Figures 9.16c and 9.16d for vertical and horizontal polarizations, respectively.

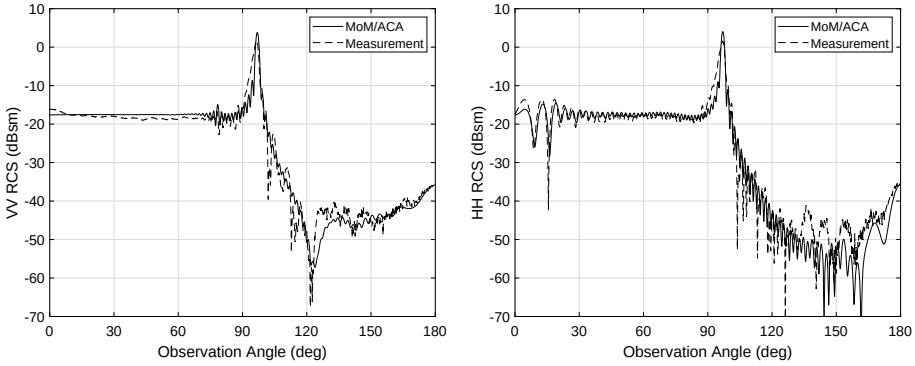
9.7.4.3 EMCC Cone-Sphere

The EMCC cone-sphere was previously described in Section 7.9.2.3, and the facet model used in this example comprises 79496 triangles, resulting in 119244 unknowns. The results from the ACA are compared to the EMCC measurements at 9.0 GHz in Figures 9.11a and 9.11b for vertical and horizontal polarizations, respectively. The



(a) VV-Polarization (b) HH-Polarization

Figure 9.10: EMCC Double Ogive: MoM/ACA vs. Measurement at 9.0 GHz



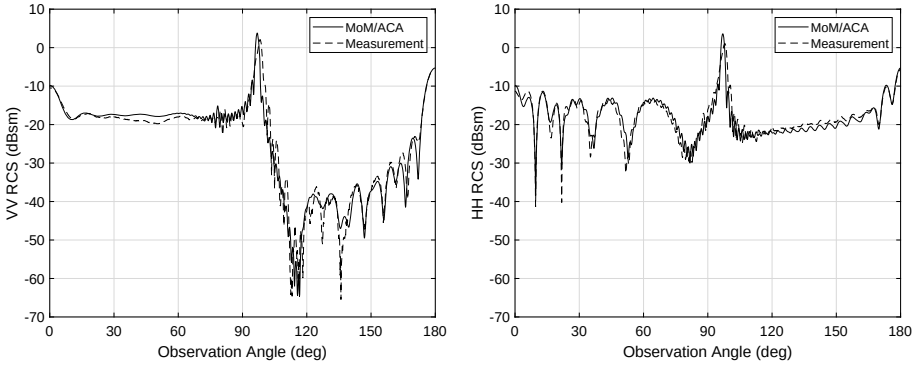
(a) VV-Polarization (b) HH-Polarization

Figure 9.11: EMCC Cone-Sphere: MoM/ACA vs. Measurement at 9.0 GHz

agreement is fairly good at all angles, although there is some disagreement at angles above 120 degrees. Surface current maps for $\theta^i = \pi/4$ are shown in Figures 9.16e and 9.16f for vertical and horizontal polarizations, respectively.

9.7.4.4 EMCC Cone-Sphere with Gap

The EMCC cone-sphere with gap was previously described in Section 7.9.2.4, and the facet model used in this example comprises 82528 triangles, resulting in 123792 unknowns. The results from the ACA are compared to the EMCC measurements at 9.0 GHz in Figures 9.12a and 9.12b for vertical and horizontal polarizations, respectively. The agreement is fairly good at all angles. Surface current maps for $\theta^i = \pi/4$ are shown in Figures 9.16g and 9.16h for vertical and horizontal polarizations, respectively.



(a) VV-Polarization (b) HH-Polarization

Figure 9.12: EMCC Cone-Sphere with Gap: MoM/ACA vs. Measurement at 9.0 GHz

9.7.4.5 NASA Almond

The 9.936-inch NASA almond of Figure 9.13 is a target comprising a mostly smooth surface that supports specular and creeping wave scattering, and a pointed end that gives rise to tip diffraction effects. It can be expressed for $0 \leq \phi \leq 2\pi$ as

$$x = dt \,, \tag{9.33}$$

$$y(t) = 0.193333d\sqrt{1 - \left(\frac{t}{0.416667}\right)^2}\cos\phi \,, \tag{9.34}$$

$$z(t) = 0.64444d\sqrt{1 - \left(\frac{t}{0.416667}\right)^2}\sin\phi \,, \tag{9.35}$$

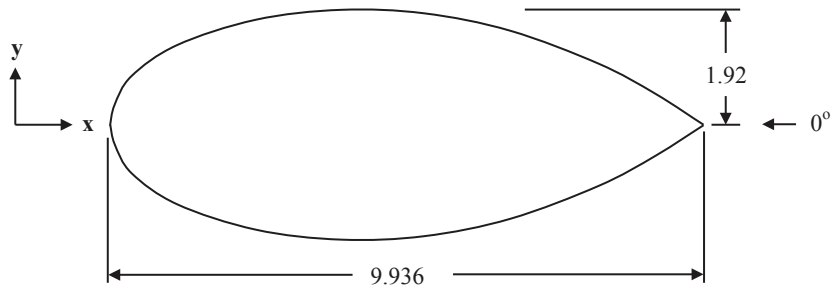
for $-0.41667 < t < 0$ inches, and

$$x = dt \,, \tag{9.36}$$

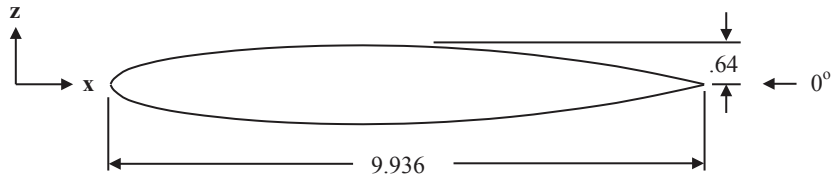
$$y(t) = 4.83345d\left[\sqrt{1 - \left(\frac{t}{2.08335}\right)^2} - 0.96\right]\cos\phi \,, \tag{9.37}$$

$$z(t) = 1.61115d\left[\sqrt{1 - \left(\frac{t}{2.08335}\right)^2} - 0.96\right]\sin\phi \,, \tag{9.38}$$

for $0 < t < 0.58333$, where $d = 9.936$ inches. The facet model for this example comprises 46984 triangles, resulting in 70476 unknowns. Details of the surface mesh for the pointed and rounded ends are shown in Figures 9.14a and 9.14b, respectively. Monostatic observations are made in the xy plane of Figure 9.13a. The results from the ACA are compared to the EMCC measurements at 7 GHz in Figures 9.15a and 9.15b for vertical and horizontal polarizations, respectively. The computed results were lower than the measurements by approximately 1.5 dB across a large range of angles, and

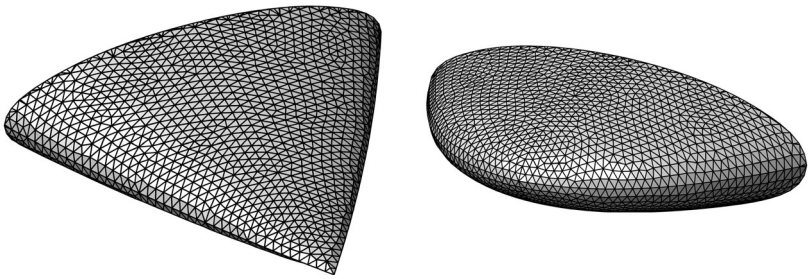


(a) Profile In xy Plane



(b) Profile In xz Plane

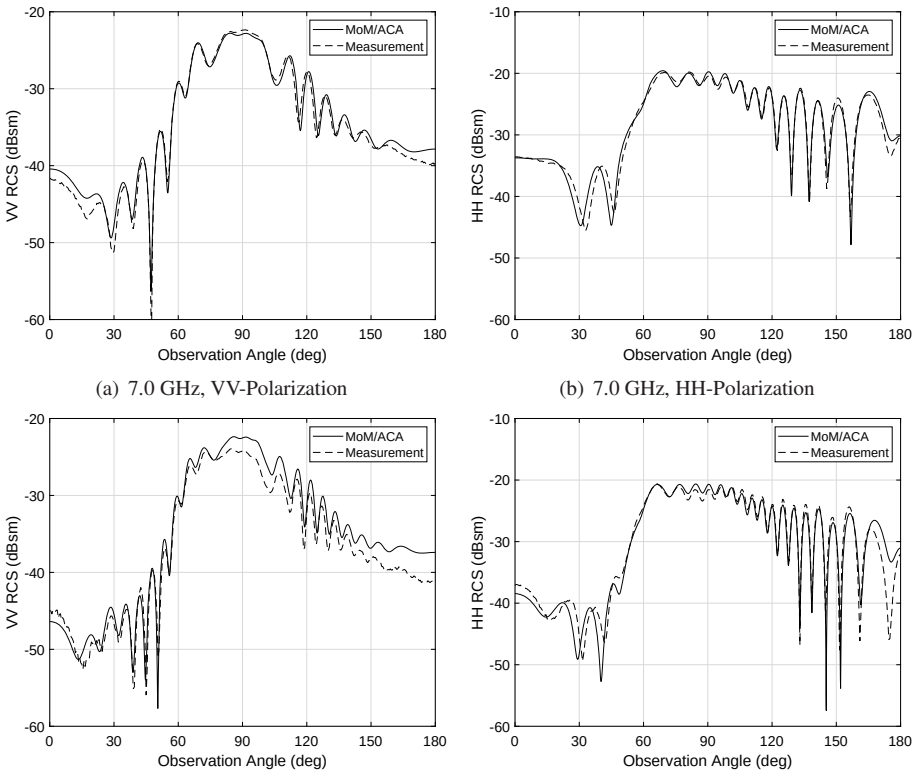
Figure 9.13: NASA Almond



(a) Pointed End (b) Rounded End

Figure 9.14: Almond Facet Model Detail

were adjusted by that amount in each figure. After this adjustment, the comparison is very good. Results at 9.92 GHz are shown in Figures 9.15c and 9.15d for vertical and horizontal polarizations, respectively. A similar bias was also observed in this case, and after adjustment the comparison is again fairly good. However, the difference is much greater in this case, particularly in VV polarization. The reason for the difference is not known, though a similar difference was also noted in [15]. The authors of that paper suggested the cause may have been the density of the unknowns ($6/\lambda$) in the CAD model that was used. As we use more than double this density in our model (about $13/\lambda$), we suspect the cause to be something else.



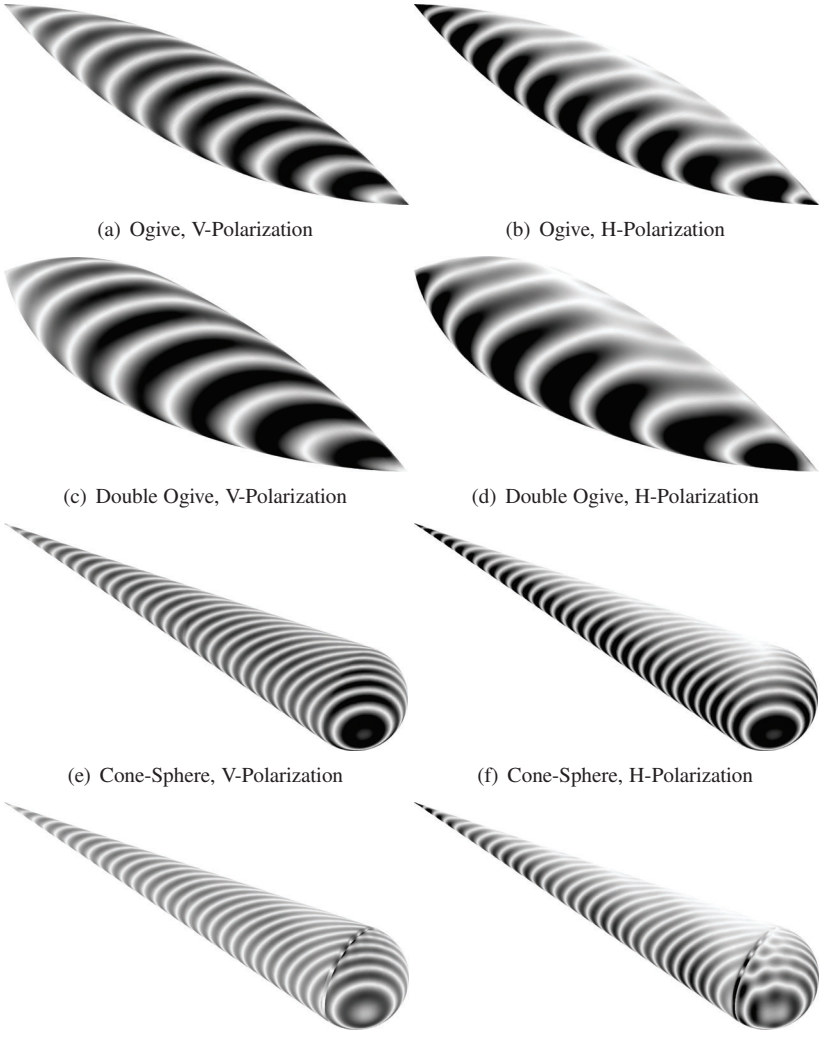
(a) 7.0 GHz, VV-Polarization

(b) 7.0 GHz, HH-Polarization

(c) 9.92 GHz, VV-Polarization

(d) 9.92 GHz, HH-Polarization

Figure 9.15: NASA Almond: MoM/ACA vs. Measurement



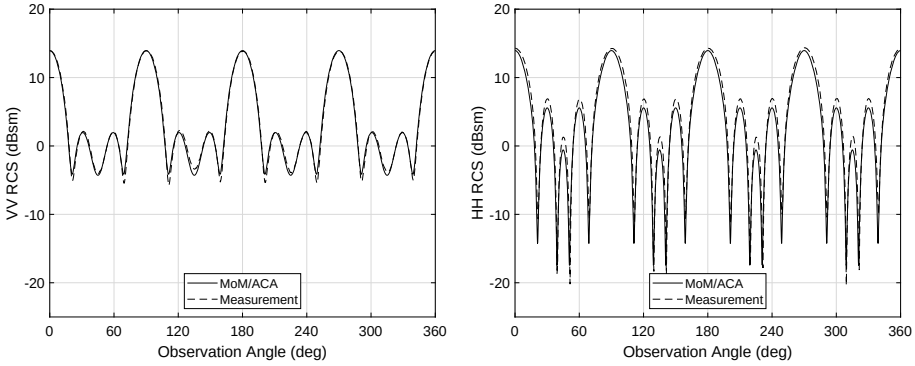
(a) Ogive, V-Polarization (b) Ogive, H-Polarization

(c) Double Ogive, V-Polarization (d) Double Ogive, H-Polarization

(e) Cone-Sphere, V-Polarization (f) Cone-Sphere, H-Polarization

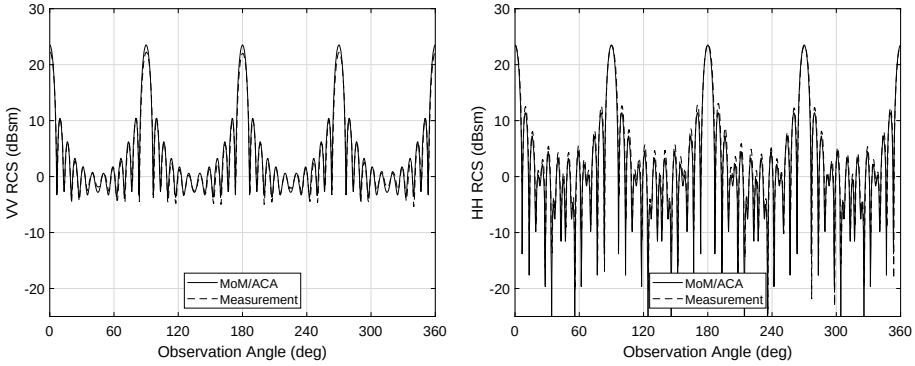
(g) Cone-Sphere with Gap, V-Polarization (h) Cone-Sphere with Gap, H-Polarization

Figure 9.16: Surface Currents on EMCC Test Targets at 9.0 GHz



(a) VV-Polarization (b) HH-Polarization

Figure 9.17: EMCC Cube: MoM/ACA vs. Measurement at 0.43 GHz



(a) VV-Polarization (b) HH-Polarization

Figure 9.18: EMCC Cube: MoM/ACA vs. Measurement at 1.3 GHz

9.7.4.6 EMCC Cube

The EMCC cube is a high-precision, conducting cube with sides of length 1 meter and axis-aligned faces [16]. Monostatic observations are made in the xy plane at 0.43 and 1.3 GHz, for azimuth angles from 0 to 360 degrees. For the cube, facet models with 4542 and 41818 triangles were constructed, yielding a total of 6813 and 62727 unknowns, respectively. The results from the ACA are compared to the EMCC measurements at 0.43 GHz in Figures 9.17a and 9.17b, and at 1.3 GHz in Figures 9.18a and 9.18b, respectively. The comparison is very good at all angles and frequencies.

9.7.4.7 EMCC Prism

The EMCC prism [16] has a base with dimensions shown in Figure 9.19, and an overall thickness of 30.48 cm. Monostatic observations are made at an elevation of 10 degrees, at 0.43 and 1.3 GHz, for azimuth angles from 0 to 360 degrees. The purpose of this

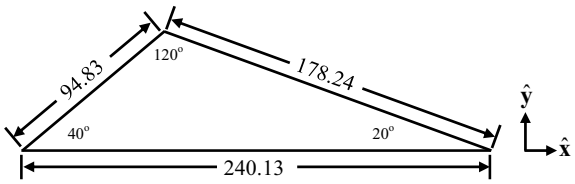
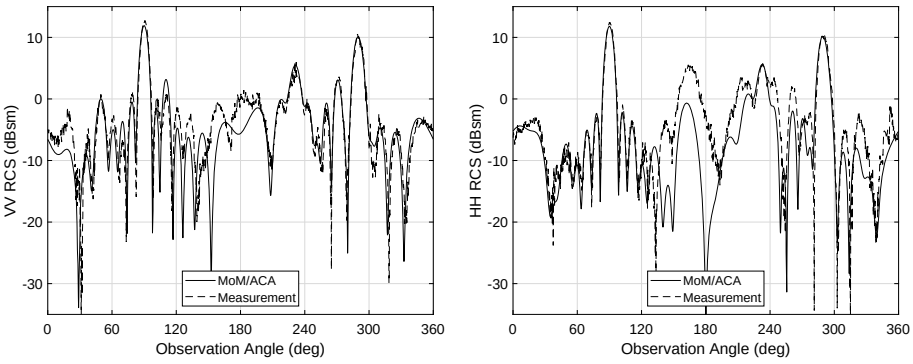
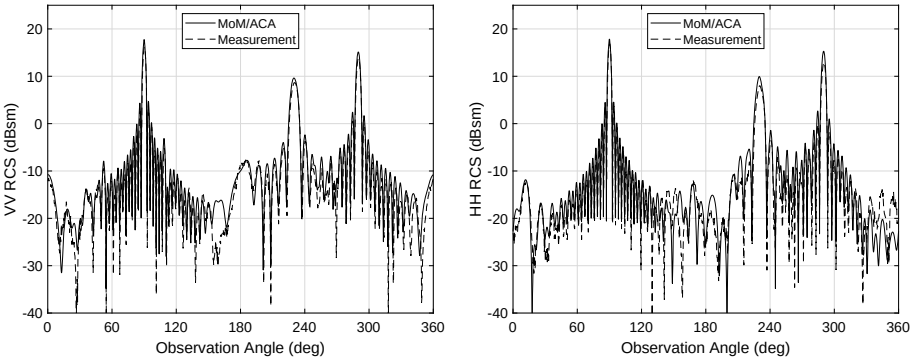


Figure 9.19: EMCC Prism (Dimensions in Centimeters)



(a) VV-Polarization (b) HH-Polarization

Figure 9.20: EMCC Prism: MoM/ACA vs. Measurement at 0.43 GHz



(a) VV-Polarization (b) HH-Polarization

Figure 9.21: EMCC Prism: MoM/ACA vs. Measurement at 1.3 GHz

object and measurement setup is to capture traveling wave effects for three different wedge angles [16].

For the prism, facet models with 3610 and 31830 triangles were constructed, yielding a total of 5415 and 47745 unknowns, respectively. The results from the ACA are compared to the EMCC measurements at 0.43 GHz in Figures 9.20a and 9.20b, and at 1.3 GHz in Figures 9.21a and 9.21b, respectively. The comparison is fairly good at and

near specular angles, but there is a fair amount of disagreement elsewhere, particularly near 180 degrees azimuth. Comparisons in [16] against results obtained from the Fast Illinois Solver Code (FISC) [17] showed similar disagreements.

9.7.5 Dielectric Cube and Ogive

Next, we will consider a small cube and ogive, two fully dielectric benchmark targets that were fabricated and measured in [18]. These targets were milled from a solid block of polyethylene, which was measured to have an average permittivity $\epsilon_r \approx 2.35$. As the measured data was taken from a scanned version of the document, only the HH-polarized RCS was visible enough to allow extraction.

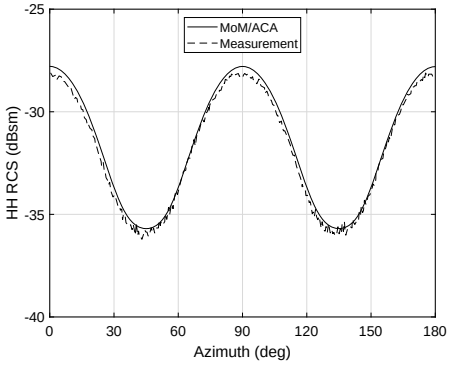
9.7.5.1 Small Polyethylene Cube

The small polyethylene cube has a side length of approximately 1.181 inches, with an error of ± 0.002 inches [18]. We consider the RCS at 6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 16.0 and 18.0 GHz. For the cube, a facet model with 10800 facets was constructed, comprising 10 edges per wavelength at 18.0 GHz, and a total of 32400 unknowns. This model was used at all frequencies.

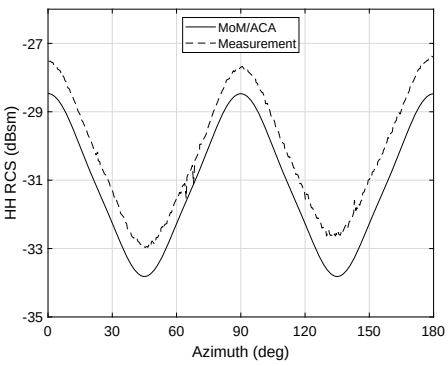
The computed and measured results are compared for horizontal polarization in Figures 9.22a–9.22h. The comparison is fairly good at all frequencies, though the simulated results consistently underpredict the RCS. This was also observed by the author in [18] using results from a full-matrix MoM solver.

9.7.5.2 Polyethylene Ogive

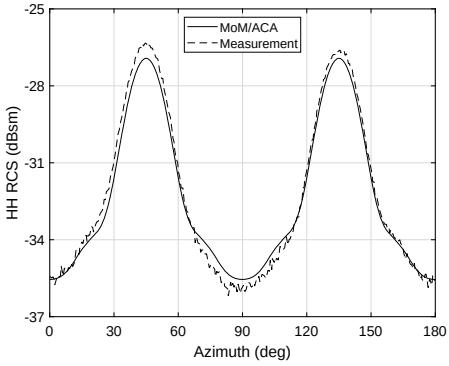
The polyethylene ogive has the same exterior dimensions as the EMCC ogive described previously in Section 7.9.2.1, with an error of ± 0.005 inches. We consider the RCS at 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0 and 18.0 GHz. For convenience, a single facet model was constructed having 10 edges per wavelength at 18.0 GHz, and was used at all frequencies. This model had 77624 facets, resulting in 116436 edges and 232872 unknowns. The computed and measured results are compared for horizontal polarization in Figures 9.23a–9.23h. The comparison is quite good at all frequencies, and across all incident angles except for those near tip-on incidence where the RCS is much lower than at broadside. The memory requirements versus frequency are summarized in Table 9.1. Interestingly, the amount of compression is observed to decrease versus frequency.



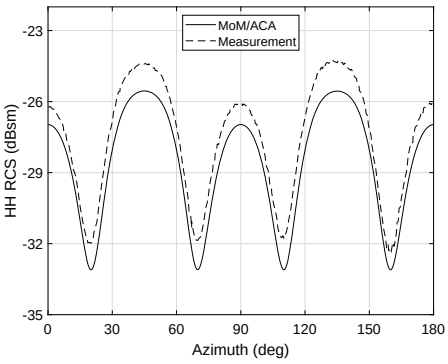
(a) 6.0 GHz, HH-Polarization



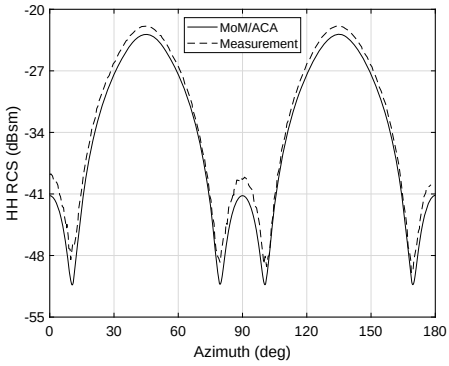
(b) 8.0 GHz, HH-Polarization



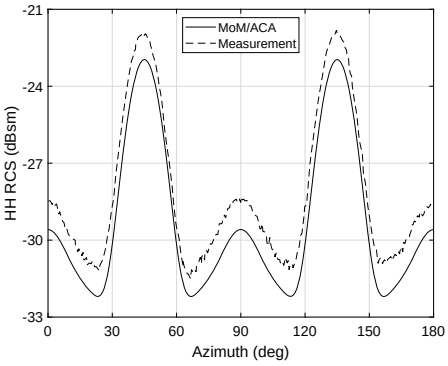
(c) 10.0 GHz, HH-Polarization



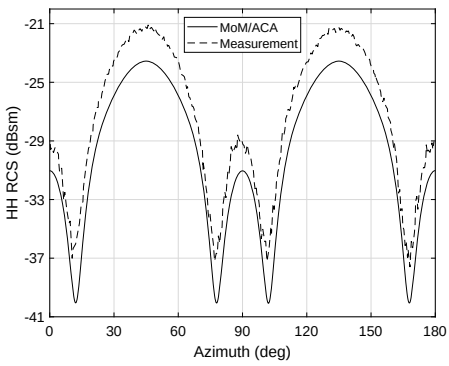
(d) 12.0 GHz, HH-Polarization



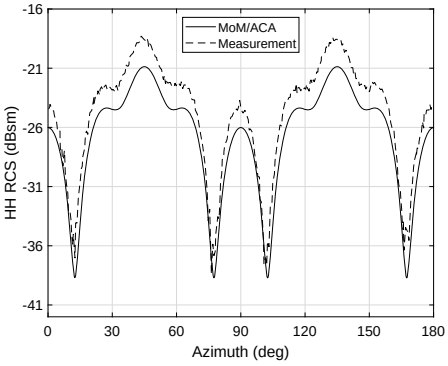
(e) 13.0 GHz, HH-Polarization



(f) 14.0 GHz, HH-Polarization

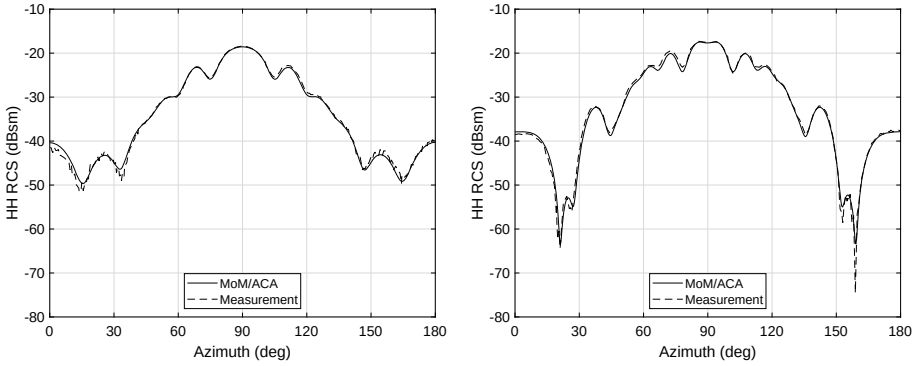


(g) 16.0 GHz, HH-Polarization



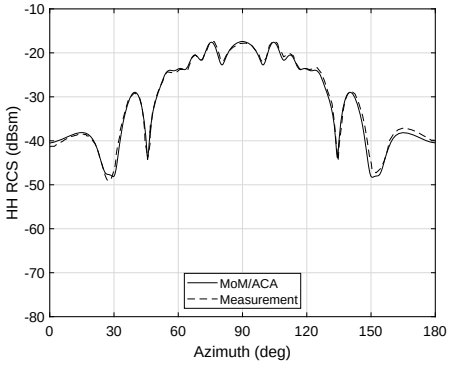
(h) 18.0 GHz, HH-Polarization

Figure 9.22: Small Polyethylene Cube: RCS

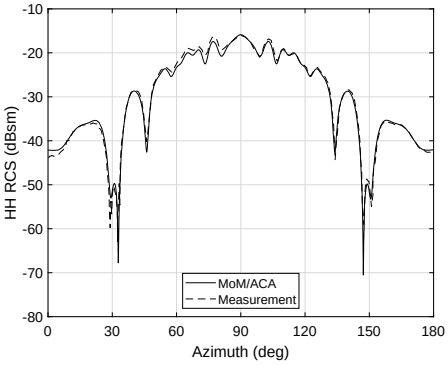


(a) 4.0 GHz, HH-Polarization

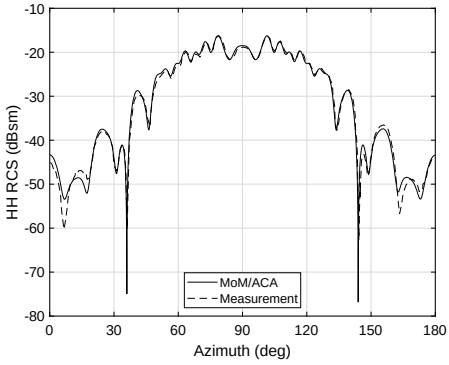
(b) 6.0 GHz, HH-Polarization



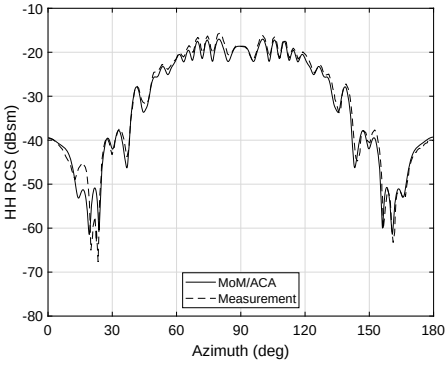
(c) 8.0 GHz, HH-Polarization



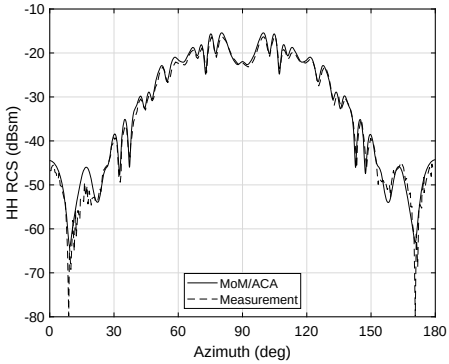
(d) 10.0 GHz, HH-Polarization



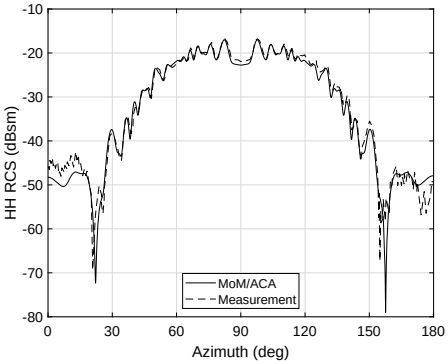
(e) 12.0 GHz, HH-Polarization



(f) 14.0 GHz, HH-Polarization



(g) 16.0 GHz, HH-Polarization



(h) 18.0 GHz, HH-Polarization

Figure 9.23: Polyethylene Ogive: RCS

9.7.6 UT Austin Benchmark Targets

We will next consider several conducting, dielectric and composite targets from the Austin Benchmark Suites for Computational Electromagnetics, a publicly available benchmark suite for RCS simulations [19]. The targets fabricated for this suite were 3D-printed in resin, measured to have an average permittivity of $\epsilon_r = 3.0 - j0.1$. The printed parts were then smoothed by hand. Bulk dielectric parts comprise the bare resin, and a thin layer of highly conductive silver paint was applied to create the conducting surfaces [20, 21].

9.7.6.1 PEC Almond

We first consider the PEC almond, which was fabricated to have the same external dimensions as in Figure 9.13. We consider the monostatic RCS at 3.5, 5.125, 7.0 and 10.25 GHz, and reuse the facet model from Section 9.7.4.5. The computed and measured results are compared for vertical and horizontal polarizations in Figures 9.25a–9.25h, where the comparison is fairly good.

9.7.6.2 Solid Resin Almond

We next consider a solid resin almond of the same dimensions, again using the same facet model. We consider the monostatic RCS at 2.58, 5.125, 7.0 and 10.25 GHz. The computed and measured results are compared for vertical and horizontal polarizations in Figures 9.26a–9.26h, where the comparison is again fairly good.

9.7.6.3 Closed-Tail Almond

We next consider the closed-tail almond of the same exterior dimensions. In this case, a solid resin almond was fabricated and then split as shown in Figure 9.24. The 4.14-inch ellipsoidal portion was coated with conducting paint, and the two halves were joined with a thin layer of epoxy. For the MoM simulation, new facet models were constructed, having 31892, 28246, and 3876 facets, comprising the exterior of the solid resin tip, conducting rear, and the septum between the halves, respectively. The computed and measured results are compared for vertical and horizontal polarizations in Figures 9.27a–9.27h, where the comparison is again fairly good.

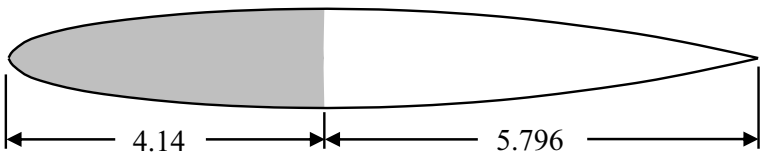
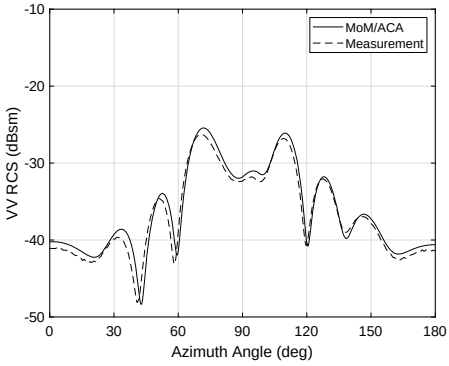
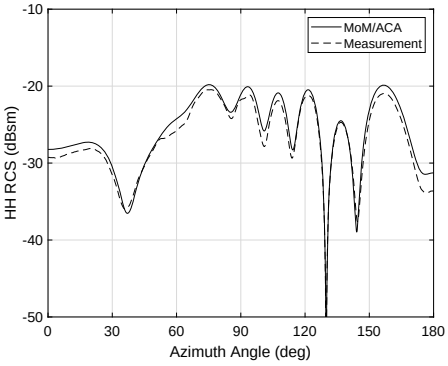


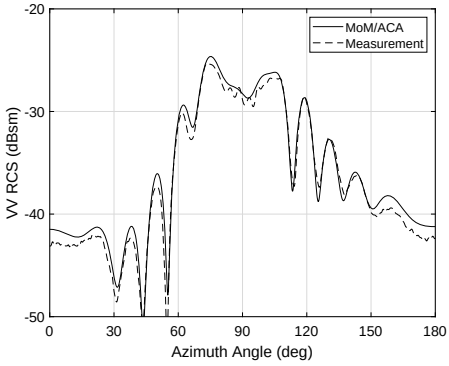
Figure 9.24: Closed-Tail Almond



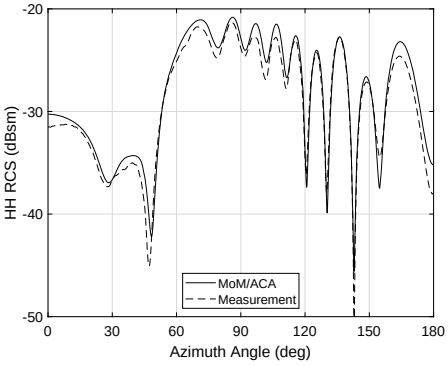
(a) 3.5 GHz, VV-Polarization



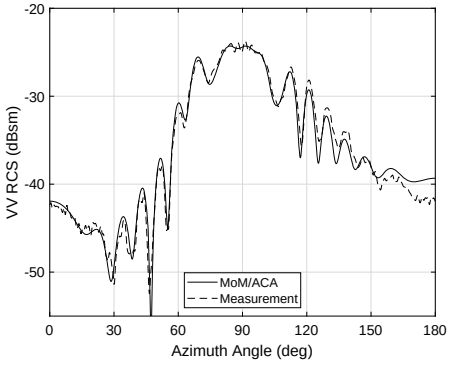
(b) 3.5 GHz, HH-Polarization



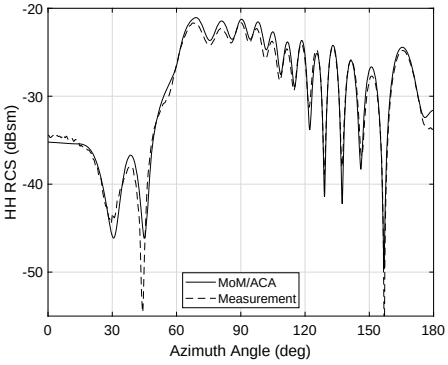
(c) 5.125 GHz, VV-Polarization



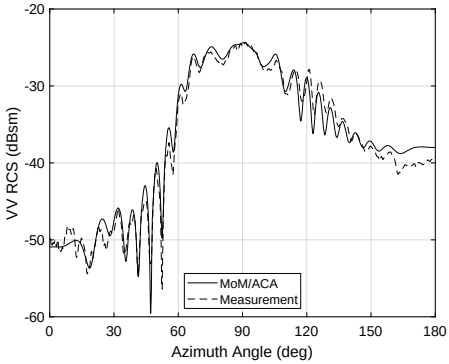
(d) 5.125 GHz, HH-Polarization



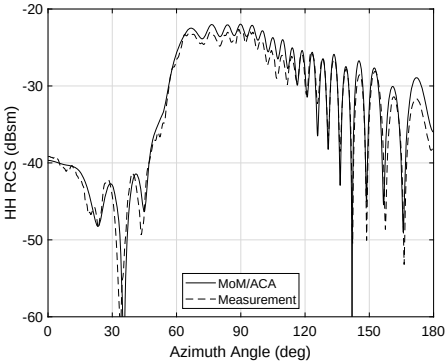
(e) 7.0 GHz, VV-Polarization



(f) 7.0 GHz, HH-Polarization

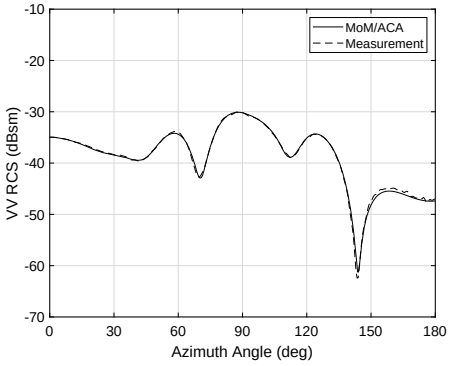


(g) 10.25 GHz, VV-Polarization

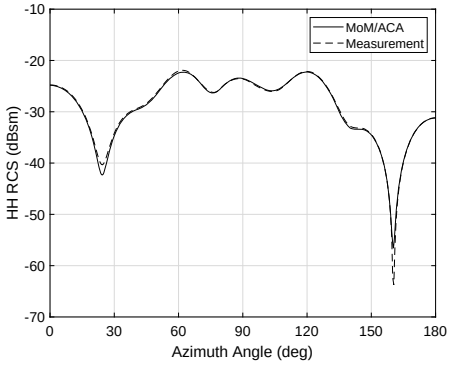


(h) 10.25 GHz, HH-Polarization

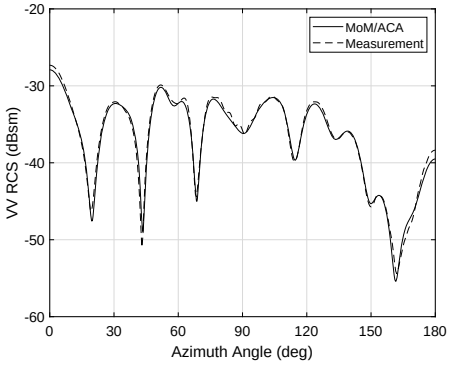
Figure 9.25: Austin PEC Almond: RCS



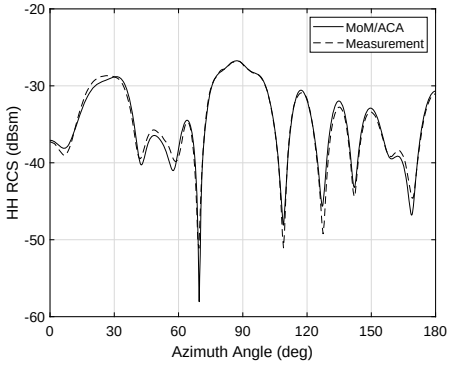
(a) 2.58 GHz, VV-Polarization



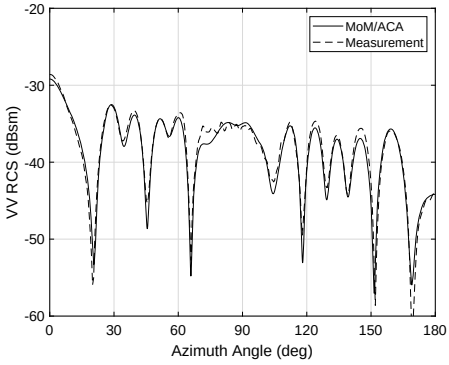
(b) 2.58 GHz, HH-Polarization



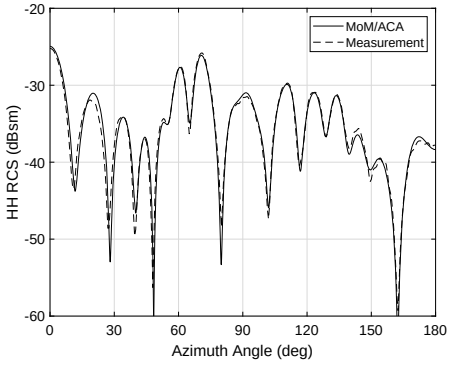
(c) 5.125 GHz, VV-Polarization



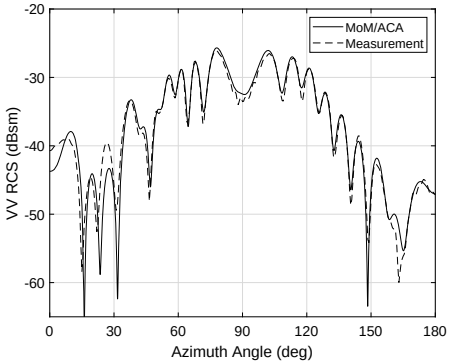
(d) 5.125 GHz, HH-Polarization



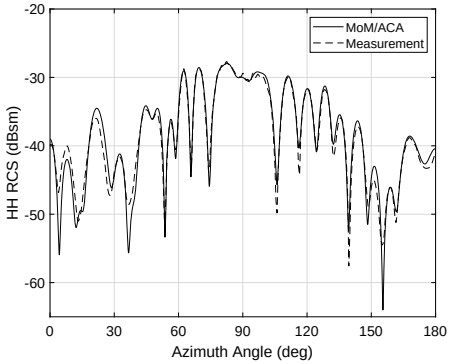
(e) 7.0 GHz, VV-Polarization



(f) 7.0 GHz, HH-Polarization

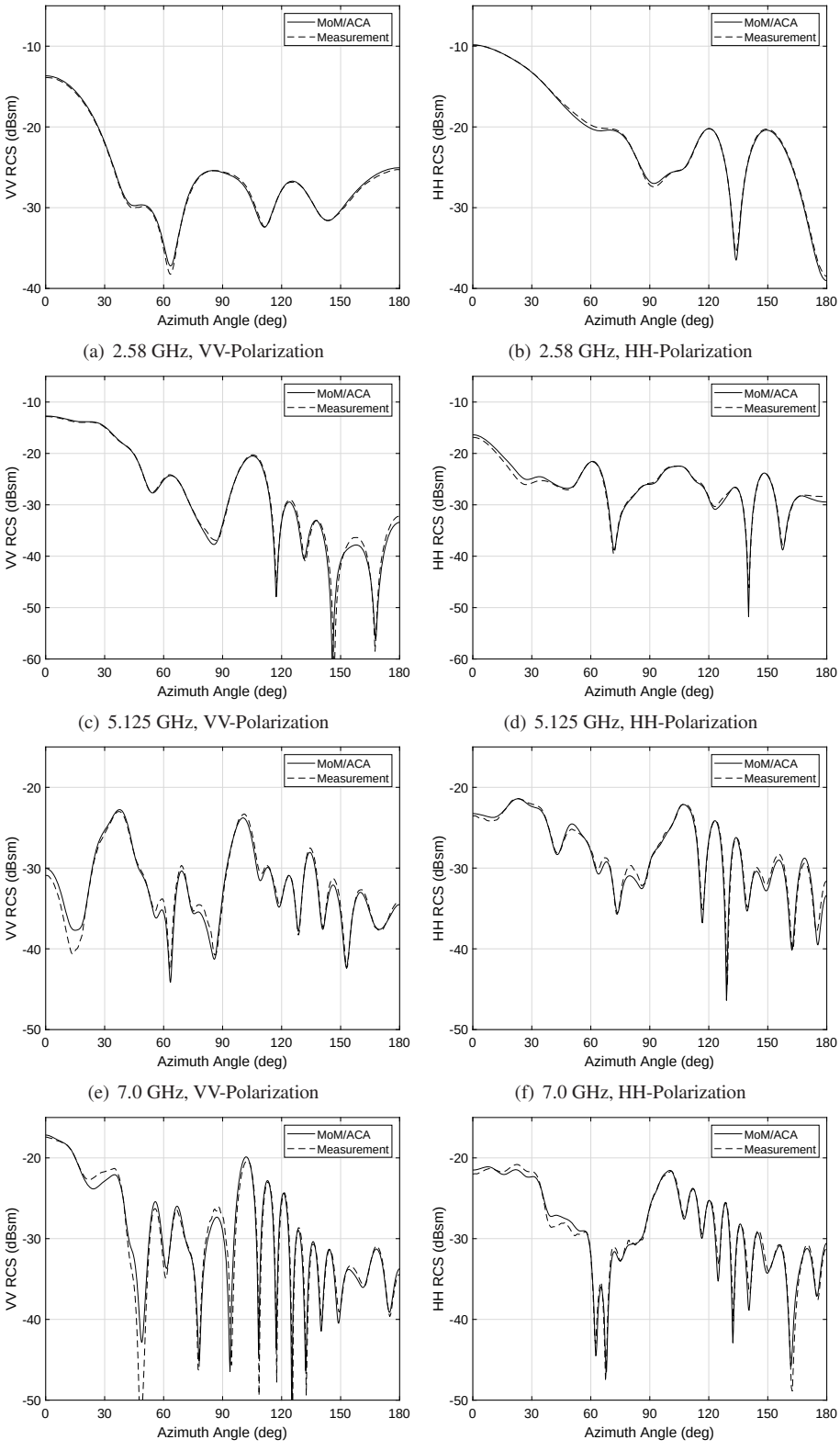


(g) 10.25 GHz, VV-Polarization



(h) 10.25 GHz, HH-Polarization

Figure 9.26: Austin Solid Resin Almond: RCS



(a) 2.58 GHz, VV-Polarization

(b) 2.58 GHz, HH-Polarization

(c) 5.125 GHz, VV-Polarization

(d) 5.125 GHz, HH-Polarization

(e) 7.0 GHz, VV-Polarization

(f) 7.0 GHz, HH-Polarization

(g) 10.25 GHz, VV-Polarization

(h) 10.25 GHz, HH-Polarization

Figure 9.27: Austin Closed-Tail Almond: RCS

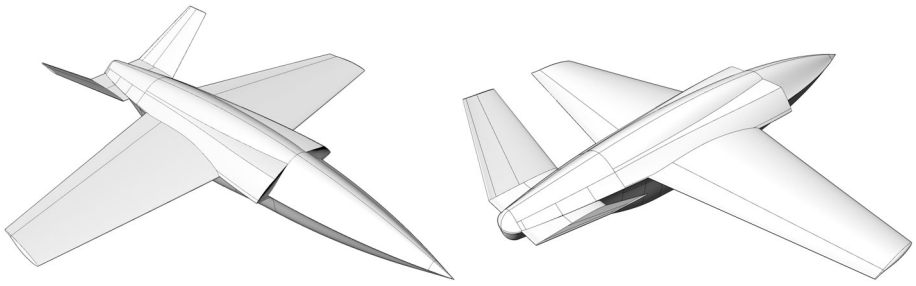
9.7.6.4 Open-Tail Almond

We next consider the open-tail almond, similar to the closed-tail almond in Figure 9.24. In this case, a solid resin almond was fabricated, the 5.796-inch tip portion covered in masking tape, and the 4.14-inch ellipsoidal portion sprayed with conducting paint [21]. This yields an open rear cavity with a thin, conducting wall. For the MoM simulation, the facet models comprising the exterior surfaces of the closed-tail almond were reused. On the tip portion, comprising the air-dielectric interface, PMCHWT is applied as usual. The rear portion, however, comprises a thin conducting interface that borders two conducting regions. Thus, separate electric basis functions are assigned to each side of the interface and only the EFIE is applied. The computed and measured results are compared for vertical and horizontal polarizations in Figures 9.29a–9.29h, where the comparison is again fairly good.

9.7.6.5 EXPEDITE-RCS Aircraft

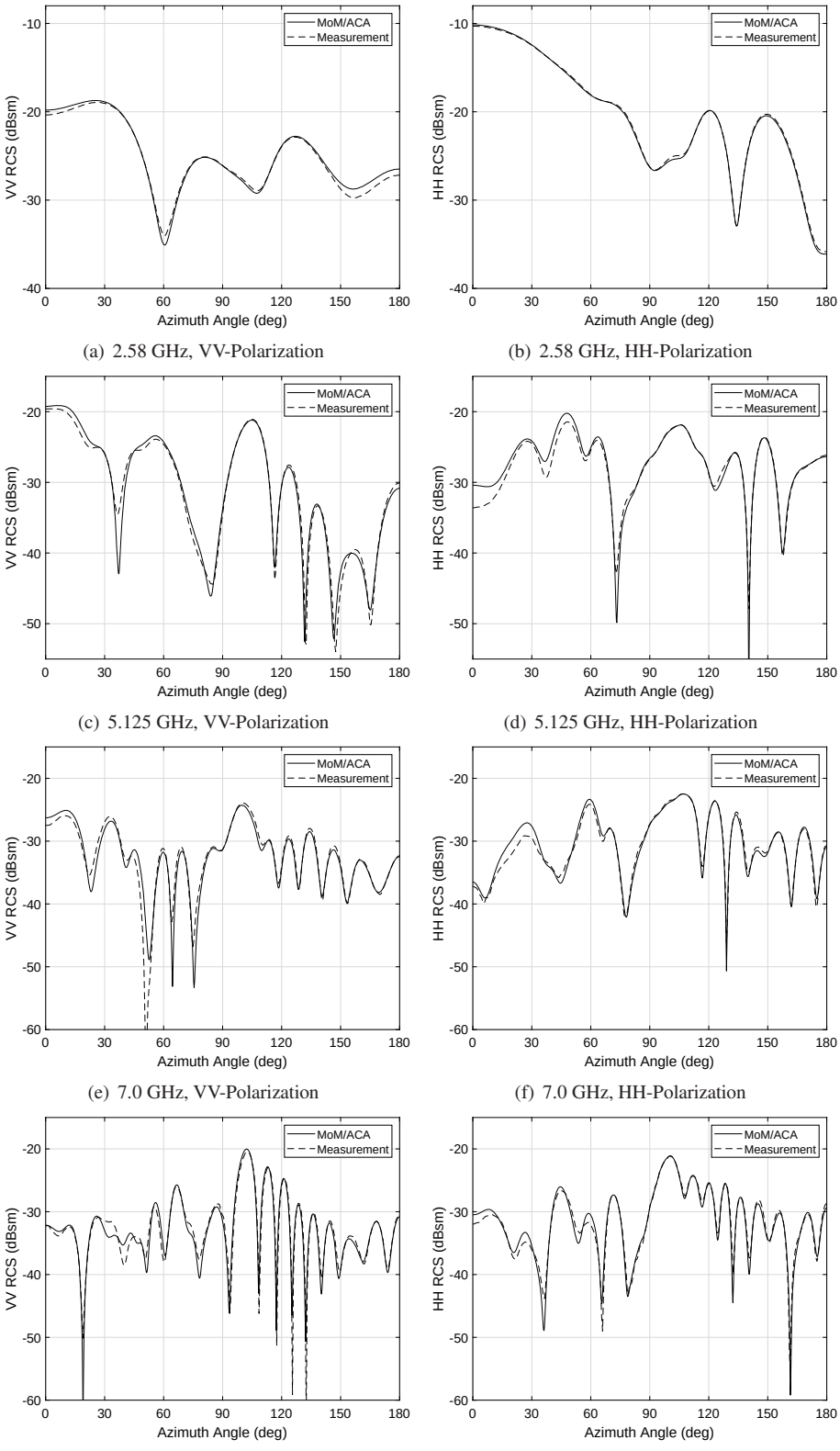
We next consider the EXPEDITE-RCS aircraft model described in [22], and depicted in Figures 9.28a and 9.28b from front and rear perspectives, respectively. This geometry was developed to create a complex, realistic aircraft model that could be used and disseminated in a collaborative manner. The 9.1875-inch long test article was fabricated and measured in the same manner as the almond in [21]. This particular model is watertight, with engine intake and exhaust cavities completely closed off.

A number of meshes for the aircraft are available in the UT Austin Benchmark suite. From these, a model having 155432 triangles and 233148 edges was chosen for the MoM simulation. This results in 233148 and 466296 unknowns for the PEC and dielectric cases, respectively. The computed and measured results for the PEC model are compared at 2.58, 5.125, 7.0 and 10.25 GHz in Figures 9.30a–9.30h, in vertical and horizontal polarizations, where the comparison is fairly good. A similar comparison is made for the solid resin model in Figures 9.31a–9.31h, where the comparison is again fairly good.



(a) Front View (b) Rear View

Figure 9.28: Austin EXPEDITE-RCS Aircraft Model



(a) 2.58 GHz, VV-Polarization

(b) 2.58 GHz, HH-Polarization

(c) 5.125 GHz, VV-Polarization

(d) 5.125 GHz, HH-Polarization

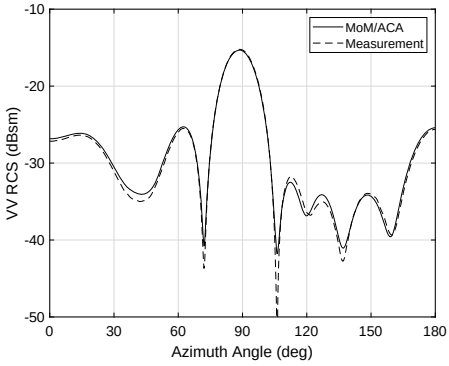
(e) 7.0 GHz, VV-Polarization

(f) 7.0 GHz, HH-Polarization

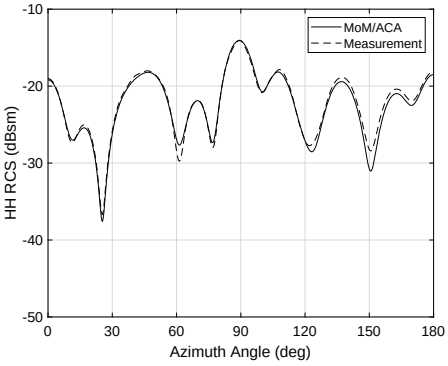
(g) 10.25 GHz, VV-Polarization

(h) 10.25 GHz, HH-Polarization

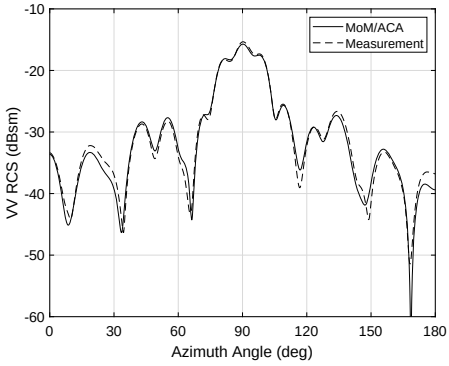
Figure 9.29: Austin Open-Tail Almond: RCS



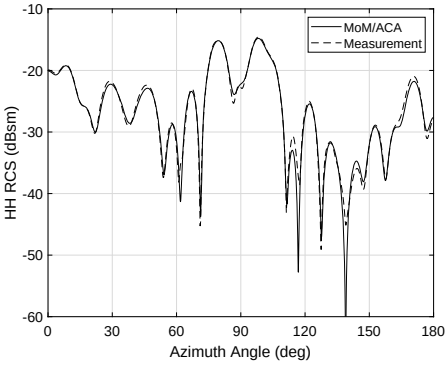
(a) 2.58 GHz, VV-Polarization



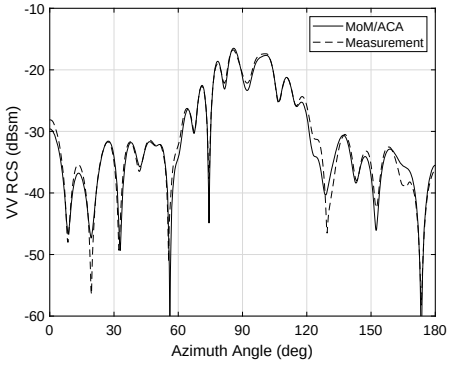
(b) 2.58 GHz, HH-Polarization



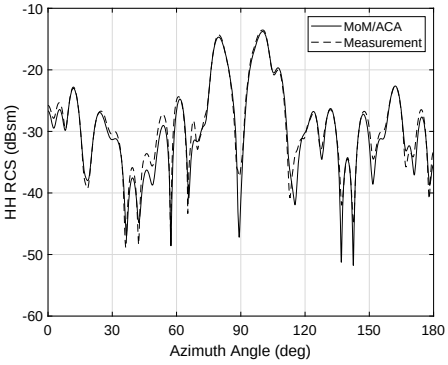
(c) 5.125 GHz, VV-Polarization



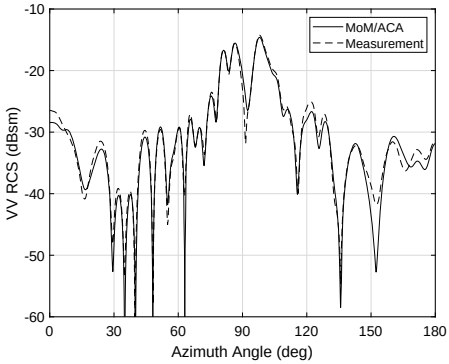
(d) 5.125 GHz, HH-Polarization



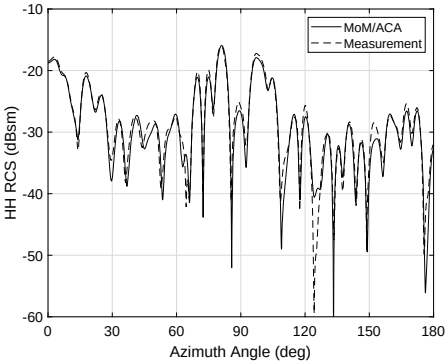
(e) 7.0 GHz, VV-Polarization



(f) 7.0 GHz, HH-Polarization

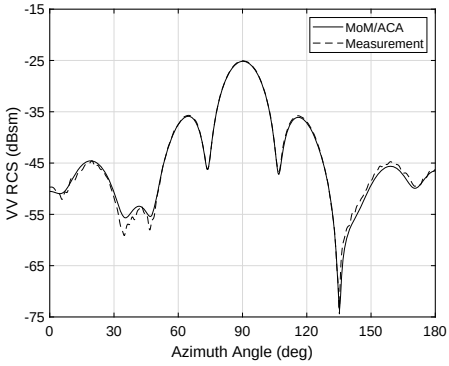


(g) 10.25 GHz, VV-Polarization

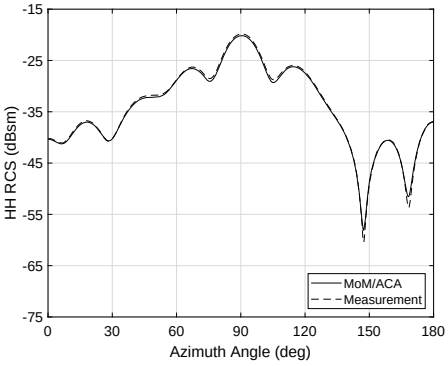


(h) 10.25 GHz, HH-Polarization

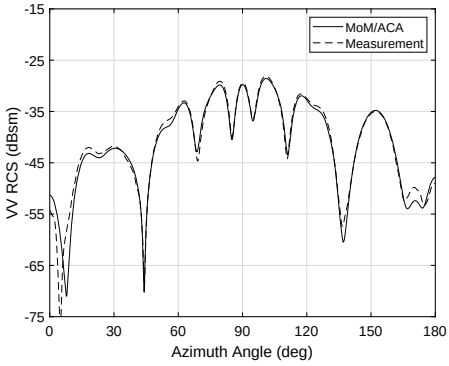
Figure 9.30: PEC EXPEDITE-RCS Aircraft: RCS



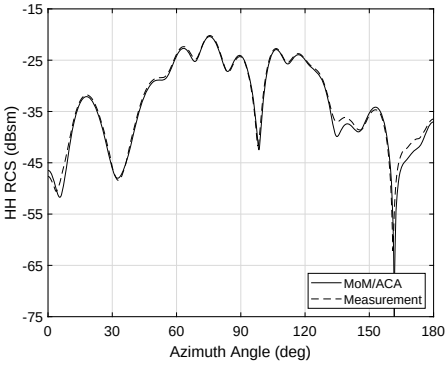
(a) 2.58 GHz, VV-Polarization



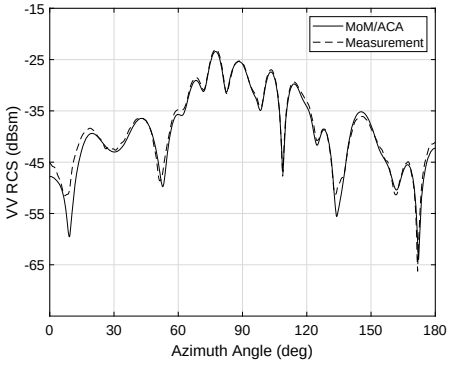
(b) 2.58 GHz, HH-Polarization



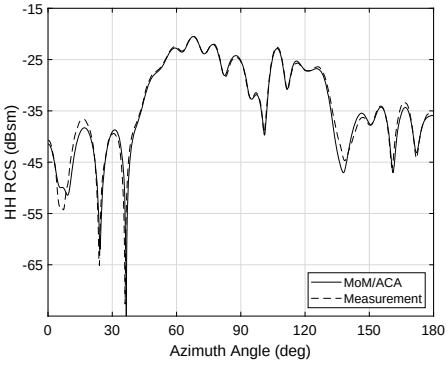
(c) 5.125 GHz, VV-Polarization



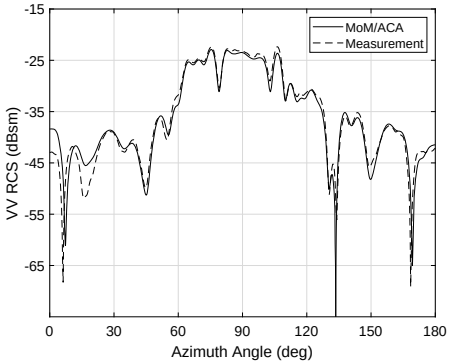
(d) 5.125 GHz, HH-Polarization



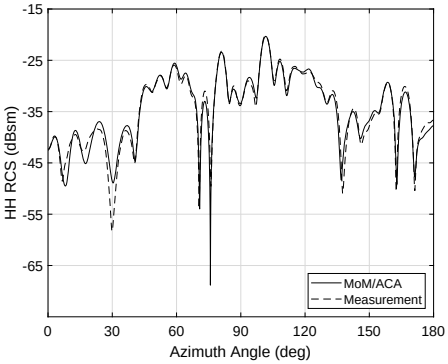
(e) 7.0 GHz, VV-Polarization



(f) 7.0 GHz, HH-Polarization



(g) 10.25 GHz, VV-Polarization



(h) 10.25 GHz, HH-Polarization

Figure 9.31: Solid Resin EXPEDITE-RCS Aircraft: RCS

9.7.7 Monoconic Reentry Vehicle

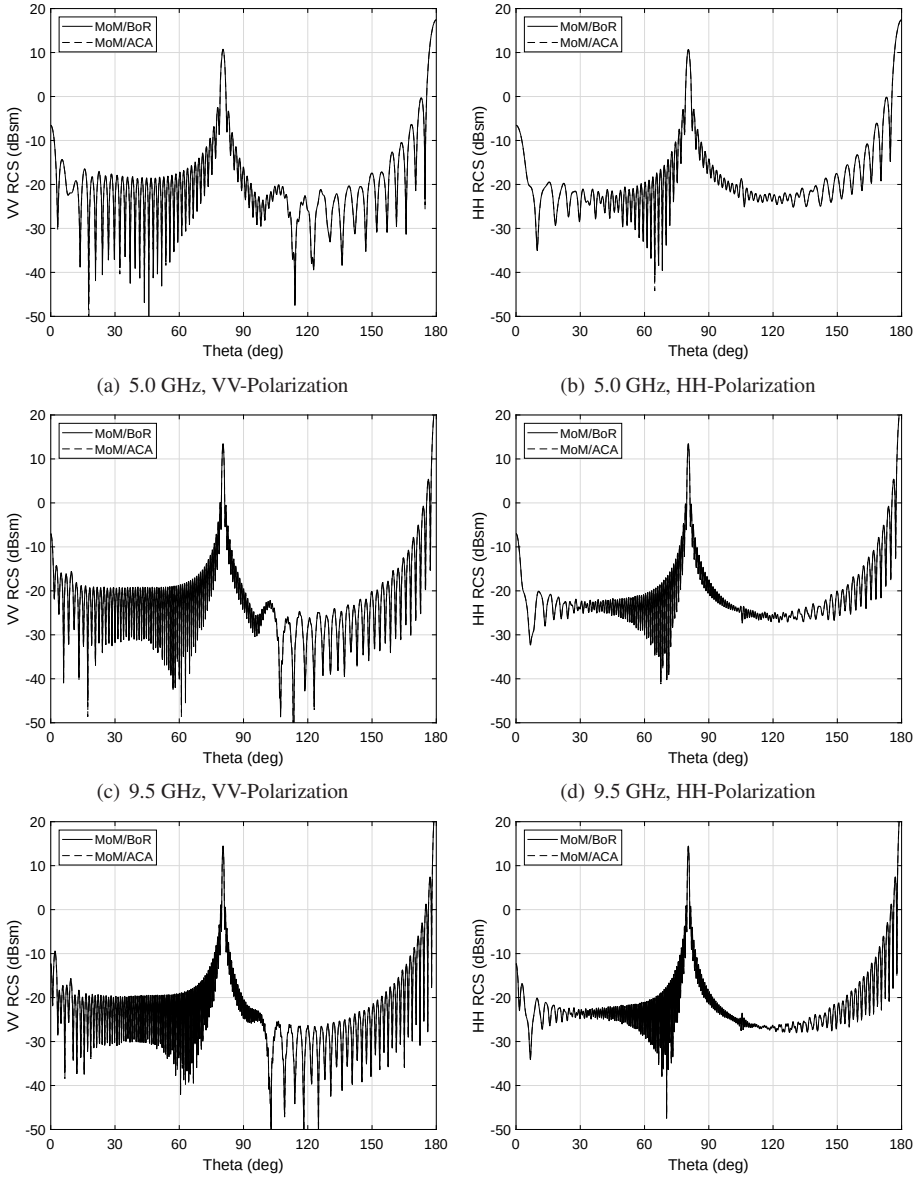
Finally, we will examine the monoconic reentry vehicle considered previously in Sections 7.10.2.4 and 8.8.8. We consider two configurations: the totally conducting RV, and the RV with a dielectric nose. As before, we will compare the results from Serenity to those from our *Galaxy* MoM/BoR solver, as there are no measurements available. For these examples, we use a value of $\tau_{ACA} = 10^{-4}$.

9.7.7.1 Conducting RV

We first consider the conducting RV, which has exterior dimensions identical to those in Figure 7.28. We compute the monostatic RCS at 5.0, 9.5 and 12.0 GHz, using facet models having 144592, 480592, and 776080 triangles. This results in 216888, 720888, and 1171872 unknowns, respectively. The results are compared for vertical and horizontal polarizations in Figures 9.32a–9.32f, where the comparison is very good.

9.7.7.2 RV with Dielectric Nose

We next consider the RV with a lossless dielectric nose ($\epsilon_r = 5$), with dimensions as in Figure 7.28. We compute the monostatic RCS at 5.0, 9.5 and 12.0 GHz, using facet models having 145760, 484384, and 781740 triangles, respectively. These comprise the total number of facets used for the nose, base and septum interfaces, resulting in a total of 225536, 751232, and 1213896 unknowns, respectively. The results are compared for vertical and horizontal polarizations in Figures 9.33a–9.33f, where the comparison is fairly good, although there is some disagreement at nose-forward viewing angles. We suspect that this is due to the facetization density on the nose and septum, though we did not investigate the issue further. Compared to the fully conducting RV, there is a significant increase in RCS at nose-forward angles due to multiple bounces inside the dielectric nose tip.



(a) 5.0 GHz, VV-Polarization

(b) 5.0 GHz, HH-Polarization

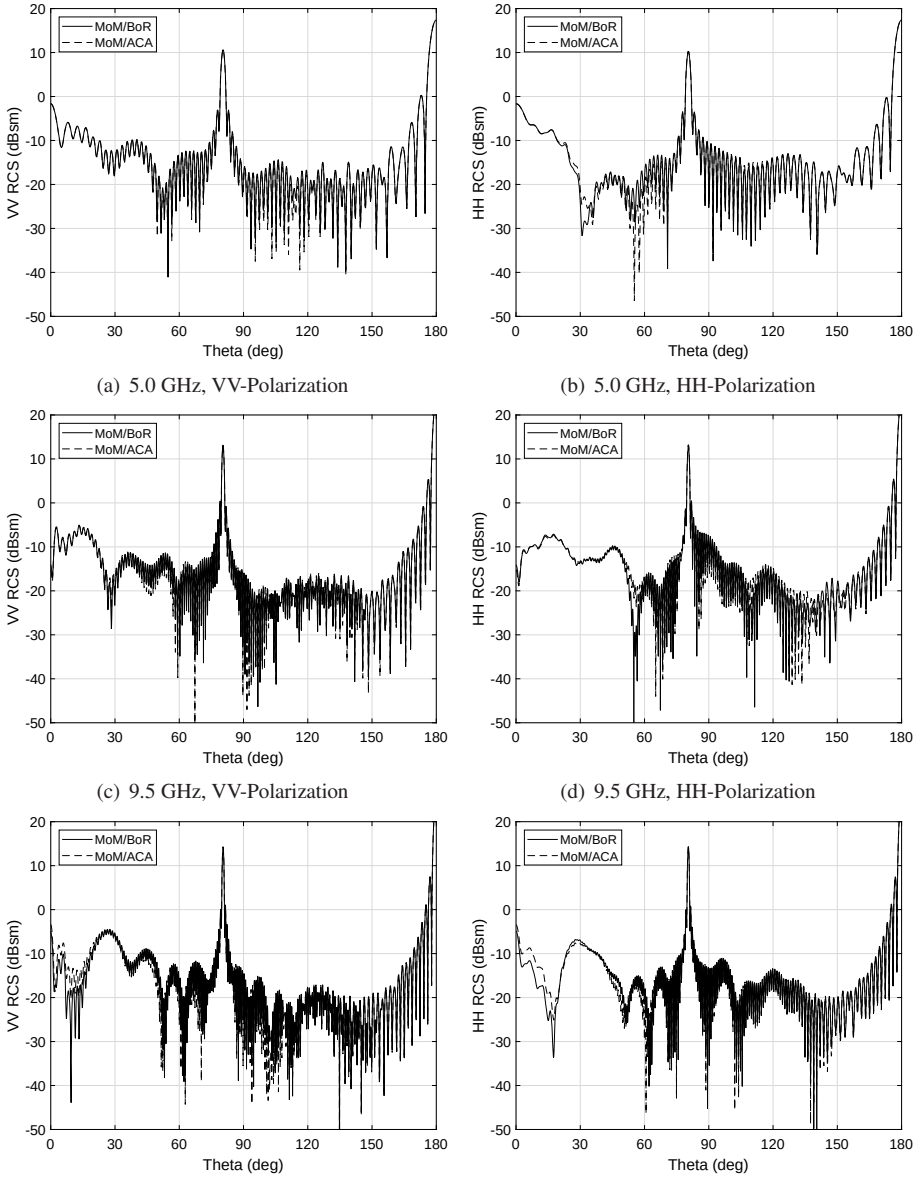
(c) 9.5 GHz, VV-Polarization

(d) 9.5 GHz, HH-Polarization

(e) 12.0 GHz, VV-Polarization

(f) 12.0 GHz, HH-Polarization

Figure 9.32: Conducting RV: RCS at 5.0, 9.5 and 12.0 GHz



(a) 5.0 GHz, VV-Polarization

(b) 5.0 GHz, HH-Polarization

(c) 9.5 GHz, VV-Polarization

(d) 9.5 GHz, HH-Polarization

(e) 12.0 GHz, VV-Polarization

(f) 12.0 GHz, HH-Polarization

Figure 9.33: RV with Lossless Nose: RCS at 5.0, 9.5 and 12.0 GHz

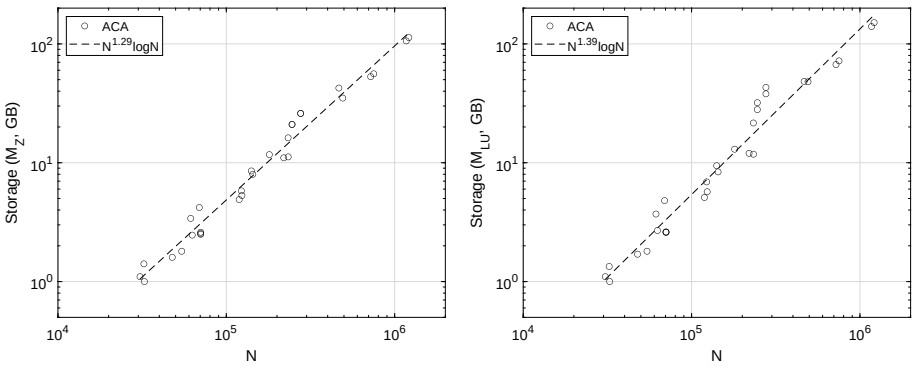
9.7.8 Summary of Examples

Run metrics for the examples in this section are listed in Table 9.1. For problems that used the same facet model at several frequencies, only the results at the highest frequency are shown. Summarized are the operating frequency f_0, number of electric and magnetic basis functions (N_J and N_M), storage for the ACA-compressed matrix ($M_{\mathbf{Z}}$), matrix fill time ($T_{\mathbf{Z}}$), storage for the ACA-compressed **LU** matrix ($M_{\mathbf{LU}}$), LU factorization time ($T_{\mathbf{LU}}$), and amount of compression of the **LU** matrix (C). Frequency is in GHz, storage in GB, time in seconds, and compression in percent. Compression is computed as

$$C = (1 - M_{\mathbf{LU}}/M_0) \cdot 100\% \tag{9.39}$$

where M_0 is the memory required to store the matrix in uncompressed form. We note that the level of compression is excellent overall, and increases with electrical size, exceeding 98 percent in some cases. The PEC RV, for example, having 1171872 basis functions at 12.0 GHz would require approximately 10 TB of memory for the **LU** matrix if stored in uncompressed form. Solving this directly would require supercomputing resources out of the reach of all but the most privileged users. Using the ACA, the storage requirement was reduced to 140 GB, and the problem was solved on our desktop workstation in under two hours.

In Figures 9.34a and 9.34b are shown $M_{\mathbf{Z}}$ and $M_{\mathbf{LU}}$ versus the number of unknowns N, for all the test cases in Table 9.1. We see that the storage requirement of $M_{\mathbf{Z}}$ scales approximately as $O(N^{1.29} \log N)$, slightly less than the $O(N^{4/3} \log N)$ reported for the ACA previously in [3]. The storage requirement of $M_{\mathbf{LU}}$ scales approximately as $O(N^{1.39} \log N)$, slightly higher than $O(N^{4/3} \log N)$.



(a) $M_{\mathbf{Z}}$ versus N (b) $M_{\mathbf{LU}}$ versus N

Figure 9.34: ACA Storage versus Number of Unknowns

Table 9.1: ACA Run Metrics

Object	f_0	N_J	N_M	$M_{\mathbf{Z}}$	$T_{\mathbf{Z}}$	$M_{\mathbf{LU}}$	$T_{\mathbf{LU}}$	$C(\%)$
PEC Sphere	1.5	30720	0	1.1	4.8	1.1	5.7	84.3
PEC Sphere	3.0	122880	0	5.8	20	6.9	35	93.9
PEC Sphere	6.0	491520	0	35	114	48	650	97.3
Lossless Dielectric Sphere	1.5	30720	30720	3.4	19	3.7	23	86.9
Lossless Dielectric Sphere	3.0	122880	122880	21	125	32	176	92.9
Lossy Dielectric Sphere	3.0	122880	122880	21	123	28	162	93.7
Lossless Coated Sphere	1.5	38400	30720	4.2	23	4.8	30	86.5
Lossless Coated Sphere	3.0	153600	122880	26	157	43	243	92.5
Lossy Coated Sphere	3.0	153600	122880	26	156	38	225	93.4
Almond	7.0	70476	0	2.5	10	2.6	15	93.2
Almond	9.92	70476	0	2.6	11	2.6	15	93.0
Ogive	9.0	32640	0	1.0	7.2	1.0	7.5	87.7
Double Ogive	9.0	54360	0	1.8	8.0	1.8	11	91.7
Cone-Sphere	9.0	119244	0	4.9	17	5.1	31	95.2
Cone-Sphere with Gap	9.0	123792	0	5.3	19	5.7	39	95.2
PEC Cube	0.43	6813	0	0.1	1	0.1	1	72.0
PEC Cube	1.3	62717	0	2.5	15	2.7	25	90.0
Prism	0.43	5415	0	0.063	1	0.062	1	72.0
Prism	1.3	47745	0	1.6	14	1.7	18	90.2
Small Polyethylene Cube	18.0	16200	16200	1.41	21	1.34	28	82.9
Polyethylene Ogive	18.0	116436	116436	16.2	110	21.6	234	94.7
Austin PEC Almond	10.25	70476	0	2.54	14.4	2.62	23	92.9
Austin Solid Resin Almond	9.25	70476	70476	8.53	65.1	9.44	121	93.6
Austin Closed-Tail Almond	10.25	95920	47752	7.99	62	8.37	142	94.6
Austin Open-Tail Almond	10.25	132460	47752	11.7	110	13.0	200	94.6
PEC Aircraft	10.25	233148	0	11.2	74	11.8	202	97.0
Solid Resin Aircraft	10.25	233148	233148	42.5	529	48.3	1238	97.0
PEC RV	5.0	216888	0	11	34	12	94	96.6
PEC RV	9.5	720888	0	53	171	67	1722	98.3
PEC RV	12.0	1171872	0	106	359	140	6500	98.6
Dielectric Nose RV	5.0	219508	6028	11	39	12	103	96.7
Dielectric Nose RV	9.5	730576	20656	56	204	72	1927	98.3
Dielectric Nose RV	12.0	1180548	33348	113	423	151	7365	98.6

References

[1] E. Michielssen and A. Boag, "A multilevel matrix decomposition algorithm for analyzing scattering from large structures," *IEEE Trans. Antennas Propagat.*, vol. 44, pp. 1086–1093, August 1996.

[2] M. Bebendorf, "Approximation of boundary element matrices," *Numer. Math.*, vol. 86, pp. 565–589, June 2000.

[3] K. Zhao, M. N. Vouvakis, and J.-F. Lee, "The adaptive cross approximation algorithm for accelerated method of moments computations of EMC problems," *IEEE Trans. Electromagn. Compat.*, vol. 47, pp. 763–773, Nov 2005.

[4] X. Chen, C. Gu, A. Heldring, Z. Li, and Q. Cao, "Error bound of the multilevel adaptive cross approximation (MLACA)," *IEEE Trans. Antennas Propagat.*, vol. 64, pp. 374–378, January 2016.

[5] A. Heldring, E. Ubeda, and J. M. Rius, "On the convergence of the ACA algorithm for radiation and scattering problems," *IEEE Trans. Antennas Propagat.*, vol. 62, pp. 3806–3809, July 2014.

[6] A. Heldring, E. Ubeda, and J. M. Rius, "Stochastic estimation of the frobenius norm in the ACA convergence criterion," *IEEE Trans. Antennas Propagat.*, vol. 63, pp. 1155–1158, March 2015.

[7] M. Bebendorf and S. Kunis, "Recompression techniques for adaptive cross approximation," *J. Integral Equations Applications*, vol. 21, pp. 331–357, July 2009.

[8] J. Shaeffer, "Direct solve of electrically large integral equations for problem sizes to 1 M unknowns," *IEEE Trans. Antennas Propagat.*, vol. 56, pp. 2306–2313, August 2008.

[9] E. Lezar and D. Davidson, "GPU-accelerated method of moments by example: Monostatic scattering," *IEEE Antennas Propagat. Mag.*, vol. 52, pp. 120–135, December 2010.

[10] B. Virk, "Implementing Method of Moments on a GPGPU using NVIDIA CUDA," Master's thesis, Georgia Institute of Technology, May 2010.

[11] CUDA C Programming Guide, docs.nvidia.com/cuda/cuda-c-programming-guide.

[12] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with gpus," *Numerical Computations with GPUs*, pp. 1–26, 2014.

[13] OpenMPI, www.open-mpi.org.

[14] MPI Forum, www.mpi-forum.org.

[15] A. C. Woo, H. T. G. Wang, M. J. Schuh, and M. L. Sanders, "Benchmark radar targets for the validation of computational electromagnetics programs," *IEEE Antennas Propagat. Magazine*, vol. 35, pp. 84–89, February 1993.

[16] A. Greenwood, "Electromagnetic Code Consortium Benchmarks," Tech. Rep. AFRL-DE-TR-2001-1086, Air Force Research Laboratory, December 2001.

[17] J. Song, C. C. Lu, W. C. Chew, and S. W. Lee, "Fast Illinois solver code (FISC)," *IEEE Antennas Propagat. Mag.*, vol. 40, no. 3, pp. 27–34, 1998.

[18] J. M. Parks, "Scattering from Dielectric Bodies," Master's thesis, Air Force Institute of Technology, December 1997.

[19] github.com/UTAustinCEMGroup/AustinCEMBenchmarks.

[20] J. T. Kelley, D. A. Chamulak, C. C. Courtney, and A. E. Yilmaz, "Rye canyon radar cross-section measurements of benchmark almond targets," *IEEE Antennas Propagat. Mag.*, vol. 52, pp. 120–135, February 2020.

[21] J. T. Kelley, D. A. Chamulak, C. C. Courtney, and A. E. Yilmaz, "Measurements of non-metallic targets for the Austin RCS benchmark suite," in *Proc. Ant. Meas. Tech. Assoc. (AMTA) Symp*, 2019.

[22] J. T. Kelley, A. Maicke, D. A. Chamulak, C. C. Courtney, and A. E. Yilmaz, ""Adding a reproducible airplane model to the Austin RCS benchmark suite," in *Proc. Applied Comp. Electromagnetics Society (ACES) Symp.*, July 2020.