



UNIVERSITEIT•STELLENBOSCH•UNIVERSITY  
jou kennisvennoot • your knowledge partner

# **Radar Cross Section Prediction of Arbitrary Shaped Targets**

Cullen Stewart-Burger  
20751028

Report submitted in partial fulfilment of the requirements of the module  
Project (E) 448 for the degree Baccalaureus in Engineering in the Department of  
Electrical and Electronic Engineering at Stellenbosch University.

Supervisor: Dr D. J. Ludick

November 2020

# Acknowledgements

I would like to thank my supervisor, Dr. Danie Ludick for his guidance and assistance over the course of this project.

I would also like to thank Dr. Jacques Cilliers and Dr. Monique Potgieter from the CSIR for the help they gave me in learning and understanding the fundamentals of radar and computational electromagnetics, as well as their comments and assistance in the completion of this report.

Finally, thank you to my parents for their continual love and support and the opportunities they gave me to get to this point.



UNIVERSITEIT • STELLENBOSCH • UNIVERSITY  
jou kennisvennoot • your knowledge partner

## Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.

*Plagiarism is the use of ideas, material and other intellectual property of another's work and to present it as my own.*

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.

*I agree that plagiarism is a punishable offence because it constitutes theft.*

3. Ek verstaan ook dat direkte vertalings plagiaat is.


*I also understand that direct translations are plagiarism.*

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelike aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.

*Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.*

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.

*I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.*

20751028 Studentenommer / <i>Student number</i>	 Handtekening / <i>Signature</i>
C.D. Stewart-Burger Voorletters en van / <i>Initials and surname</i>	09 November 2020 Datum / <i>Date</i>

# Abstract

## English

The radar cross section (RCS) of a radar target is a difficult and potentially expensive quantity to measure. Computational electromagnetics (CEM) simulations are commonly used to predict this quantity without the need for extensive experimental measurements. In this work, a CEM solver based on the physical optics approximation is developed for the RCS prediction of electrically large problems. The solver is extended to model multiple reflections to increase the accuracy of the RCS prediction of complex targets. The solver implemented in this work is shown to perform with an accuracy similar to the physical optics solver implemented in FEKO, a well known and widely used commercial CEM package.

## Afrikaans

Die radar-deursnit van 'n radar-teiken is 'n moeilike en potensieel duur hoeveelheid om te meet. Berekenings-elektromagnetiese simulاسies word gewoonlik gebruik om hierdie hoeveelheid te voorspel sonder dat uitgebreide eksperimentele metings nodig is. In hierdie projek word 'n berekeningsoplosser vir elektromagnetika gebaseer op die fisiese optiese benadering ontwikkel vir die voorspelling van radar-deursnit van elektries groot probleme. Die oplossing word uitgebrei om meerdere weerkaatsings te modelleer om die akkuraatheid van die radar-deursnee-voorspelling van komplekse teikens te verhoog. Die oplossing wat in hierdie werk geïmplementeer word toon akkuraatheid soortgelyk aan die fisiese optika oplossing in FEKO, 'n bekende en wyd gebruikte kommersiële berekenings-elektromagnetika pakket.

# Contents

<b>Declaration</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>Nomenclature</b>	<b>viii</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Background . . . . .	1
1.2. Project objectives . . . . .	2
1.3. Scope . . . . .	3
1.4. Structure of this report . . . . .	3
<b>2. Literature Review</b>	<b>4</b>
2.1. Summary of computational electromagnetics methods . . . . .	4
2.1.1. Full-wave methods . . . . .	4
2.1.2. Asymptotic methods . . . . .	5
2.2. Rao-Wilton-Glisson basis functions . . . . .	6
2.3. Methods of determining illumination . . . . .	7
2.3.1. Buffer-based shadowing algorithms . . . . .	8
2.3.2. Ray-based shadowing algorithms . . . . .	8
<b>3. Project Overview</b>	<b>9</b>
3.1. The SUN-EM and FEKO interface . . . . .	9
3.2. Software structure . . . . .	9
<b>4. Single Reflection Physical Optics Assuming Full Illumination</b>	<b>11</b>
4.1. Calculating the surface current according to the physical optics approximation	11
4.2. Calculating the scattered electric field . . . . .	13
4.3. Basic physical optics implementation . . . . .	13
4.4. Validation . . . . .	15

<b>5. Single Reflection Physical Optics with Shadowing</b>	<b>17</b>
5.1. Determining illuminated elements . . . . .	17
5.1.1. Interfacing with NVIDIA OptiX Prime . . . . .	17
5.1.2. Using the ray-tracer to identify illuminated basis functions . . . . .	18
5.2. Comparison with full illumination assumption . . . . .	20
<b>6. Multiple Reflection Physical Optics</b>	<b>22</b>
6.1. Modelling multiple reflections . . . . .	22
6.2. Determining the visibility matrix . . . . .	23
6.3. Calculating the reflected magnetic field . . . . .	25
6.4. Results . . . . .	27
6.4.1. Comparison with single reflection algorithm . . . . .	27
6.4.2. Testing polarisation results . . . . .	28
6.4.3. Application to non-idealised radar targets . . . . .	29
6.4.4. Performance . . . . .	30
<b>7. Bistatic Radar Cross Section Calculations</b>	<b>32</b>
7.1. Results . . . . .	33
<b>8. Summary and Conclusion</b>	<b>35</b>
8.1. Improvements and recommendations . . . . .	35
8.1.1. Accuracy based improvements . . . . .	35
8.1.2. Performance based improvements . . . . .	36
<b>Bibliography</b>	<b>37</b>
<b>A. Project Planning Schedule</b>	<b>40</b>
<b>B. Outcomes Compliance</b>	<b>41</b>
<b>C. <i>runPOsolver.m</i></b>	<b>43</b>
<b>D. <i>CalcRCS.m</i></b>	<b>48</b>
<b>E. <i>selfShadow.m</i></b>	<b>51</b>
<b>F. <i>Raytracer.m</i></b>	<b>54</b>
<b>G. <i>mexRaytracer.cpp</i></b>	<b>55</b>

# List of Figures

2.1. RWG basis function . . . . .	6
2.2. Sphere illuminated by a plane wave showing illuminated and shadowed regions. . . . .	8
3.1. High level software structure . . . . .	10
4.1. Dipole model . . . . .	13
4.2. Basic PO implementation . . . . .	14
4.3. RCS of a square plate . . . . .	15
4.4. RCS of a disc . . . . .	16
5.1. Shadowing algorithm implementation . . . . .	19
5.2. Testing the shadowing algorithm on a simple target . . . . .	20
5.3. Testing the shadowing algorithm on a complex target . . . . .	20
5.4. Comparison of SRPO with and without shadowing . . . . .	21
6.1. Pseudo code for the construction of the visibility matrix. . . . .	25
6.2. Evaluating the dipole model for reflected field calculations . . . . .	26
6.3. RCS of a trihedral . . . . .	28
6.4. Polarisation test results . . . . .	29
6.5. RCS of a generic cruise missile . . . . .	30
7.1. Bistatic Radar . . . . .	32
7.2. Bistatic RCS of a trihedral . . . . .	33
7.3. Bistatic RCS of a generic cruise missile . . . . .	33

# List of Tables

1.1. Definition of quantities used in the radar equation. . . . .	1
6.1. Results of orientation test between two RWG elements . . . . .	25
6.2. Calculation time and memory usage for the trihedral. . . . .	30
6.3. Calculation time and memory usage for the generic cruise missile. . . . .	31
A.1. Week by week breakdown of the project planning schedule. . . . .	40
B.1. ECSA Exit Level Outcomes (ELO) compliance. . . . .	41



# Nomenclature

## Variables and functions

$\sigma$	Radar cross section.
$P_r$	Reflected power.
$P_t$	Transmitted power.
$G_t$	Transmitter gain.
$A_r$	Effective area of receiving antenna.
$F$	Propagation factor.
$R_t$	Target distance to transmitter.
$R_r$	Target distance to receiver.
$\omega$	Angular frequency.
$\mathbf{J}$	Surface current density.
$\eta_0$	Intrinsic impedance of free space.
$\mathbf{k}$	Wave vector.
$\lambda$	Wavelength.
$\mathbf{H}$	Magnetic field.
$\mathbf{E}$	Electric field.

**Acronyms and abbreviations**

RCS	Radar cross section.
PO	Physical Optics.
MoM	Method of moments.
MLFMM	Multi-level, fast multipole method.
GO	Geometric Optics.
CEM	Computational electromagnetics.
SRPO	Single reflection physical optics.
MRPO	Multiple reflection physical optics.
SBR	Shooting and bouncing rays.
GTD	Geometric theory of diffraction.
UTD	Uniform theory of diffraction.
RWG	Rao-Willton-Glisson.
CAD	Computer aided design.
PEC	Perfect electrical conductor.
3D	Three dimensional.
LE-PO	Large element physical optics.
FDTD	Finite difference time domain method.
FEM	Finite element method.
PTD	Physical theory of diffraction.
CPU	Central processing unit.
GPU	Graphical processing unit.
BVH	Bounding volume hierarchy.
API	Application programming interface.
CUDA	Compute unified device architecture.

# Chapter 1

## Introduction

### 1.1. Background

Radar is a widely used sensor, with both commercial and military applications. It has applications including air traffic control, autonomous vehicle navigation, meteorology, mining, security, surveillance and the military. Although radar increasingly finds application in other fields, its original and primary use remains in the detection, identification and tracking of targets. When considering these applications, the radar cross section (RCS) of a target is of particular interest. The radar cross section gives a measure of the detectability of a target. As seen from the radar equation (1.1), the received power  $P_r$  is directly proportional to the RCS  $\sigma$  of a target.

$$P_r = \frac{P_t G_t A_r \sigma F^4}{(4\pi)^2 R_t^2 R_r^2} \quad (1.1)$$

With the quantities defined in table 1.1.

**Table 1.1:** Definition of quantities used in the radar equation.

$\sigma$	Radar cross section.
$P_r$	Reflected power.
$P_t$	Transmitted power.
$G_t$	Transmitter gain.
$A_r$	Effective area of receiving antenna.
$F$	Propagation factor.
$R_t$	Target distance to transmitter.
$R_r$	Target distance to receiver.

Designers of radar systems require information about how the RCS of a target fluctuates with aspect angle or frequency to design for a specific probability of detection. Similarly, designers of stealth platforms are interested in minimising RCS to avoid detection, particularly at aspect angles from which the platform is expected to approach enemy targets. RCS characteristics can be used to provide operators of stealth platforms insight as to from which directions the platform is most detectable. RCS characteristics also play a role in developing models used for classification of targets. These applications make it apparent that knowledge of the RCS and RCS fluctuations of a target is extremely useful information. Measuring the RCS of large targets can be extremely costly, generally

requiring sophisticated and very accurate instruments. Experimentally determining the RCS of a target in its environment (such as a ship sailing or an aircraft flying) is extremely expensive, as it requires multiple runs to achieve full coverage of the target. This is where the use of computational electromagnetics (CEM) methods to perform RCS predictions becomes useful. Additionally, radar designers are often interested in the RCS of enemy platforms that they do not have access to. In these cases, an experimental approach is not possible and CEM becomes the only option for RCS prediction.

Using CEM simulations for RCS prediction is significantly cheaper than an experimental approach, however CEM methods employed can have a high computational complexity. These methods generally involve breaking up the target into a mesh of triangular patches, with the size of the triangular patches varying with wavelength (A typical mesh size is  $\lambda/10$  [1]). Solving problems with targets that are very large relative to wavelength requires solving over many mesh elements, therefore becoming increasingly computationally expensive as electrical size increases - often to the point where full wave CEM methods are completely impractical for determining RCS over full coverage of the target. In an interest to speed up CEM RCS predictions involving electrically large targets, certain asymptotic CEM methods have been developed. One commonly used asymptotic method involves the use of the physical optics (PO) approximation[2]. With methods such as this, designers of both radar systems and stealth platforms can gain insights into the RCS of targets inexpensively and within reasonable time. This helps to rapidly increase development cycles and drive down the cost of development for such systems.

## 1.2. Project objectives

The aim of this project is to implement a CEM solver based on the physical optics (PO) approximation that can reasonably predict the RCS of electrically large perfect electrical conductor (PEC) targets. Development of this solver is broken down into three iterations each with increasing complexity and accuracy. The objectives are:

1. To implement a solver that uses the PO approximation, assuming full illumination of the target. In this iteration, only the first reflection is modelled.
2. To improve the single reflection PO (SRPO) solver by accounting for shadowing. This involves correctly determining which points on a three dimensional (3D) target are illuminated by an incident plane wave.
3. To implement a multiple reflection PO (MRPO) solver. For this, the SRPO algorithm can be expanded to model successive reflections. This extension allows the solver to be used to accurately predict the RCS of complex targets, particularly those with concave features or cavities.

## 1.3. Scope

This project focuses on the implementation of a mesh-based PO solver. The surface current over the mesh is discretised using first order Rao-Wilton-Glisson [3] (RWG) basis functions. While higher order functions exist, they are not considered for this implementation. The CEM solver designed in this project is intended to be used for RCS calculations in the far field. Therefore, the accuracy of the solution in the near-field is not considered. When extending the solver to model multiple reflections, the high computational cost is noted, but a straight-forward implementation is used despite this. The priority when implementing the MRPO solver is to improve upon the accuracy of the SRPO solution, rather than ensuring that the computational cost remains low enough for this method to be viable for problems with a very large electrical size. Additionally, only targets made of perfect electrical conductor (PEC) surrounded by free space are considered in this project.

## 1.4. Structure of this report

Chapter 2 details a number of CEM methods commonly used in RCS prediction, as well as a description of RWG basis functions as a current discretisation scheme. This chapter also describes a number of shadowing algorithms commonly used. Chapter 3 looks at the general structure of the software used in the report, and the interfaces with other existing software. Chapters 4 to 6 deal with the three iterations of the PO solver as discussed in the project objectives. In each of these chapters, the theory of the implementation, implementation details, and numerical results of the particular method are given. Chapter 8 concludes the report, summarising what was achieved and the findings of the report. Possible improvements and recommendations on future work follow in this chapter.

# Chapter 2

## Literature Review

### 2.1. Summary of computational electromagnetics methods

There are two classes of computational electromagnetic methods, viz., full-wave (also known as exact) methods and asymptotic (approximate) methods. The latter exchanges accuracy for computational efficiency, and increases in accuracy as the electrical size of the target increases. The revision of CEM methods in this chapter is based on texts such as [1], [2], [4] and [5].

#### 2.1.1. Full-wave methods

Full-wave methods approximate the solution to Maxwell's equations [6] in either the integral form (eg. method of moments (MoM)) or differential form (eg. finite difference time domain method (FDTD) or finite element method (FEM)).

##### Method of moments

The method of moments (MoM) is a well known technique used to solve electromagnetic boundary or volume integral equations in the frequency domain [7]. The first step in the MoM involves setting up and solving the following equation,

$$\mathbf{Z}\mathbf{I} = \mathbf{V}, \quad (2.1)$$

where  $\mathbf{Z}$  is the impedance matrix,  $\mathbf{I}$  is a vector of RWG expansion coefficients to be calculated and  $\mathbf{V}$  is a voltage excitation vector [8]. Applying the MoM to a problem with  $N$  mesh elements requires storing a matrix in memory with a size of order  $N^2$  [5]. Additionally, using a direct linear equation solver for (2.1) results in a runtime that scales with  $N^3$  [5]. Therefore, applying the MoM to electrically large problems (ie. at high frequencies) is very computationally expensive and results in unrealistic runtimes. Methods such as the multi-level fast multi-pole method (MLFMM) are sometimes applied to the MoM to accelerate solving the moment equation (2.1) [9], however even with such

accelerations lengthy runtimes may render the MoM impractical when applied to certain large problems.

### 2.1.2. Asymptotic methods

Asymptotic methods approximate the solutions to Maxwell's equations. These methods can be divided further into two categories, viz.,

- ray-based methods, which neglect the wave properties of electromagnetism entirely,
- and current-based methods, which form an intermediate between ray-based methods and full-wave methods.

#### Geometric optics [4]

Geometric optics (GO) is a ray-based asymptotic method. Rays are used to trace specular reflections. This method is very fast, however accuracy is compromised. GO is known to sometimes produce infinite scattering results for flat or singularly curved surfaces and does not account for diffraction or other edge effects.

*Extensions to GO:*

Geometric theory of diffraction (GTD) and uniform theory of diffraction (UTD) are two extensions and refinements to GO that account for diffraction.

#### Shooting and bouncing rays [10]

The shooting and bouncing rays (SBR) method is another ray launching method. Rays or ray tubes are launched towards the target from the source at a sufficient density to cover the whole target. The specular reflections of each ray are traced. Equivalent sources are then placed wherever rays interact with the target. The scattered field is calculated by integrating over these equivalent sources.

#### Physical optics [2]

Physical optics (PO) is a current-based method that approximates the solution, assuming no surface current exists over shadowed regions of the target. PO relates the magnetic field to the induced surface current on all points on the target with line-of-sight visibility to the source. PO does not model surface waves or edge diffraction phenomena.

*Extensions to PO:*

Modified PO (MPO) and the physical theory of diffraction (PTD) are improvements to conventional PO that have been proposed to enhance the solution through the inclusion of

edge diffraction phenomena. PTD was used in the design process of the two well known stealth aircraft, the F-117A and B-2 [11].

Multiple reflection physical optics (MRPO) refers to an extension where the present current solution is used with the PO approximation to iteratively model multiple reflections. MRPO significantly increases the accuracy of the solution when considering complex targets with concave features.

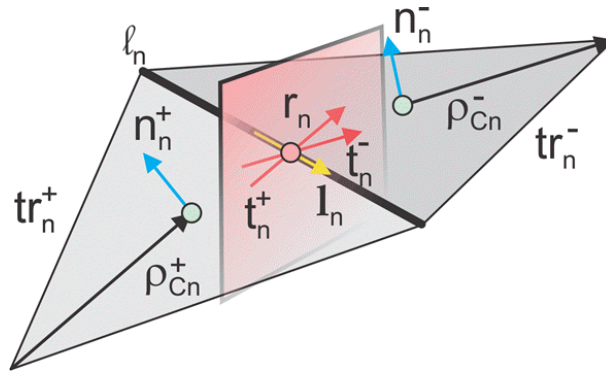
### Large element physical optics [12]

Both the MoM and PO solvers in FEKO [13] employ RWG basis functions which are linear, and do not give any phase variation over an individual mesh element. This means that each mesh element needs to be electrically small (typically  $\lambda/6$  to  $\lambda/12$ ) to give an accurate solution. Large element physical optics (LE-PO) incorporates phase variation of the PO current into the basis functions. This allows the solution to be accurate while using much larger mesh elements ( $\lambda/2$  if near-field sources/observation points are used and limited only by the geometry for far-field calculations). This means that far fewer triangular patches are needed to model electrically large targets. This results in significantly lower memory usage and runtimes compared to standard PO.

As noted in Chapter 1, the use of RWG basis functions is central to the PO implementation followed in this project. This concept is discussed in more detail in the following section.

## 2.2. Rao-Wilton-Glisson basis functions

First order rooftop basis functions, known as Rao-Wilton-Glisson (RWG) [3] basis functions are a well known and widely adopted discretisation scheme. In this project, RWG basis functions are used to describe the surface current density over the supporting mesh. RWG basis functions are defined for each shared (ie. non-boundary) edge within the mesh.



**Figure 2.1:** Two arbitrary triangular patches supporting an RWG basis function [14]



For a given edge  $l_n$  shared by two triangular patches,  $tr_n^+$  and  $tr_n^-$  with areas  $A_n^+$  and  $A_n^-$  respectively as shown in figure 2.1, the RWG basis function is defined as

$$\mathbf{f}_n(\mathbf{r}) = \begin{cases} \frac{l_n}{2A_n^+} \boldsymbol{\rho}_n^+ & \text{for } \mathbf{r} \text{ in } tr_n^+. \\ \frac{l_n}{2A_n^-} \boldsymbol{\rho}_n^- & \text{for } \mathbf{r} \text{ in } tr_n^-. \end{cases} \quad (2.2)$$

$\boldsymbol{\rho}_n^+$  and  $\boldsymbol{\rho}_n^-$  are defined from the free vertex to the centroid of the relevant triangle. Along with this standard definition of an RWG basis function, it is also useful to define two normal vectors  $\mathbf{n}_n^\pm$  and unit vectors  $\mathbf{t}_n^\pm$  as shown in figure 2.1 [15].  $\mathbf{t}_n^+$  and  $\mathbf{t}_n^-$  lie in the plane of  $tr_n^+$  and  $tr_n^-$  respectively, and are perpendicular to the edge  $l_n$  and are both directed from  $tr_n^+$  to  $tr_n^-$ . Vector  $\mathbf{l}_n$  is a unit vector defined by

$$\mathbf{l}_n = \mathbf{t}_n^\pm \times \mathbf{n}_n^\pm \quad (2.3)$$

RWG basis functions have a number of unique properties detailed in [3]. One of these properties is that the current has no component normal to the boundary of the surface formed by the two adjacent triangles  $tr_n^+$  and  $tr_n^-$ . This property becomes particularly useful when applying the PO approximation, as seen later in this report (see Chapter 4). Reference [3] goes on to show that the surface current density  $\mathbf{J}$  over a mesh with  $N$  shared edges can be approximated in terms of  $\mathbf{f}_n(\mathbf{r})$  as shown in equation 2.4, where  $I_n$  is the coefficient associated with basis function  $\mathbf{f}_n$ .

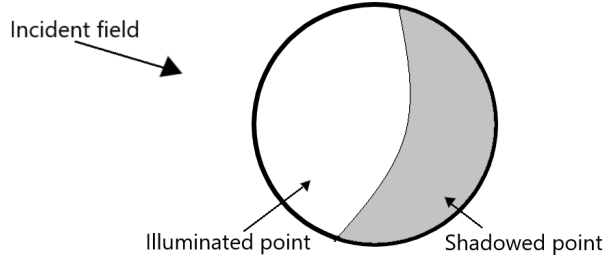
$$\mathbf{J}(\mathbf{r}) \approx \sum_{n=1}^N I_n \mathbf{f}_n(\mathbf{r}) \quad (2.4)$$

The RWG basis functions in (2.4) are used as a discretisation scheme for the surface current over a mesh throughout this report.

## 2.3. Methods of determining illumination

As noted in section 2.1.2, the PO approximation relates the incident magnetic field to the surface current for illuminated points on the target, while setting  $\mathbf{J}(\mathbf{r}) = 0$  for all shadowed points. Figure 2.2 shows what is meant by a shadowed or illuminated point. Applying the PO approximation therefore requires knowledge of which regions are illuminated, and which are shadowed. A number of different approaches exist for this problem.

For closed, convex surfaces illuminated by a plane wave, it is sufficient to perform simple orientation tests by determining the inner product of the outward facing surface normal for each point and the incident plane wave normal. For complex targets or open surfaces, more sophisticated shadowing algorithms are required. The most common approaches are buffer-based and ray-tracing shadowing algorithms.



**Figure 2.2:** Sphere illuminated by a plane wave showing illuminated and shadowed regions.

### 2.3.1. Buffer-based shadowing algorithms

Buffer based algorithms, such as that proposed in [16], create an independent z-buffer split into pixels to store the depth of the surface in each pixel. Each sample point falling within a pixel is compared to the value in the z-buffer. If the sample point is in front of the point stored in the buffer, the buffer is updated to the current sample point. In this way, the nearest points in each buffer are found and considered illuminated, while all other points are considered shadowed. Xiang, [17], proposes another fast buffer based shadowing algorithm, where buffer boxes are set up dynamically to maintain efficiency in cases of grazing incidence. Buffer based methods can be fast, but the creation of the buffer can be complex, as using buffer boxes that are either too large or too small can lead to inaccuracies [16]. The method proposed in [17] also requires homogeneous triangular meshes.

### 2.3.2. Ray-based shadowing algorithms

Ray-based (or ray-tracing) shadowing algorithms are typically more computationally demanding than buffer based methods, but can be more accurate. These methods involve launching a number of rays in the incident direction, and determining the nearest intersection point of each ray with a triangular patch. Ray-tracing algorithms can be accelerated by using bounding volume hierarchies (BVH) [18], which are constructed by creating hierarchical groups of mesh elements. Modern graphical processing units (GPU) can accelerate ray-tracing algorithm through parallelisation, with some GPUs even having hardware acceleration for ray-tracing. Ray-tracing is also more intuitive to use than buffer-based algorithms when determining point-to-point visibility (as is required for MRPO).

# Chapter 3

## Project Overview

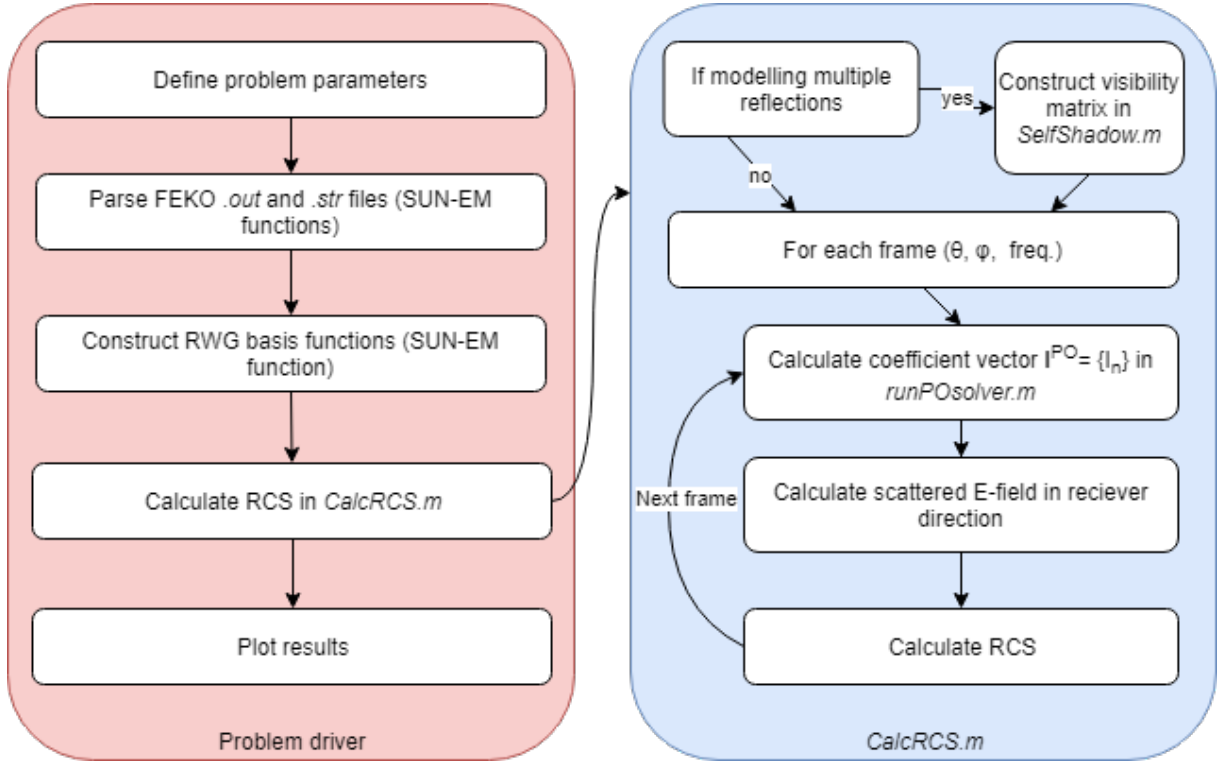
### 3.1. The SUN-EM and FEKO interface

With the focus of the project being on implementing and testing a PO solver, it is desirable to use other pre-existing software for peripheral functions, such as setting up meshes and parsing geometry data. SUN-EM is a MATLAB based CEM package developed by the Computing and Electromagnetics Group at Stellenbosch University [19]. This project was built to be used with this package, contributing a PO solver to the package. The package includes functions used to parse and process certain output files produced by FEKO. This interface with FEKO is used both in the problem set-up (parsing the mesh and other parameters), as well as validating some of the results obtained later. SUN-EM's MoM solver also makes use of RWG basis functions, which allows the reuse of some functions. Some other basic functions are borrowed from this package. The PO solver designed and implemented in this project is later referred to as SUN-EM PO.

### 3.2. Software structure

For each problem, a small control script is set up in MATLAB. Here we define the names of the FEKO output files to be parsed, as well as other problem specific parameters, such as the angular ranges over which the RCS should be calculated. SUN-EM functions are used to parse the the FEKO *.out* and *.str* files, which contain the problem setup (mesh data, frequencies and other constants) and the basis function coefficients  $\{I_n\}$  calculated by FEKO (for verification). From the mesh data, SUN-EM constructs RWG basis functions. When using multiple reflections, the next step is to determine the visibility matrix as discussed in Chapter 6. This step is skipped for SRPO. The *CalcRCS* function then loops over each defined incident angle  $(\theta, \phi)$  and frequency, for which the RCS of the target should be calculated. For each angle, the PO solver, *runPOsolver.m*, is called to calculate the basis function coefficients  $\mathbf{I}^{PO} = \{I_n\}$ . From the PO solution, the scattered field in the receiver direction is calculated. The RCS at the specified angle can then be computed from the scattered field. Finally, once this process has been repeated for each defined incident angle, the code returns to the driver file where the results are plotted.

Figure 3.1 summarises the overarching software structure. The structure of the functions *runPOsolver.m* and *selfShadow.m* is provided in Chapters 4 and 6 respectively. The full



**Figure 3.1:** Flow diagram depicting the software structure on a high level

code listings for the essential functions *CalcRCS.m*, *runPOsolver.m*, *selfShadow.m* and the raytracing source code are given in the appendices.

# Chapter 4

## Single Reflection Physical Optics Assuming Full Illumination

The process of determining the RCS of a target using single reflection physical optics (SRPO) is as follows. Firstly, an incident electromagnetic field is applied to the target. The induced surface current is then calculated according to the PO approximation. From the induced surface current, the scattered electromagnetic field is calculated in the direction of the receiver. In this chapter, it is assumed that all points on the target are illuminated. This assumption eliminates the need for a shadowing function to determine which RWG elements are illuminated.

### 4.1. Calculating the surface current according to the physical optics approximation

Consider a magnetic field  $\mathbf{H}^{\text{inc}}(\mathbf{r})$  incident on a three-dimensional (3D) PEC target with a surface  $\Gamma$  and outward facing surface normal  $\mathbf{n}(\mathbf{r})$ . The PO approximation gives the induced surface current density  $\mathbf{J}(\mathbf{r})$  at a point  $\mathbf{r} \in \Gamma$  on the surface of the target as shown in equation 4.1[2].

$$\mathbf{J}(\mathbf{r}) = 2\delta(\mathbf{r})[\mathbf{n}(\mathbf{r}) \times \mathbf{H}^{\text{inc}}(\mathbf{r})] \quad (4.1)$$

where the continuous shadowing function  $\delta(\mathbf{r})$  is zero for  $\mathbf{r}$  shadowed from the source or  $\pm 1$  depending on whether the point is illuminated from the front or back of the surface (the  $\delta = -1$  case is only needed for cases with open surfaces - closed surfaces can only be illuminated from the front of each triangular patch).

Combining the current discretisation (2.4) and the PO approximation (4.1), we obtain

$$\begin{aligned} \sum_{n=1}^N I_n^{PO} \mathbf{f}_n(\mathbf{r}) &= 2\delta(\mathbf{r})[\mathbf{n}(\mathbf{r}) \times \mathbf{H}^{\text{inc}}(\mathbf{r})] \\ &= \sum_{n=1}^N 2\delta_n[\mathbf{n}(\mathbf{r}_n) \times \mathbf{H}^{\text{inc}}(\mathbf{r}_n)] \end{aligned} \quad (4.2)$$

Also noting the use of a discrete shadowing function  $\delta_n$  defined

$$\delta_n = \begin{cases} 0 & \text{for shadowed RWG elements.} \\ 1 & \text{for RWG elements illuminated from the front.} \\ -1 & \text{for RWG elements illuminated from the back.} \end{cases} \quad (4.3)$$

[14] simplifies equation (4.2) elegantly using the properties of RWG basis functions as follows. Firstly, a point  $\mathbf{r}_n$  located inside of  $tr_n^+$  that tends towards the centre of edge  $l_n$  is considered. Equation (4.2) is dotted with vector  $\mathbf{t}_n^+$ , while noting that the normal component to the basis function at the shared edge is unity and all other basis functions sharing edge  $l_n$  have no normal component. From this dot product, equation (4.4) is obtained.

$$I_n^{PO} = 2\delta_n \mathbf{t}_n^+ \cdot [\mathbf{n}_n^+ \times \mathbf{H}^{\text{inc}}(\mathbf{r}_n)] \quad (4.4)$$

Similarly, a point in  $tr_n^-$  is considered and the same process is followed to obtain

$$I_n^{PO} = 2\delta_n \mathbf{t}_n^- \cdot [\mathbf{n}_n^- \times \mathbf{H}^{\text{inc}}(\mathbf{r}_n)] \quad (4.5)$$

Summing (4.4) and (4.5) and dividing by two while re-arranging the vector products gives

$$I_n^{PO} = \frac{2\delta_n \mathbf{H}^{\text{inc}}(\mathbf{r}_n) \cdot [(\mathbf{t}_n^+ \times \mathbf{n}_n^+) + (\mathbf{t}_n^- \times \mathbf{n}_n^-)]}{2} \quad (4.6)$$

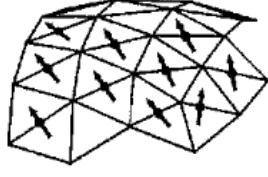
Noting that  $\mathbf{l}_n = \mathbf{t}_n^+ \times \mathbf{n}_n^+ = \mathbf{t}_n^- \times \mathbf{n}_n^-$ , (4.7) simplifies to

$$I_n^{PO} = 2\delta(\mathbf{r}_n) \mathbf{H}^{\text{inc}}(\mathbf{r}_n) \cdot \mathbf{l}_n \quad (4.7)$$

If we assume that the target is fully illuminated, we can set  $\delta_n = 1$  for all  $n$ . With RWG basis functions defined for a given mesh and an expression for the incident magnetic field, we can use equation 4.6 to determine the coefficients  $I_n^{PO}$  and thus an expression for the surface current density (2.4). An expression for the the incident magnetic field  $\mathbf{H}_{\text{inc}}$  can be obtained from the incident electric field  $\mathbf{E}_{\text{inc}}$  for a plane wave travelling in a direction of unit vector  $\mathbf{a}_n$  using the equation,

$$\mathbf{H}_{\text{inc}} = \frac{1}{\eta} \mathbf{a}_n \times \mathbf{E}_{\text{inc}}, \quad (4.8)$$

where  $\eta$  is the impedance of free space.



**Figure 4.1:** Dipole model approximation (from [8] fig. 3.4, p. 42).

## 4.2. Calculating the scattered electric field

It now remains to calculate the scattered electric field from the surface current induced on the target. In [8], the surface current for each RWG element is modelled with an infinitesimal dipole with an equivalent dipole moment, shown in figure 4.1. This approximation simplifies the calculation of the scattered field, and is sufficiently accurate in the far-field to be used in RCS calculations. From [8], the dipole moment  $\mathbf{m}$  for each RWG element is calculated as

$$\mathbf{m} = \int_{Tr_n^+ + Tr_n^-} I_n \mathbf{f}_n(\mathbf{r}) dS = l_n I_n (\mathbf{r}_n^{c-} - \mathbf{r}_n^{c+}) \quad (4.9)$$

Where  $\mathbf{r}_n^{c+}$  and  $\mathbf{r}_n^{c-}$  are the centroids of  $tr_n^+$  and  $tr_n^-$  respectively and  $l_n$  is the dipole length  $|\mathbf{r}_n^{c-} - \mathbf{r}_n^{c+}|$ . [8] goes on to show that the scattered electric field at  $\mathbf{r}$  from a single element is given by

$$\mathbf{E}_n(\mathbf{r}) = \frac{\eta_0}{4\pi} \left( (\mathbf{M} - \mathbf{m}) \left[ \frac{jk}{r} + C \right] + 2\mathbf{M}C \right) e^{-jkr} \quad (4.10)$$

with

$$C = \frac{1}{r^2} \left[ 1 + \frac{1}{jkr} \right], \quad \mathbf{M} = \frac{(\mathbf{r} \cdot \mathbf{m})\mathbf{r}}{r^2} \quad (4.11)$$

The total electric field at  $\mathbf{r}$  is calculated by summing over the all RWG elements comprising the target mesh (4.12).

$$\mathbf{E}_{\text{scat}}(\mathbf{r}) = \sum_{n=1}^N \mathbf{E}_n \left( \mathbf{r} - \frac{1}{2}(\mathbf{r}_n^{c+} + \mathbf{r}_n^{c-}) \right) \quad (4.12)$$

The scattered electric field at the receiver is used to calculate the RCS according to equation (4.14). The following section explains how these calculations are implemented in MATLAB code to predict the RCS of a target.

## 4.3. Basic physical optics implementation

The calculation of the RWG basis function coefficients  $\{I_n^{PO}\}$  for each angle of incidence are calculated in *runPOsolver.m* as follows. While most of the variables linked to the RWG basis functions are calculated by a SUN-EM function when the mesh is parsed,

this does not include the calculation of vector  $\mathbf{l}_n$  used in equation 4.6. Before 4.7 can be implemented, we need to calculate this vector for each RWG element. As vector  $\mathbf{l}_n$  is defined in the clockwise direction around  $tr_n^+$ , special attention has to be paid to ensure the correct direction is obtained. To obtain the correct direction,  $\mathbf{l}_n$  is arbitrarily directed from one end of the shared edge to the other end. A second vector  $\mathbf{a}_n$  is then constructed from the same starting vertex as  $\mathbf{l}_n$ , but directed towards the free vertex of  $tr_n^+$ . If  $(\mathbf{a}_n \times \mathbf{l}_n) \cdot \mathbf{n}_n^+ > 0$ , then  $\mathbf{l}_n$  is pointing in the correct direction. Otherwise, we reverse  $\mathbf{l}_n$ .

This implementation assumes a plane wave travelling through empty space with a  $1 \text{ V m}^{-1}$   $\theta$ -polarised electric field. The incident magnetic field at point  $\mathbf{r}$  is therefore calculated as

$$\mathbf{H}(\mathbf{r}) = \frac{1}{\eta_0} e^{j(\mathbf{k} \cdot \mathbf{r})} \cdot \mathbf{a}_\phi \quad (4.13)$$

where  $\eta_0$  is defined as the intrinsic impedance of free space, and  $\mathbf{k}$  is the wave vector and  $\mathbf{a}_\phi$  is a unit vector pointing in the  $\phi$  direction. Using (4.13), the impressed magnetic field at the edge centre  $\mathbf{r}_n$  of each basis function is calculated. Equation 4.7 (with  $\delta_n = 1$  for the full illumination assumption) is used to calculate the coefficients  $I_n^{PO}$  for each basis function.

Once  $I^{PO}$  has been calculated for all basis functions, *runPOsolver.m* returns to *CalcRCS.m* and the scattered field is calculated according with equation 4.12. From the scattered field, the RCS  $\sigma$  can then be calculated as

$$\sigma = \lim_{r \rightarrow \infty} 4\pi r^2 \frac{|E_{scat}|^2}{|E_{inc}|^2} \quad (4.14)$$

In the implementation,  $r$  is set to 100 km from the target, which is large enough to approximate the far field at  $r \rightarrow \infty$ . Figure 4.2 summarises the process carried followed to determine the RCS for each frame (incident angle  $(\theta, \phi)$  and frequency.)

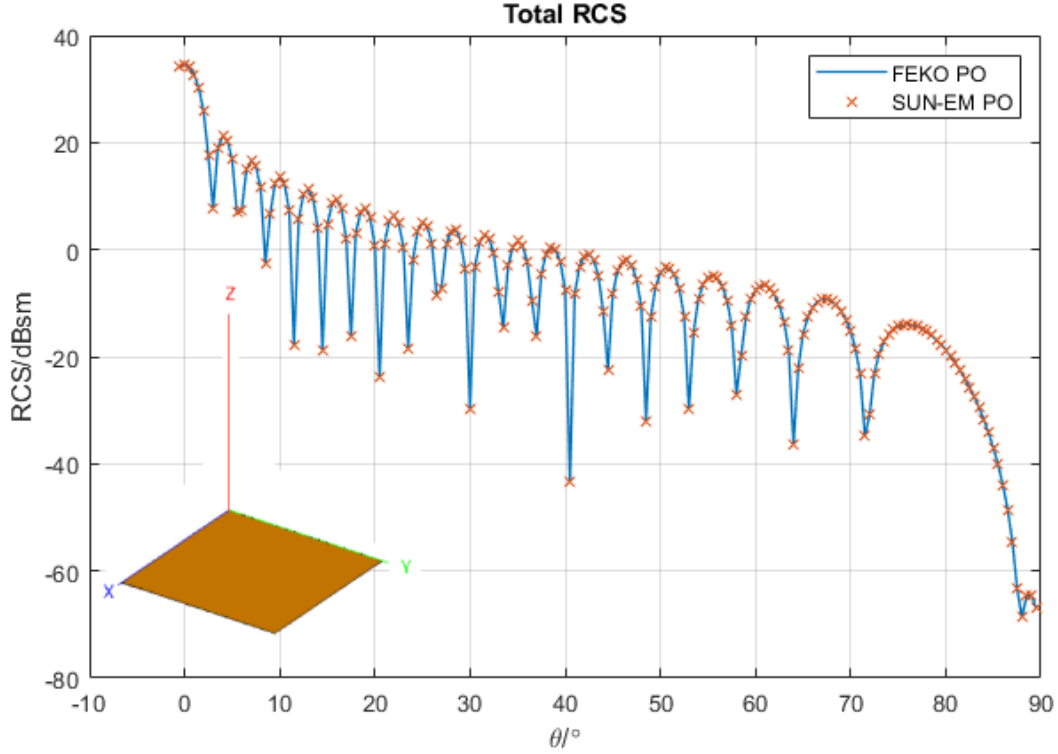
- |  |
|--|
| <ol style="list-style-type: none"> <li>1. Enter <i>runPOsolver.m</i></li> <li>2. Calculate <math>\mathbf{l}_n</math> for each basis function</li> <li>3. Calculate the wave vector <math>\mathbf{k}</math>;</li> <li>4. Calculate the H-field at the edge centre <math>\mathbf{r}_n</math> of each basis function (eq. 4.13)</li> <li>5. Set <math>\delta_n = 1</math></li> <li>6. Calculate the basis function coefficients <math>\{I_n^{PO}\}</math> (eq. 4.7)</li> <li>7. return to <i>CalcRCS.m</i></li> <li>8. Set <math>r = 100\,000\text{m}</math></li> <li>9. Calculate the scattered E-field (eq. 4.12 implemented in SUN-EM function)</li> <li>10. Calculate the RCS (eq. 4.14)</li> </ol> |
|--|

**Figure 4.2:** Pseudo code for the basic PO implementation in *runPOsolver.m* and RCS calculations in *CalcRCS.m*.



## 4.4. Validation

The full illumination assumption holds true for any flat plate. In this section, only such targets are considered. Targets where this assumption fails are discussed in later chapters. The PO implementation was tested over both a frequency sweep, and an angular sweep. Figure 4.3 shows the total monostatic RCS of a  $1\text{ m} \times 1\text{ m}$  square plate at 2 GHz over a  $90^\circ$  elevation sweep from  $\theta = 0^\circ$  to  $\theta = 90^\circ$ . FEKO's PO solution is given as a reference.



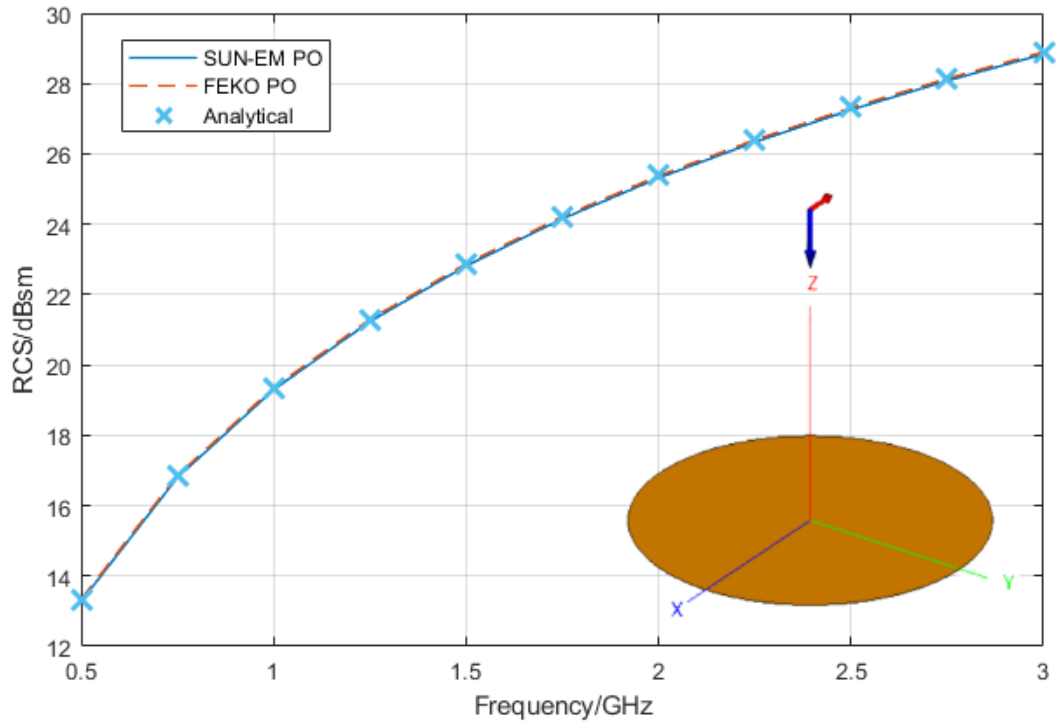
**Figure 4.3:** Total monostatic RCS of a  $1\text{ m} \times 1\text{ m}$  square plate at 2 GHz with FEKO's PO solution as a reference.

From figure 4.3, it can be seen that the SUN-EM implementation agrees strongly with FEKO's solution. The error in  $\mathbf{I}^{PO(SUN-EM)}$  relative to  $\mathbf{I}^{PO(FEKO)}$  is calculated using the  $L^2$ -norm for each incident angle. Expressed as a percentage, this error is on average 0.2%.

The PO implementation is also verified against an analytical solution. According to [20], the analytical solution for the RCS of a disc at normal incidence is given as

$$\sigma = \frac{4\pi S^2}{\lambda^2} \quad (4.15)$$

where  $S$  is the area of the disc, and  $\lambda$  is the wavelength. Figure 4.4 shows the RCS of a disc with  $r = 0.5\text{ m}$  at normal incidence plotted against frequency, calculated using the given PO implementation and compared to the analytical solution calculated with (4.15).



**Figure 4.4:** Total monostatic RCS of a disc with radius 0.5 m at normal incidence vs frequency.

The PO solution is shown to agree with the analytical solution over this frequency range.

# Chapter 5

## Single Reflection Physical Optics with Shadowing

In Chapter 4, it is assumed that all RWG basis functions are illuminated. This assumption simplified the algorithm, making it useful for an initial verification of the method. However, for most 3D targets, this is an unrealistic assumption. In this chapter, a shadowing algorithm is implemented to correctly determine the shadowing coefficient

$$\delta_n = \begin{cases} 0 & \text{for shadowed RWG elements.} \\ 1 & \text{for RWG elements illuminated from the front.} \\ -1 & \text{for RWG elements illuminated from the back.} \end{cases} \quad (5.1)$$

Once this shadowing coefficient is correctly computed, it can be used in the algorithm discussed in chapter 4.

### 5.1. Determining illuminated elements

A number of different shadowing algorithms are discussed in Section 2.3. For this project, it was decided that a ray-tracing approach was the best option to use. Ray-tracing is both accurate, and can be extremely fast when carried on a modern graphical processing unit (GPU), due to the high degree of parallelism in the algorithm. Additionally, there are powerful application programming interfaces (API) available that make ray-launching fairly straightforward to implement. Determining which RWG elements are illuminated can be done according to a number of shadowing rules. In this implementation, the centroids of each triangular patch comprising an RWG element are used as control points. An RWG element is consider illuminated if both centroids are visible.

#### 5.1.1. Interfacing with NVIDIA OptiX Prime

NVIDIA OptiX Prime [21] is a versatile, low-level ray-launching C++ API that supports both GPU and CPU. The low-level API simply supports the construction of an acceleration structure (BVH discussed in section 2.3), and performing ray-triangle intersections using

the acceleration structure. This API was chosen to build a simple ray-tracer because these basic tasks are all that is required to perform simple visibility tests, while doing so efficiently. The ray-tracer is written in C++ and compiled as a MATLAB executable (MEX) file to be used with the rest of the MATLAB code. The basic process for interfacing with the API is as follows.

First, a ‘context’ must be created and initialised. From here on-wards, this context is used to reference the geometry, ray and intersection data passed between the Compute Unified Device Architecture (CUDA) device and the C++ ray-tracer routine. All data passed to and from the context is done through correctly formatted buffers set up through the API’s buffer descriptors. The mesh data (consisting of a list of node co-ordinates and triangle vertex lists) is passed to the C++ API, which then constructs the acceleration structure. Once the acceleration structure is constructed, the ray data (consisting of an origin and direction vector for each ray) can be passed. At this point the context can be used to traverse the acceleration structure, performing ray-triangle intersections between the loaded mesh and rays. Once this process is completed, the context is queried to return the index of the first triangle mesh element each ray intersects.

### 5.1.2. Using the ray-tracer to identify illuminated basis functions

The mesh data parsed from FEKO’s *.out* file can be passed to the ray-tracer with minimal formatting done in MATLAB. Before the rays can be passed to the ray-tracer, however, the rays have to be constructed. Typically, when ray-tracing is performed, rays are either set up in a uniform grid (typically spaced  $\lambda/10$  apart), or in more complex algorithms set up in an adaptive grid where the spacing is finer over smaller mesh elements and coarser over larger mesh elements. This sort of ray construction may be necessary for general ray-tracing or ray intensive techniques such as SBR. However, it is noted that for this application we are only interested in the visibility of the centroid of each triangular mesh element. The minimum number of rays required to carry out this task is therefore equal to the number of triangles in the mesh (ie. one ray per centroid). It follows that the rays can be set up by projecting the centroid of each triangle onto the incident plane, positioned outside the bounding radius of the target geometry. The projection  $\mathbf{p}'$  of point  $\mathbf{p}$  onto a plane with normal  $\mathbf{n}$  distance  $d$  from the origin is given by equation 5.2.

$$\mathbf{p}' = \mathbf{p} - (\mathbf{n} \cdot \mathbf{p} + d)\mathbf{n} \quad (5.2)$$

The projection of each centroid is used as a ray origin point. The ray direction is equal to the incident direction of the plane wave illuminating the target for all rays. Once the mesh and rays are loaded, and the acceleration structure has been traversed, the ray-tracer can be queried to find the index of the triangle the ray first intersects. From this, a list of illuminated triangles is constructed. According to our shadowing rules, basis function  $n$  is

considered illuminated if the centroids of both  $tr_n^+$  and  $tr_n^-$  are visible from the source. We can now set  $\delta_n = 0$  for any basis function that is not illuminated. For RWG basis functions that are illuminated, however, one more subtlety remains. If we are considering targets with open surfaces, we are also interested in whether each basis functions is illuminated from the front or the back. If both triangles are visible, we can assume that they are illuminated from the same side. Orientation can therefore be tested using the dot product of the intersecting ray and the plus triangle outward pointing normal  $\mathbf{n}_n^+$ . In the case where the dot product is greater than 0 the element is illuminated from the back, so we set  $\delta_n = -1$ .  $\delta_n$  is set to +1 for all other illuminated basis functions. Figure 5.1 summarises how the shadowing algorithm was implemented in MATLAB.

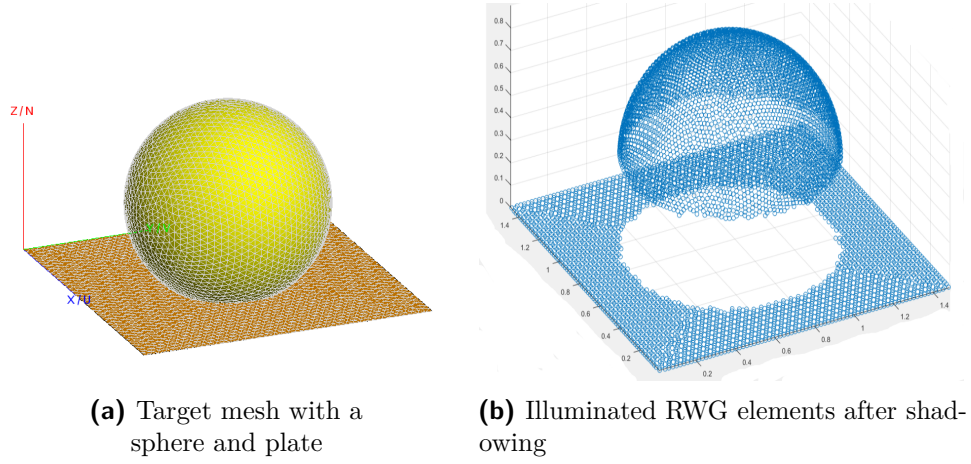
1. Pass mesh to ray-tracer.
2. Determine bounding radius of mesh.
3. Set illumination plane at bounding radius + offset.
4. Project each centroid onto the illumination plane.
5. Set ray origins equal to the projected centroids.
6. Set ray directions equal to the incident direction,  $\mathbf{a}_{inc}$ .
7. Pass rays to ray-tracer.
8. Get list of intersected triangles from ray-tracer.
9. Compare list of visible triangles against  $tr_n^+$  and  $tr_n^-$  for each basis function.
10. Set  $\delta_n = 0$  for all basis functions that aren't visible.
11. For each illuminated basis function, determine the orientation  $\mathbf{n}_n^+ \cdot \mathbf{a}_{inc}$ .
12. For illuminated basis functions, set  $\delta_n = -1$  where the dot product  $> 0$  and  $\delta_n = +1$  otherwise.

**Figure 5.1:** Pseudo code for the shadowing algorithm.

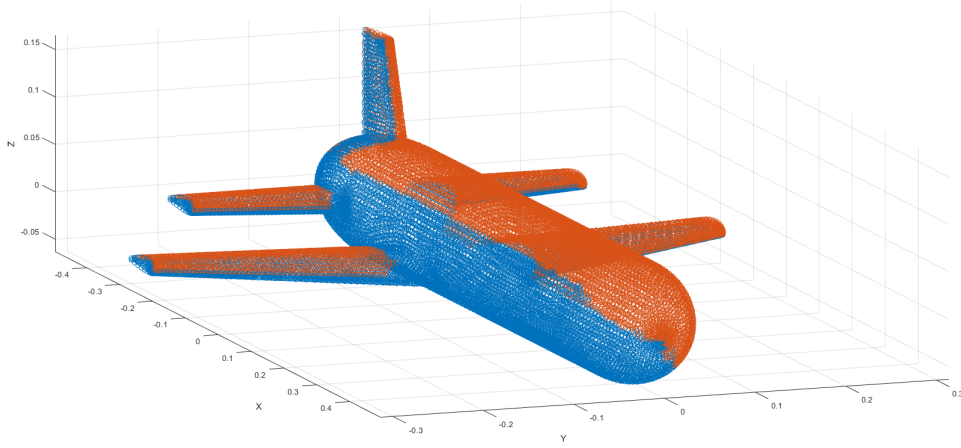
Once  $\delta_n$  has been computed for each basis function, this value can be used in the PO algorithm discussed in chapter 4. The shadowing algorithm therefore replaces step 5 in figure 4.2.

### Verifying the shadowing algorithm

The shadowing algorithm was first tested on some simple targets to ensure that it is working as expected. Figure 5.2 shows a simple target illuminated from the z direction. Figure 5.3 shows the shadowing applied to a complex target illuminated from  $(\theta = 45^\circ, \phi = 45^\circ)$ .



**Figure 5.2:** Testing the shadowing algorithm on a simple target

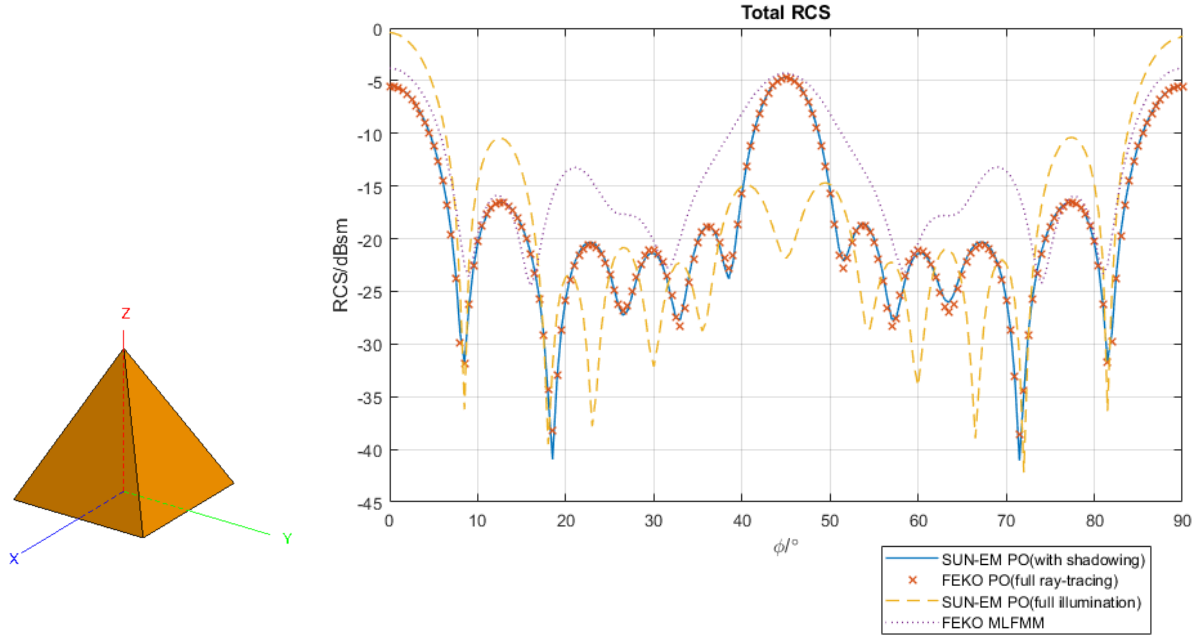


**Figure 5.3:** Shadowing algorithm applied to a complex target. (Red indicates illuminated RWG elements, while blue indicates shadowed RWG elements.)

Both these shadowing tests confirm that the shadowing algorithm is working as expected.

## 5.2. Comparison with full illumination assumption

Both SRPO algorithms with and without the shadowing function are used to determine the RCS of a square-based pyramid target at 1 GHz over a range of azimuth angles. The square based pyramid target has base  $1\text{ m} \times 1\text{ m}$  and height 1 m. Figure 5.4 shows these results, as well as FEKO's PO and MLFMM solution.



(a) Square based pyramid target

(b) Total RCS against azimuth angle  $\phi$

**Figure 5.4:** Comparison of the SRPO algorithms with the full-illumination assumption and with shadowing.

FEKO's MLFMM solution can be taken as a reference here. It can be seen that the algorithm corrected for shadowing is more accurate (ie. closer to the MLFMM solution) than the algorithm with the full-illumination assumption, particularly below  $18^\circ$ , at  $90^\circ$  and above  $72^\circ$ . It is also seen that the SUN-EM implementation with shadowing strongly agrees with FEKO's PO solution.

# Chapter 6

## Multiple Reflection Physical Optics

Up until now, only a single reflection has been considered in the formulation of a PO algorithm. While this is sufficient for convex 3D targets, multiple reflections can have significant contributions to the scattering from concave elements. To implement a solver that is useful in predicting the RCS of complex targets, multiple reflections should be considered.

### 6.1. Modelling multiple reflections

Modelling multiple reflections requires recursively calculating the field radiated by the surface current induced in previous reflections, and applying the PO approximation to the interaction of this reflected field with the target. It is useful to separate the surface currents induced by successive reflections. From here on the notation  $\mathbf{J}_{(k)}$  will be used to denote the surface current contribution from the  $k^{th}$  reflection. Additionally, from (2.4), we can define

$$\mathbf{J}_{(k)}^{PO}(\mathbf{r}) = \sum_{n=1}^N I_{n(k)} \mathbf{f}_{\mathbf{n}}(\mathbf{r}) \quad (6.1)$$

where  $N$  is the number of RWG basis functions.

For now, let us assume that we can calculate the radiated magnetic field  $\mathbf{H}(\mathbf{J}, \mathbf{r})$  from the surface current calculated after the previous reflection (starting with the current  $\mathbf{J}_{(1)}$  calculated in the SRPO algorithm). Using the PO approximation, the recursive formula for the surface current density after the  $k^{th}$  reflection is

$$\mathbf{J}_{(k)}^{PO}(\mathbf{r}) = \mathbf{J}_{(1)}^{PO}(\mathbf{r}) + 2\mathbf{n}(\mathbf{r}) \times \mathbf{H}(\delta_{\mathbf{r}, \mathbf{r}'} \mathbf{J}_{(k-1)}^{PO}(\mathbf{r}'), \mathbf{r}), \quad \{\mathbf{r}' = \Gamma\} \quad (6.2)$$

where  $\delta_{\mathbf{r}, \mathbf{r}'}$  is a shadowing function, equal to one for any two points  $\mathbf{r}$  and  $\mathbf{r}'$  with line-of-sight visibility and zero otherwise, and  $\mathbf{n}(\mathbf{r})$  is the surface normal at  $\mathbf{r}$ .

It is also useful to define the discretised magnetic field at each RWG element  $\mathbf{H}_{\mathbf{n}}$  as the sum of the magnetic fields radiated from each source RWG element  $\mathbf{H}_{\mathbf{m}}$ :

$$\mathbf{H}_{\mathbf{n}}(\mathbf{J}_{(k-1)}^{PO}(\mathbf{r}'), \mathbf{r}_{\mathbf{n}}) = \sum_{m=1}^N \mathbf{H}_{\mathbf{m}}(I_{m(k-1)} \mathbf{f}_{\mathbf{m}}(\mathbf{r}_{\mathbf{m}}), \mathbf{r}_{\mathbf{n}}) \quad (6.3)$$



Applying the RWG basis function discretisation scheme and the simplifications made in section 4.1 to (6.2) combined with (6.3), we obtain

$$\mathbf{J}_{(k)}^{PO}(\mathbf{r}) = \mathbf{J}_{(1)}^{PO}(\mathbf{r}) + \sum_{n=1}^N 2\mathbf{l}_n \cdot \left[ \sum_{m=1}^N \delta_{n,m} \mathbf{H}_m \left( I_{m(k-1)} \mathbf{f}_m(\mathbf{r}_m), \mathbf{r}_n \right) \mathbf{f}_n \right] \quad (6.4)$$

The basis function coefficient  $I_{n(k)}^{PO}$  after  $k$  reflections can therefore be expressed as

$$I_{n(k)}^{PO} = I_{n(1)}^{PO} + 2\mathbf{l}_n \cdot \left[ \sum_{m=1}^N \delta_{n,m} \mathbf{H}_m \left( I_m^{PO} \mathbf{f}_m, \mathbf{r}_n \right) \right] \quad (6.5)$$

In order to implement this formula, we need a method of calculating  $\mathbf{H}_m \left( I_{m(k-1)} \mathbf{f}_m(\mathbf{r}_m), \mathbf{r}_n \right)$ , as well as the inter-edge shadowing coefficient  $\delta_{n,m}$ . Each of these calculations are detailed in the sections below. Once these have been calculated, implementing equation (6.5) becomes straightforward.

## A consideration for Multiple Reflection Physical Optics applied to open surfaces

Only one side of any RWG element can be illuminated when considering MRPO applied to a closed surface or in the SRPO algorithm (as the geometry is illuminated by a plane wave from a single incident angle). This allows us to model the surface current using each RWG basis function once to representing the surface current on the illuminated side (we are uninterested in the surface current on the shadowed side). When considering multiple reflections on open surfaces, however, it is possible for a surface not illuminated by the incident plane wave to be illuminated by a reflection. Therefore, the surface current should be modelled independently for both sides of each RWG element. This can be done by associating two separate basis function coefficients  $I_n^p$  and  $I_n^n$  with each basis function to represent the surface current on the 'positive' and 'negative' surfaces respectively. The 'positive' side is defined by the outward pointing normal defining the associated RWG basis function.

## 6.2. Determining the visibility matrix

Applying the PO approximation requires knowledge of which source basis functions have line-of sight visibility to each observation basis function. For this, we define the inter-edge

shadowing coefficient

$$\delta_{n,m} = \begin{cases} 0 & \text{for } \mathbf{r}_n \text{ shadowed from } \mathbf{r}_m, \text{ or } n = m. \\ 1 & \text{for } \mathbf{r}_n \text{ visible and illuminated from the front from } \mathbf{r}_m. \\ -1 & \text{for } \mathbf{r}_n \text{ visible and illuminated from the reverse from } \mathbf{r}_m. \end{cases} \quad (6.6)$$

In order to determine visibility, we use a similar shadowing rules to those used in the SRPO algorithm to determine the illuminated RWG elements. An RWG element is considered visible from another RWG element if the centroids of both triangles of the observed element have line-of-sight visibility to the edge centre  $\mathbf{r}_n$  of the source element. To determine line-of-sight visibility in this manner, rays are projected from each edge centre of each RWG element to the centroid of each triangle. The ray-tracer detailed in Section 5 is then used to determine the ray-triangle intersections. From this, a  $N \times N$  visibility matrix  $V$  is constructed, where

$$v_{n,m} = \begin{cases} 1 & \text{where } \mathbf{r}_n \text{ has line-of-sight visibility to } tr_m^+ \text{ and } tr_m^-. \\ 0 & \text{otherwise, or } n = m. \end{cases} \quad (6.7)$$

Some strange behaviour was noticed with the visibility testing algorithm. Upon inspection, it was noticed that some rays intersected with the original triangle (ie. did not leave the surface of the target) due to numerical rounding of the ray origin. This is corrected by projecting each ray origin slightly ( $\frac{1}{10}$ ) in the ray direction. This ensures that rays leave the surface from which they originate.

To account for independently modelled currents on both sides of each RWG element, it is also important to perform an orientation test to determine which sides of any two basis functions are facing each other. This could be either positive-positive, positive-negative, negative-positive or negative-negative. In [22], four separate visibility matrices ( $C^{pp}$ ,  $C^{pn}$ ,  $C^{np}$  and  $C^{nn}$ ) are set up for this purpose. These matrices can, however, become very large (as each contain  $N \times N$  elements). In the interest of memory conservation in this project, a single visibility matrix is set up, with values 0 to 4 representing shadowing, positive-positive visibility, positive-negative visibility, negative-positive visibility and negative-negative visibility respectively.

The orientation test between a pair of edges with centres  $\mathbf{r}_n$  and  $\mathbf{r}_m$  is performed using the dot products of the unit vector  $\mathbf{r}_{nm}$  with the each of the basis functions' normal vectors  $\mathbf{n}_n$  and  $\mathbf{n}_m$ . The normal vector for basis function  $p$  is defined as  $\mathbf{n}_n = \frac{\mathbf{n}_n^+ + \mathbf{n}_n^-}{2}$ . A small tolerance term  $\alpha$  is used in the orientation test to ensure that co-planar edges are flagged as 'shadowed'. In this implementation,  $\alpha = \sin(1^\circ)$  was selected (ie. edges that deviate from the same plane by less than  $1^\circ$  are considered co-planar and hence shadowed from one another). This test is outlined in table 6.1.

**Table 6.1:** Results of orientation test between two RWG elements with edge centres  $\mathbf{r}_n$  and  $\mathbf{r}_m$ 

$\mathbf{r}_{nm} \cdot \mathbf{n}_n$	$\mathbf{r}_{nm} \cdot \mathbf{n}_m$	Orientation	Flag in orientation matrix $O$ at $o_{nm}$
$< \alpha$	$> -\alpha$	co-planar	0
$> \alpha$	$< -\alpha$	front-front	1
$> \alpha$	$> \alpha$	front-back	2
$< -\alpha$	$< -\alpha$	back-front	3
$< -\alpha$	$> \alpha$	back-back	4

Element-wise multiplication of the the line-of-sight visibility test matrix  $V$  with the orientation matrix  $O$  results give the final visibility matrix. Figure 6.1 gives a summary of the algorithm used in *SelfShadow.m* to determin the visibility matrix.

1. For each of  $N$  basis functions.
2. Set up  $N_T$  rays from the edge centre of basis function  $n$  to the centroid of each of the  $N_T$  triangular patches in the mesh.
3. Project each ray a distance of  $\frac{l_{min}}{10}$  towards the destination centroid where  $l_{min}$  is the length of the shortest shared edge.
4. Determine the nearest intersection of each ray.
5. Construct column  $n$  of matrix  $V$  according to (6.7) from knowledge of visible triangles.
6. Determine  $\mathbf{r}_{nm} \cdot \mathbf{n}_n$  and  $\mathbf{r}_{nm} \cdot \mathbf{n}_m$  for all the  $M$  other basis functions.
7. Construct column  $n$  of the orientation matrix according to (6.1).
8. Determine the final visibility matrix through element-wise multiplication of  $O$  and  $V$ .

**Figure 6.1:** Pseudo code for the construction of the visibility matrix.

## 6.3. Calculating the reflected magnetic field

The magnetic field at point  $\mathbf{r}$  radiated by surface current  $\mathbf{J}$  is most accurately calculated with the magnetic field integral operator [23],

$$\mathbf{H}(\mathbf{J}, \mathbf{r}) \equiv \int_{supp(\mathbf{J})} \nabla G_0(\mathbf{r}, \mathbf{r}') \times \mathbf{J}(\mathbf{r}') d\mathbf{S}', \quad (6.8)$$

where  $G_0$  is the free space scalar Green function

$$G_0(\mathbf{r}, \mathbf{r}') = \frac{e^{jk_0|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|}, \quad k_0 = \frac{2\pi}{\lambda_0}. \quad (6.9)$$

Evaluating this integral, however, can be computationally expensive. In this work, the dipole model from [8] as used in chapter 4 is used to approximate the magnetic field. Using the dipole model, the field at point  $\mathbf{r}_n$  due to the surface current supported by basis

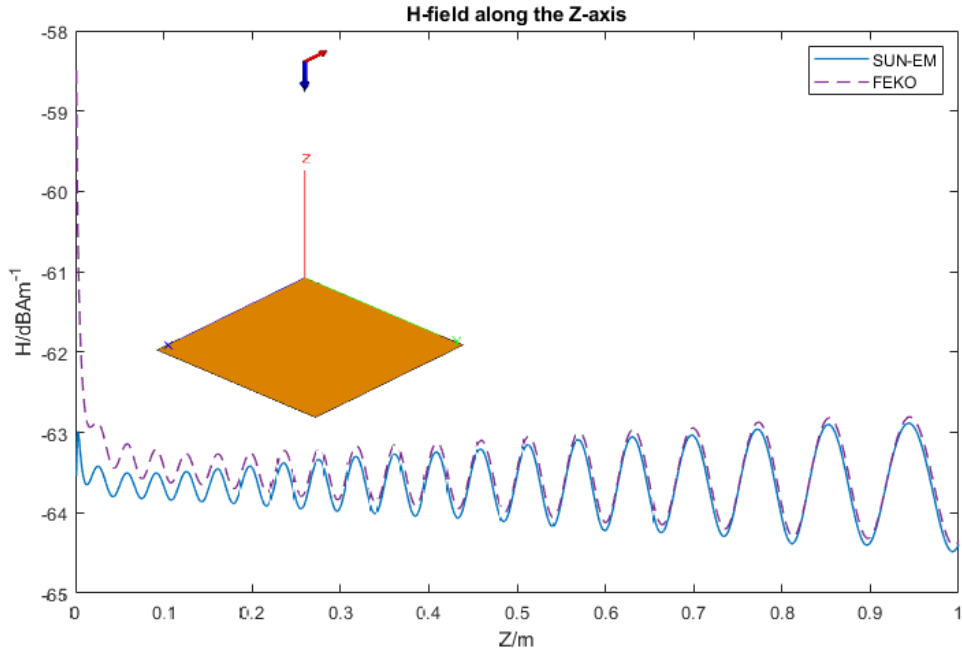
function  $m$ ,  $\mathbf{H}_m(\mathbf{r}_n)$  can be approximated with,

$$\mathbf{H}_m(\mathbf{r}) = \frac{jk}{4\pi}(\mathbf{m} \times \mathbf{r})Ce^{-jkr} \quad (6.10)$$

with  $C$  and  $\mathbf{m}$  defined in (4.9) and (4.11) respectively [8]. Summing over each source basis function gives the total magnetic field  $\mathbf{H}_n$  at  $\mathbf{r}_n$ ,

$$\mathbf{H}_n(\mathbf{r}) = \sum_{m=1}^N \mathbf{H}_m\left(\mathbf{r} - \frac{1}{2}(\mathbf{r}_m^{c+} + \mathbf{r}_m^{c-})\right). \quad (6.11)$$

In [8], it is noted that the dipole approximation is poor for observation distances on the order of the dipole length. Before this approximation is used in the implementation of (6.5), it is worth testing to see how well it performs. A quick test was performed by illuminating a  $1\text{ m} \times 1\text{ m}$  square plate (figure 6.2) with a  $10\text{ GHz}$ ,  $1\text{ V m}^{-1}$  plane wave travelling in the  $-z$  direction. The magnitude of the scattered magnetic field was calculated using the dipole approximation, and compared (figure 6.2) to the the result calculated FEKO using the magnetic field integral operator.



**Figure 6.2:** Comparison of the magnetic field calculated with the dipole model to that calculated in FEKO.

It can be seen that from a couple wavelengths away the dipole model holds up nicely, although in the very near field (less than one wavelength), the results from the approximation differ from the magnetic field integral results by up to approximately  $3\text{ dBAm}^{-1}$ . Despite this error in the very near field, the dipole approximation is considered good enough to be used in the implementation of the MRPO algorithm. Equations (6.10) and (6.11) can be implemented in MATLAB code. A straightforward implementation of

these equations scales with  $O(N^2)$ , as the field has to be calculated at the edge centre of  $N$  basis functions and the calculation at each point requires summation over all  $N$  basis functions. For this project, the scaling was improved slightly. The visibility matrix is calculated before the field calculations are done. It is therefore already known which basis functions have no interactions with one-another (ie. when  $\delta_{nm} = 0$ ). The field calculations can therefore be done by summing only over the basis functions known to have an interaction. Basis functions with  $I_n = 0$  are also excluded from the calculation, further decreasing the number of elements to be summed over.

## 6.4. Results

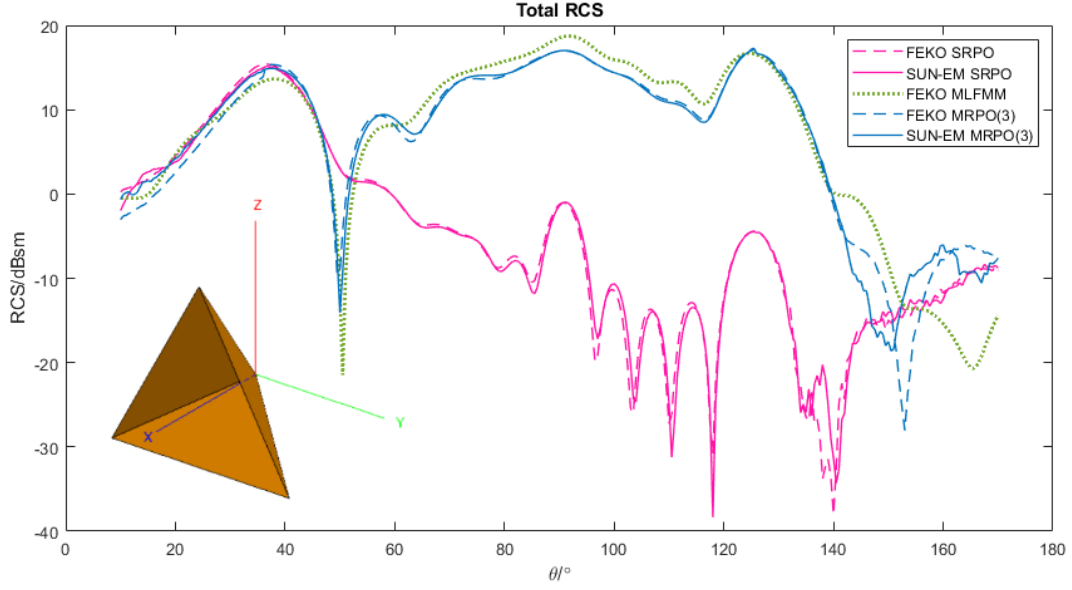
The MRPO implementation was tested with a selection of test targets, aimed at testing specific aspects of the solution. Specifically, we are interested in:

- A comparison between the SRPO and MRPO algorithms, highlighting the differences and confirming that modelling multiple reflections increases the accuracy of the solution.
- Confirming that the MRPO implementation gives the correct polarisation in the solution.
- Gauging how MRPO algorithm performs compared to full-wave solutions when applied to a complex, non-idealised radar target.

### 6.4.1. Comparison with single reflection algorithm

A trihedral corner reflector is a good option as a target for comparing the SRPO and MRPO implementations. When illuminated from the front, the dominant scattering mechanism for a trihedral corner reflector comes from the interactions between the faces (ie. multiple reflections). It is therefore expected that the SRPO algorithm will perform poorly, while MRPO will give significantly more accurate results. The total RCS of a trihedral corner reflector (with edge lengths of 1 m) was calculated as a function of  $\theta$  with an angular resolution of  $0.5^\circ$  at a frequency of 1 GHz using both the SRPO and MRPO algorithms. For the MRPO algorithm, three reflections are used, as required to correctly model the interactions of a trihedral. Figure 6.3 shows these results, along with FEKO's SRPO and MRPO solutions as a reference. The MLFMM solution, as calculated by FEKO, is also given as a reference.

Below  $36^\circ$ , the RCS is dominated by specular scattering. All the methods shown here model specular scattering, and therefore closely agree. Between  $36^\circ$  and  $126^\circ$ , we are in the region where multiple reflection interactions dominate the response. Here we see the MLFMM and MRPO solutions continuing to agree, while the SRPO method performs



**Figure 6.3:** Total monostatic RCS of a trihedral with an edge length of 1 m at 1 GHz.  $\phi = 0^\circ$  for all incident angles.

poorly. This is expected, as these interactions are not modelled by SRPO. Above  $126^\circ$ , edge diffraction effects can be seen in the MLFMM solution. Beyond  $140^\circ$ , these edge diffraction effects become significant to the point where the MLFMM solution and the PO-based solutions, which do not model diffraction, do not agree. In the region between  $36^\circ$  and  $126^\circ$ , the relative error (calculated using the  $L^2$ -norm) between  $\mathbf{I}^{PO(SUN-EM)}$  and  $\mathbf{I}^{PO(FEKO)}$  is approximately 3.6 %. The currents calculated in the SUN-EM MRPO implementation are shown to agree with those calculated by FEKO. These results confirm that modelling multiple reflections using MRPO improves the accuracy of RCS predictions compared to SRPO, as well as showing that MRPO holds up well compared to MLFMM when edge diffraction effects are not significant.

### 6.4.2. Testing polarisation results

The polarisation of the solution is relevant, as many radar antennae only transmit and receive certain polarisations. When considering polarisation, it is useful to refer to a scattering matrix  $A$ , where

$$\begin{bmatrix} E_x^{scat} \\ E_y^{scat} \end{bmatrix} = [A] \begin{bmatrix} E_x^{inc} \\ E_y^{inc} \end{bmatrix}. \quad (6.12)$$

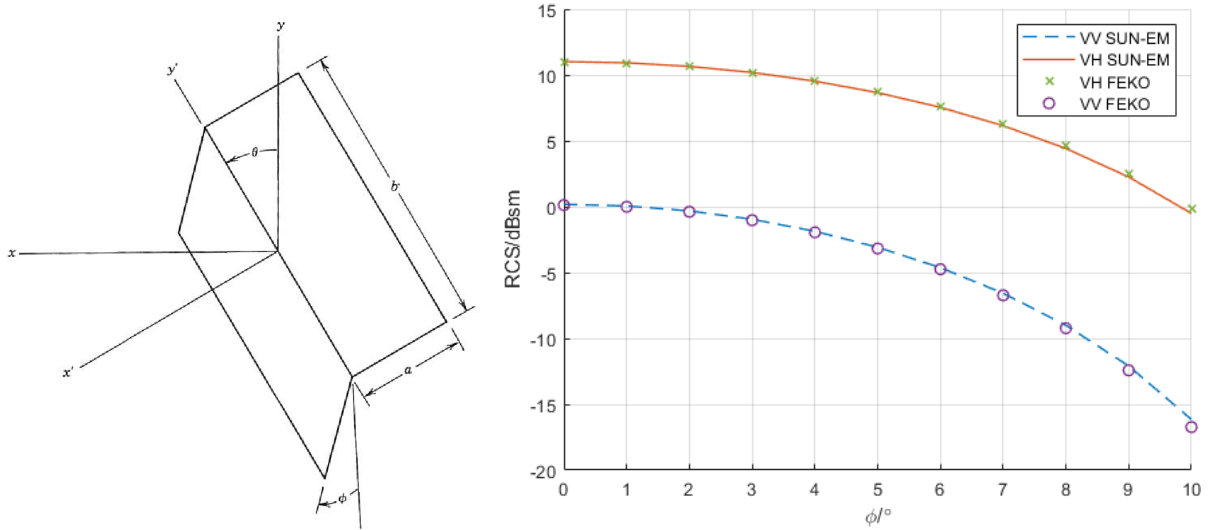
For a tilted dihedral corner reflector as shown in figure 6.4a, [24] shows that

$$A = \frac{4\sqrt{\pi}ab \sin\left(\frac{\pi}{4} + \phi\right)}{\lambda} \begin{bmatrix} -\cos 2\theta & \sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix}. \quad (6.13)$$

When  $\theta = 45^\circ$ , we see

$$A = \frac{4\sqrt{\pi}ab \sin\left(\frac{\pi}{4} + \phi\right)}{\lambda} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad (6.14)$$

meaning that a horizontally polarised incident wave will produce a vertically polarised scattered field and vice versa. This means that a dihedral corner reflector is a well suited target to validating the polarisation of the MRPO solution. For a small azimuth sweep from  $\phi = 0^\circ$  to  $\phi = 10^\circ$ , the monostatic RCS of a dihedral tilted  $45^\circ$  was calculated for both VV- and VH-polarisations. These results are shown in figure 6.4, with FEKO's MRPO solution as a reference.



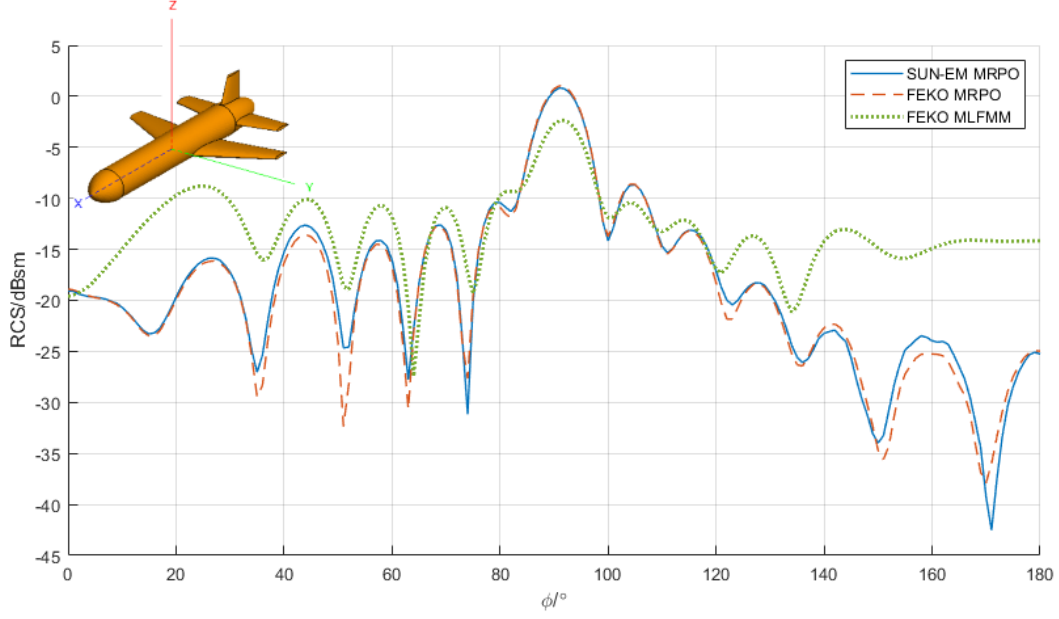
**(a)** Tilted dihedral corner reflector according to [24] **(b)** Monostatic RCS of a dihedral tilted  $45^\circ$  for VV- and VH-polarisations, with FEKO's solution as a reference.

**Figure 6.4:** Monostatic RCS of a dihedral tilted  $45^\circ$  testing the polarisation result.

It can be seen that the RCS for VH-polarisation is much greater than the RCS for VV-polarisation. While we expect the RCS for VV-polarisation to be 0, this is still an acceptable result, as the VV-polarised RCS is negligible compared to the VH-polarised RCS. The small error can be attributed to the current discretisation and shortcomings of the dipole approximation used to calculate the scattered magnetic field in the MRPO algorithm.

### 6.4.3. Application to non-idealised radar targets

The generic cruise missile (according to [25]) is an example of an electrically large, complex object that has been studied extensively [26][25][27]. Figure 6.5 shows the total monostatic RCS of the generic cruise missile over an azimuth sweep as a function of  $\phi$  at 1 GHz with an angular resolution of  $1^\circ$ . Both the FEKO and SUN-EM MRPO solvers were set to use 3 reflections.



**Figure 6.5:** Total monostatic RCS of a generic cruise missile at 1 GHz against azimuth angle  $\phi$ .  $\theta = 90^\circ$  for all incident angles.

It can be seen that the SUN-EM MRPO implementation strongly agrees with the FEKO MRPO implementation throughout. The MRPO solution, however does not agree well with MLFMM, particularly for angles less than  $38^\circ$  and greater than  $130^\circ$ . These differences are due to the shortcomings of the PO approximation, which does not model surface waves or edge diffraction. The solutions have all have an acceptable agreement between  $40^\circ$  and  $120^\circ$ , with only minor differences. The agreement between the MRPO and MLFMM solution show that MRPO can be used to provide insight for target detectability analysis.

#### 6.4.4. Performance

The calculation times (excluding the set-up time used for parsing data from the FEKO output files) and the memory usage of the SRPO and MRPO algorithms were recorded. Tables A.1 and 6.3 give the performance results for the trihedral and generic cruise missile respectively. The trihedral mesh consists of 1350 triangular patches, while the generic cruise missile mesh has 9788 triangular patches.

**Table 6.2:** Calculation time and memory usage for the trihedral.

Trihedral (1350 triangles)	SUN-EM SRPO	FEKO SRPO	SUN-EM MRPO	FEKO MRPO	FEKO MLFMM
Number of aspect angles	321	321	321	321	321
Total CPU time/s	16	12	574	530	572
<b>Average CPU time/s</b>	<b>0.05</b>	<b>0.04</b>	<b>1.79</b>	<b>1.65</b>	<b>1.78</b>
<b>Memory Usage/kB</b>	<b>556.23</b>	<b>722.93</b>	<b>3840</b>	<b>1268</b>	<b>189E3</b>



**Table 6.3:** Calculation time and memory usage for the generic cruise missile.

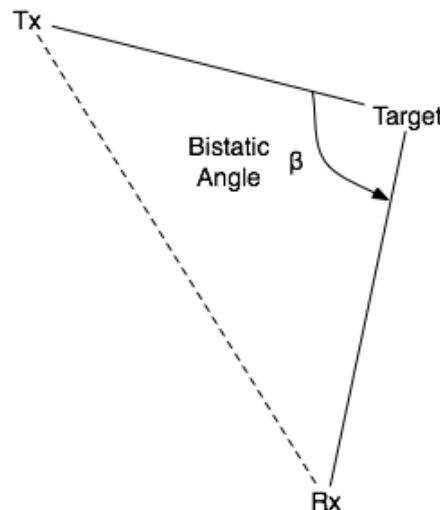
Generic cruise missile (9788 triangles)	SUN-EM SRPO	FEKO SRPO	SUN-EM MRPO	FEKO MRPO	FEKO MLFMM
Number of aspect angles	181	181	181	181	181
Total CPU time/s	21	46	4745	7635	1410
<b>Average CPU time/s</b>	<b>0.10</b>	<b>0.25</b>	<b>26.21</b>	<b>42.18</b>	<b>7.79</b>
<b>Memory Usage/MB</b>	<b>3.93</b>	<b>4.69</b>	<b>205.58</b>	<b>64.76</b>	<b>822.87</b>

A number of things can be seen from the performance results in these tables. Firstly, it can be seen that the SUN-EM and FEKO SRPO algorithms have a similar computational cost. Both SRPO algorithms arrive at a solution within a fraction of the time taken by either MRPO implementation or MLFMM, with a fraction of the memory usage. This shows the value of using the PO approximation. It is also seen that both MRPO implementations take significantly longer than the SRPO algorithms. This is primarily due to the time required to calculate the reflected field for each reflection, which require integration over the whole surface. It can be seen that neither MRPO algorithm offers significant time savings compared to MLFMM. The only advantage to using MRPO over MLFMM that can be seen here is the reduced memory requirements for MRPO. For the SUN-EM MRPO implementation, the high memory usage is a result of the large ( $N \times N$ ) visibility matrix that has to be stored in memory. This results in poor memory usage scaling with mesh size. Comparing the results from the two problems with different mesh sizes, it can be seen that the runtime for the SUN-EM implementation scales better than FEKO's implementation (despite the poorer memory usage scaling). The decreased runtime for the SUN-EM MRPO implementation is possibly due to the approximations used in the calculations of the reflected field, as well as the exclusion of all basis functions with no contribution to the field at a particular point.

# Chapter 7

## Bistatic Radar Cross Section Calculations

In the preceding sections, only monostatic RCS calculations have been considered, that is where the radar receiver and transmitter are located at the same point. While this is the most common setup, bistatic systems, where the receiver and transmitter are at different positions, have some advantages over monostatic systems. Figure 7.1 shows a bistatic radar configuration, showing the bistatic angle separating the receiver and transmitter.

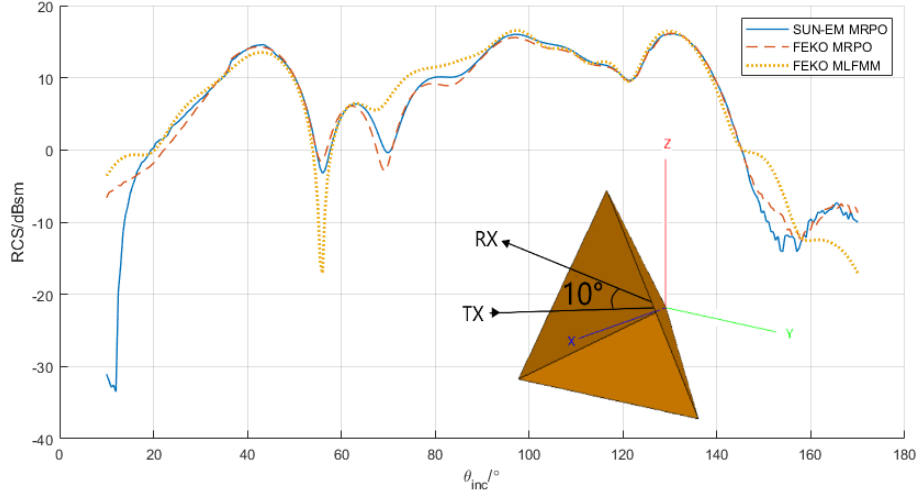


**Figure 7.1:** Bistatic radar configuration showing the bistatic angle  $\beta$ .

One such advantage of a bistatic setup is that stealth targets may have a larger bistatic RCS than monostatic RCS. This provides motivation to develop software that is capable of predicting bistatic RCS as well as monostatic RCS. Once the surface current on the target has been calculated, obtaining a solution for the bistatic RCS is relatively straightforward. The same shadowing algorithm used for determining which facets are illuminated by the transmitter is used to determine which facets are visible from the receiver. The scattered field in the receiver direction is then calculated using the surface current supported by the visible basis functions (ie. the current in all non-visible regions is set to zero according to the PO approximation).

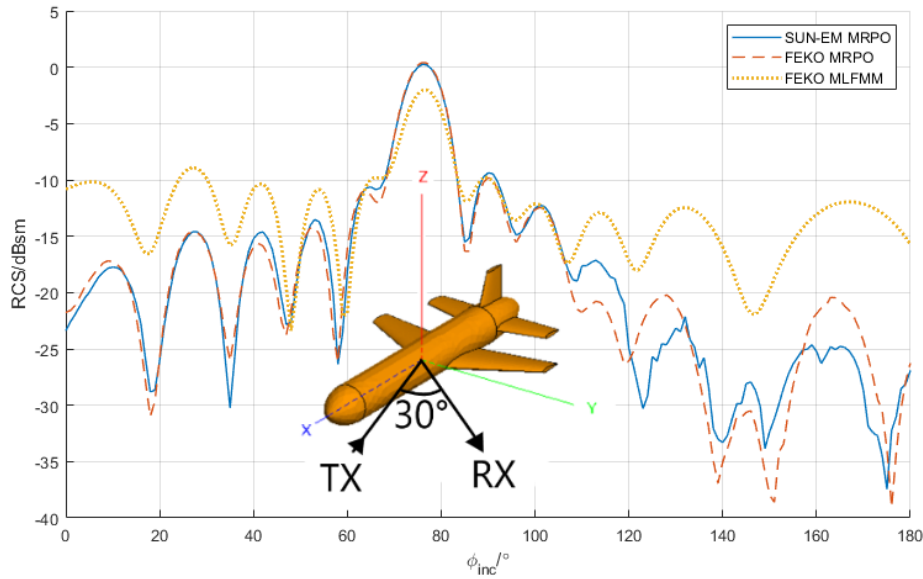
## 7.1. Results

The bistatic calculations were tested on the trihedral and generic cruise missile models used in Chapter 6. Figure 7.2 shows the RCS of a trihedral with an edge length of 1 m at 1 GHz plotted against incident angle ( $\theta_{inc}$ ), with a bistatic angle of  $10^\circ$  in the  $-\theta$  direction.



**Figure 7.2:** Total bistatic RCS of a trihedral with an edge length of 1 m at 1 GHz with the receiver separated from the transmitter by a bistatic angle of  $10^\circ$  in the  $-\theta$  direction.

Figure 7.3 shows the total bistatic RCS of a generic cruise missile at 1 GHz plotted against incident angle ( $\phi_{inc}$ ) with a bistatic angle of  $30^\circ$  in the  $\phi$  direction.



**Figure 7.3:** Total bistatic RCS of a generic cruise missile at 1 GHz with the receiver separated from the transmitter by a bistatic angle of  $30^\circ$  in the  $\phi$  direction.

In figure 7.3, it can be seen that the RCS peaks where the transmitter is at  $75^\circ$  (and

the receiver is at  $105^\circ$ ), rather than at  $90^\circ$  as seen in the monostatic RCS. This is what one would expect for the bistatic RCS, considering the effect of Snell's law (the bistatic transmitter and receiver are  $15^\circ$  either side of where the monostatic RCS peaks). From figures 7.2 and 7.3, it can be seen that the MRPO implementation can predict bistatic RCS for small bistatic angles with roughly the same accuracy with which it predicts monostatic RCS. The PO approximation is not appropriate for bistatic RCS calculations with large bistatic angles or forward scattering, because the PO approximation abruptly sets the surface current to zero in all non-illuminated regions. This results in the PO approximation significantly underestimating the scattering towards a receiver behind the target.

# Chapter 8

## Summary and Conclusion

In this report, the problem of predicting the RCS of electrically large arbitrary shaped PEC targets is discussed. In the context of this problem, the report looks at the design and implementation of an asymptotic CEM solver based on the PO approximation. A basic solver assuming full illumination of the target and no interaction between the targets surfaces is first designed and implemented. Following this, the solver is improved through two iterations that progressively improve the accuracy of the RCS predictions of complex targets obtained by the solver. The first improvement made is accounting for shadowing of the target, while the second is modelling interactions between surfaces of the target through multiple reflections.

The results obtained show that the solver performs with an accuracy very similar to that of the PO solver used in FEKO, a commercial CEM package. Moreover, the MRPO solver designed in this project is shown to solve problems with large meshes significantly faster than FEKO's MRPO solver (although an increased memory usage is also noticed). For simple targets, the solutions obtained also agree with solutions from FEKO's MLFMM solver (a full wave solver) as well as agreeing with analytical solutions for very simple objects, such as a disc. When a single reflection is modelled (SRPO), it is seen that the solver designed in this report is significantly faster than full wave solvers (FEKO's MLFMM solver). When used on appropriate targets (ie. targets with features that result in specular scattering being the dominant scattering mechanism), the SRPO implementation gives accurate results with minimal computational cost. This shows its usefulness as a RCS prediction tool that can quickly give radar engineers insight into the RCS characteristics of a particular target. In the case of this specific MRPO implementation, it is seen that the computational cost benefits of using the PO approximation do not offer a notable advantage over using a full wave solver.

### 8.1. Improvements and recommendations

#### 8.1.1. Accuracy based improvements

There are a number of ways in which the solver implemented in this project can be modified to improve the accuracy of the solution obtained. It is seen (particularly in figure 6.3) that

this method does not give accurate results when diffraction phenomena have a significant impact on the RCS of a target. Therefore, extending the given implementation with the physical theory of diffraction (PTD) may improve the accuracy of the solver for these sorts of targets.

The given implementation only uses first order basis functions to represent the surface current. The use of higher order basis function may increase the accuracy of the solution. Half RWG basis functions, as used in FEKO's PO solver, could be used to model the surface current and charges on non-shared edges. This would further increase the accuracy of the solution obtained.

In the interest of increasing computational efficiency, a dipole approximation of the surface current was used in the calculation of the reflected magnetic field in the MRPO algorithm. It was shown in section 6.3 that the approximation yields satisfactory results a few wavelengths from the source, but is inaccurate in the very near field. A hybrid technique, where the dipole model is used for field calculations at distances greater than  $\sim 5$  wavelengths and the magnetic field integral operator is used for very near field calculations below  $\sim 5$  wavelengths, could be used to yield more accurate results.

### 8.1.2. Performance based improvements

As noted, the MRPO solver discussed does not provide significant savings in runtime when compared to MoM with MLFMM. This means that as is, the MRPO solver does not have any obvious application. Before MRPO can be considered as an alternative over MLFMM, some sort of acceleration of the algorithm is required. The bottle neck in the MRPO implementation is the calculation of the reflected field. The field calculation has to be done for each of  $N$  basis functions by summing over all other  $(N - 1)$  basis functions. This results in an algorithm that scales as  $O(N^2)$ . [5] proposes a fast MRPO algorithm that makes use of the MLFMM to accelerate the calculation of the internally reflected field, effectively reducing the complexity of this calculation to  $O(N(\log(N)))$ . The MRPO implementation discussed in this report could benefit from this sort of acceleration.

# Bibliography

- [1] Davidson, D.: *Computational Electromagnetics for RF and Microwave Engineering*. Cambridge University Press, 2005. ISBN 9780521838597.
- [2] Asvestas, J.S.: The physical optics method in electromagnetic scattering. *Journal of Mathematical Physics*, vol. 21, no. 2, pp. 290–299, 1980.
- [3] Rao, S., Wilton, D. and Glisson, A.: Electromagnetic scattering by surfaces of arbitrary shape. *IEEE Transactions on Antennas and Propagation*, vol. 30, no. 3, pp. 409–418, 1982.
- [4] Balanis, C., Sevgi, L. and Ufimtsev, P.: Fifty years of high frequency diffraction. *International Journal of RF and Microwave Computer-Aided Engineering*, vol. 23, 07 2013.
- [5] Xiang, D.P.: Fast mesh-based physical optics for large-scale electromagnetic analysis. 2016. Available at: <http://hdl.handle.net/10019.1/100185>
- [6] Maxwell, J. and Torrance, T.: *A Dynamical Theory of the Electromagnetic Field*. Torrance collection. Wipf & Stock, 1996. ISBN 9781579100155.
- [7] Gibson, W.: *The Method of Moments in Electromagnetics*. CRC Press, 2014. ISBN 9781482235807.
- [8] Makarov, S.: *Antenna and EM Modeling in MATLAB*. A Wiley-Interscience publication. Wiley, 2002. ISBN 9780471218760.
- [9] Bingle, M., Burger, E., Jakobus, U. and van Tonder, J.J.: Theory and application of an MLFMM/FEM hybrid framework in FEKO. In: *2011 IEEE International Conference on Microwaves, Communications, Antennas and Electronic Systems (COMCAS 2011)*, pp. 1–3. 2011.
- [10] Ling, H., Chou, R.. and Lee, S.: Shooting and bouncing rays: calculating the rcs of an arbitrarily shaped cavity. *IEEE Transactions on Antennas and Propagation*, vol. 37, no. 2, pp. 194–205, 1989.
- [11] Times, T.N.Y.: *2 Rival Designers Led the Way To Stealthy Warplanes*. 1991. Available at: <https://www.nytimes.com/1991/05/14/science/2-rival-designers-led-the-way-to-stealthy-warplanes.html>
- [12] Jakobus, U., Schoeman, M., van Tonder, J.J., Ludick, D.J. and Burger, W.: Selection of New Features in the Electromagnetic Solution Kernel of FEKO Suite 6.0. In: *27th Annual Review of Progress in Applied Computational Electromagnetics (ACES 2011)*, pp. 308–313. 2011.

- [13] Simulation for Connectivity, Compatibility, and Radar — Altair FEKO. 2020.  
Available at: <https://www.altair.com/feko/>
- [14] MathWorks - MATLAB & Simulink: Physical Optics Solver.  
Available at: <https://www.mathworks.com/help/antenna/ug/physical-optics-solver.html>
- [15] Jakobus, U. and Landstorfer, F.M.: Improved po-mm hybrid formulation for scattering from three-dimensional perfectly conducting bodies of arbitrary shape. *IEEE Transactions on Antennas and Propagation*, vol. 43, no. 2, pp. 162–169, 1995.
- [16] Zha, F.-T., Gong, S.-X., Xu, Y.-X., Guan, Y. and Jiang, W.: Fast shadowing technique for electrically large targets using z-buffer. *Journal of electromagnetic waves and applications.*, vol. 23, no. 2-3, pp. 341–349, 2009. ISSN 0920-5071.
- [17] Xiang, D. and Botha, M.: Efficient shadowing determination at grazing incidence, for mesh-based physical optics scattering analysis. *Electronics Letters*, vol. 52, no. 22, pp. 1893–1894, 2016.
- [18] Larsson, T. and Akenine-Möller, T.: A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics*, vol. 30, no. 3, pp. 450–459, 2006.
- [19] Ludick, D.J. and CEMAGG: cemagg/sun-em. 2015.  
Available at: <https://github.com/cemagg/SUN-EM>
- [20] Pouliguen, P., Hemon, R., Burlier, C., Damians, J.F. and Saillard, J.: Analytical formulae for radar crosssection of flat plates in near field and normal incidence. In: *Progress In Electromagnetics Research B*, vol. 9, pp. 263–279. 2008.
- [21] Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A. and Stich, M.: Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, vol. 29, no. 4, July 2010. ISSN 0730-0301.  
Available at: <https://doi.org/10.1145/1778765.1778803>
- [22] Xiang, D. and Botha, M.: MLFMM-based, fast multiple-reflection physical optics for large-scale electromagnetic scattering analysis. *Journal of Computational Physics*, vol. 368, 05 2018.
- [23] Xiang, D.P. and Botha, M.M.: Fast multiple-reflection physical optics (FMRPO) through comprehensive MLFMM acceleration. In: *2017 IEEE International Symposium on Antennas and Propagation USNC/URSI National Radio Science Meeting*, pp. 1565–1566. 2017.
- [24] Mott, H.: *Polarization in Antennas and Radar*. Wiley, 1986. ISBN 9780471011675.
- [25] Youssef, N.N.: Radar cross section of complex targets. *Proceedings of the IEEE*, vol. 77, no. 5, pp. 722–734, 1989.



- [26] Weinmann, F.: Ray tracing with PO/PTD for RCS modeling of large complex objects. *IEEE Transactions on Antennas and Propagation*, vol. 54, no. 6, pp. 1797–1806, 2006.
- [27] Smit, J.C., Cilliers, J.E. and Burger, E.H.: Comparison of MLFMM, PO and SBR for RCS investigations in radar applications. In: *IET International Conference on Radar Systems (Radar 2012)*, pp. 1–5. 2012.

# Appendix A

## Project Planning Schedule

**Table A.1:** Week by week breakdown of the project planning schedule.

Week	Work planned
27 Jul-02 Aug	Research methods of calculating RCS, focusing on PO.
03-9 Aug	Familiarise myself with the existing SUN-EM package, as well as the interface with FEKO.
10-16 Aug	Implement a basic PO solver in MATLAB assuming full illumination of the target.
17-23 Aug	Test the PO solver and RCS calculations, including a comparison with FEKO.
24-30 Aug	Research methods of determining target illumination.
31 Aug-6 Sept	Implement a shadowing function in MATLAB to correctly determine illumination.
14-20 Sept	Test shadowing function and the PO solver with shadowing. Research MRPO.
21-27 Sept	Implement MATLAB code to determine visibility and object self-shadowing for multiple reflections.
28 Sept-4 Oct	Implement MRPO according to research.
5-11 Oct	Test the MRPO implementation and generate examples.
12-18 Oct	Implement bistatic RCS calculations and begin report writing.
19-25 Oct	Report writing.
26 Oct-1 Nov	Bistatic RCS implementation & Report writing.
2-8 Nov	Review and refine report before 9 Nov. hand in.

# Appendix B

## Outcomes Compliance

**Table B.1:** ECSA Exit Level Outcomes (ELO) compliance.

Exit Level Outcome	Motivation of outcome compliance	Related Sections
<b>ELO 1: Problem solving</b>	In this problem, the main problem faced is the challenge of using computational electromagnetics to predict the radar cross section of a target. A number of different methods were investigated and evaluated. Implementing the selected method posed a number of sub-problems, including identifying and implementing methods to determine the visibility of points on a target, and reasonable approximating the surface currents and radiated field from a radar target. Each of these problems required non-obvious, complex solutions that had to be that had to be developed over the course of this project and implemented in software.	Chapters 4, 5, 6 and 7.
<b>ELO 2: Application of scientific and engineering knowledge</b>	This project required the integration of knowledge over the fields of electromagnetics, radar, mathematics and software design. Knowledge of electromagnetic was required in modelling the interactions of a target with an electromagnetic field from a radar system. Mathematical models of these interactions had to be used. These models then had to be applied in software. Additionally, the project required the application of engineering based project planning and project management knowledge.	Chapters 4, 5, 6 and 7.
<b>ELO 3: Engineering design</b>	The project required the design of algorithms and software. The algorithms had to be designed to be computationally efficient. Engineering design decisions, such as which approximations are appropriate in a given context had to be made.	Chapters 3, 4, 5, 6 and 7.

<b>ELO 4: Investigations, experiments and data analysis</b>	The accuracy of the solver implemented in this project had to be established. This was done by generating results and comparing these to a number of reference solutions. These comparisons were done though both graphical means and through the calculation of the relative error in the solution. For each experiment, problems had to be set up with appropriate targets to investigate certain behaviour (eg. the use of a trihedral corner reflector to investigate multiple reflections).	Sections 4.4, 5.2, 6.4 and 7.1.
<b>ELO 5: Engineering methods, skills and tools, including information technology</b>	The completion of this report required the use of electromagnetic modelling skills and tools such as FEKO. The use of the MATLAB environment and NVIDIA OptiX API were necessary for the software design. Engineering methods for design and project management were also employed.	Chapters 4, 5, 6 and 7.
<b>ELO 6: Professional and technical communication</b>	This outcome is met through a long written technical report, as well as an oral presentation submitted as a video. Both of these methods communicate the activities and details of this project on a professional and technical level.	Entire Report & oral presentation.
<b>ELO 8: Individual work</b>	This project was completed on an individual basis. The only external input on the project were short weekly meetings with my supervisor for high level guidance and progress reports.	Entire Project.
<b>ELO 9: Independent learning ability</b>	The completion of this project required research and gaining an understanding of unfamiliar topics, such as computational electromagnetics methods.	Chapters 2, 4, 5 and 6.

# Appendix C

## *runPOsolver.m*

```
1 function [po] = runPOsolver(Const, Solver_setup, ~, ~, refIsol)
2
3 %runPOsolver
4 %   Usage:
5 %       [po] = runPOsolver(Const)
6 %
7 %   Input Arguments:
8 %       Const
9 %           A global struct, containing general data
10 %       Solver_setup
11 %           Solver specific struct, e.g. frequency range, basis function
12 %           details, geometry details, aspect angles
13 %       refIsol
14 %           The reference solution-vector data (e.g. PO solution of FEKO
15 %           or SUN-EM)
16 %
17 %   Output Arguments:
18 %       po
19 %           Structs containing PO solution and timing data
20 %
21 %   Description:
22 %       Calculates the PO solution based on the Z and Y data that was
23 %       read / parsed
24 %       from the FEKO *.out fiel
25 %
26 %       =====
27 %       Written by Cullen Stewart-Burger on 27 July 2020
28 %       Stellenbosch University
29 %       Email: 20751028@sun.ac.za
30
31 narginchk(5,5);
32 message_fc(Const, ' ');
33 message_fc(Const, '
34
35 -----
36
37 ');
```

```

33 message_fc(Const, sprintf('Running PO solver'));
34
35 % Initialise the return values
36 po = [];
37 po.name = 'po';
38 Npo = Solver_setup.num_mom_basis_functions; % Total number of basis
    functions for whole problem
39 po.numSols = 1; %numSols; % For now, set to 1. (TO-
    DO: Update)
40 numFreq = Solver_setup.frequencies.freq_num; % The number of frequency
    points to process
41 numRHSperFreq = po.numSols / numFreq; % The number of solutions
    per frequency point.
42
43 % Calculate the solution vector (observable BFs only)
44 po.Isol = complex(zeros(Npo,1));
45 %create an intermediate solution vector with all BFs positive and
    negative
46 Isol = complex(zeros(Npo,2));
47 % The timing calculations also need to take into account that there is a
48 % frequency loop
49 po.setupTime = zeros(1,numFreq);
50 % Zero also the total times (for all frequency iterations)
51 po.totsetupTime = 0.0;
52 po.totsolTime = 0.0;
53
54
55 %Test to see which basis functions are illuminated
56 ray = setRays(Solver_setup, Solver_setup.theta, Solver_setup.phi);
57 [tri_id, dist] = Raytracer.intersect_rays(ray);
58 [tp, ~, itp] = unique(Solver_setup.rwg_basis_functions_trianglePlus);
59 [tm, ~, itm] = unique(Solver_setup.rwg_basis_functions_triangleMinus);
60 [~, i_vis_pos, ~] = intersect(tp, tri_id);
61 [~, i_vis_neg, ~] = intersect(tm, tri_id);
62 tp(i_vis_pos) = -1;
63 vis_pos = (tp(itp)==-1);
64 tp(i_vis_neg) = -1;
65 vis_neg = (tp(itm)==-1);
66 visible = vis_pos & vis_neg;
67 visible = visible(1:Npo);
68 %debug: plot the visible basis functions
69 %scatter3(Solver_setup.rwg_basis_functions_shared_edge_centre(visible,1)
    ,Solver_setup.rwg_basis_functions_shared_edge_centre(visible,2),
    Solver_setup.rwg_basis_functions_shared_edge_centre(visible,3));
70 %axis('equal');
71
72 % Start the frequency loop now

```

```

73 Isol = complex(zeros(numFreq, Npo));
74 for freq=1:numFreq
75
76     % Start timing (PO setup)
77     tic
78
79     % Reset each solution per frequency point here
80     solStart = 1;
81     solEnd   = numRHSperFreq;
82
83     % End timing (calculating the impedance matrix)
84     po.setupTime(freq) = toc;
85
86     % Start timing
87     tic
88
89
90     rn = Solver_setup.rwg_basis_functions_shared_edge_centre;
91     shared_nodes = Solver_setup.rwg_basis_functions_shared_edge_nodes;
92     ln = Solver_setup.nodes_xyz(shared_nodes(:, 2), :) - Solver_setup.
nodes_xyz(shared_nodes(:, 1), :);
93     ln = ln./Solver_setup.rwg_basis_functions_length_m(1:Npo);
94     %We do not know if this direction for ln is correct according
95     %to our reference. This is checked and corrected if necessary
96     %below.
97
98     side = Solver_setup.nodes_xyz(Solver_setup.
rwg_basis_functions_trianglePlusFreeVertex, :) - Solver_setup.
nodes_xyz(shared_nodes(:, 1), :);
99     normTest = cross(side, ln, 2);
100    reverse = dot(normTest, Solver_setup.triangle_normal_vector(
Solver_setup.rwg_basis_functions_trianglePlus(1:Npo), :), 2);
101    ln = ln.*reverse./abs(reverse);
102
103
104    %The visibility term (delta) should be set to 0 if either triangle
is not
105    %visible or +-1 depending on the incident direction relative to the
106    %surface normal.
107    nDotRay = dot(Solver_setup.triangle_normal_vector(Solver_setup.
rwg_basis_functions_trianglePlus(1:Npo), :), ray.dir(Solver_setup.
rwg_basis_functions_trianglePlus(1:Npo), :), 2);
108    delta = -nDotRay./abs(nDotRay).*visible;
109    %delta = -nDotRay./abs(nDotRay); %uncomment this line to use the
full
110    %illumination assumption
111    BF_side = [delta > 0.017, delta < -0.017];

```

```

112     delta(isnan(delta)) = 0;
113
114     k = 2*pi*Solver_setup.frequencies.samples(freq)/Const.CO;
115     [kx, ky, kz] = sph2cart(Solver_setup.phi*Const.DEG2RAD, (90-
Solver_setup.theta)*Const.DEG2RAD, -k);
116     k_vec = repmat([kx, ky, kz], [Npo, 1]);
117     H = (1/Const.ETA_0)*exp(1j*(dot(k_vec, rn, 2)));    %Find impressed
H field (-r directed plane wave with E field theta-polarised)
118     a_phi = [-sind(Solver_setup.phi), cosd(Solver_setup.phi), 0];
119     a_theta = -[cosd(Solver_setup.theta)*cosd(Solver_setup.phi), cosd(
Solver_setup.theta)*sind(Solver_setup.phi), -sind(Solver_setup.theta)
];
120     H_vec = H*a_phi;
121     Isol = repmat(dot(2*delta.*H_vec, ln, 2), [1, 2]).*BF_side;
122     %Isol(freq, :) = dot(2*delta.*H_vec, ln, 2);
123
124     Isol_refl = complex(zeros(Npo, 2, Solver_setup.num_reflections));
125     Isol_refl(:, :, 1) = Isol;
126     for refl_num = 2:Solver_setup.num_reflections
127         H_vec_pos = zeros(Npo, 3);
128         H_vec_neg = zeros(Npo, 3);
129         for n = 1:Npo
130             Isol_pos = complex(zeros(Npo, 1));
131             Isol_neg = complex(zeros(Npo, 1));
132             Isol_pos((Solver_setup.Visibility_matrix(:, n) == 1)) =
Isol_refl((Solver_setup.Visibility_matrix(:, n) == 1), 1, refl_num-1)
;
133             Isol_pos((Solver_setup.Visibility_matrix(:, n) == 3)) =
Isol_refl((Solver_setup.Visibility_matrix(:, n) == 3), 2, refl_num-1)
;
134             Isol_neg((Solver_setup.Visibility_matrix(:, n) == 2)) =
Isol_refl((Solver_setup.Visibility_matrix(:, n) == 2), 1, refl_num-1)
;
135             Isol_neg((Solver_setup.Visibility_matrix(:, n) == 4)) =
Isol_refl((Solver_setup.Visibility_matrix(:, n) == 4), 2, refl_num-1)
;
136             H_vec_pos(n, :) = calculateHfieldAtPointRWGCart(Const, rn(n,
:), Solver_setup, Isol_pos);
137             H_vec_neg(n, :) = calculateHfieldAtPointRWGCart(Const, rn(n,
:), Solver_setup, Isol_neg);
138         end
139         Isol_refl(:, :, refl_num) = [dot(2*conj(H_vec_pos), ln, 2) dot
(-2*conj(H_vec_neg), ln, 2)];
140     end
141     Isol = sum(Isol_refl, 3);
142
143

```



```

144 % Memory usage remains constant between frequency iterations
145 %po.memUsage = byteSize(Solver_setup.Visibility_matrix);
146
147
148 end%for freq=1:numFreq
149
150 if(Solver_setup.is_bistatic)
151     %Test to see which basis functions are visible from the reciever
152     theta = Solver_setup.theta + Solver_setup.theta_bistatic;
153     phi = Solver_setup.phi + Solver_setup.phi_bistatic;
154     ray = setRays(Solver_setup, theta, phi);
155     [tri_id, dist] = Raytracer.intersect_rays(ray);
156     [tp,~,itp] = unique(Solver_setup.rwg_basis_functions_trianglePlus);
157     [tm,~,itm] = unique(Solver_setup.rwg_basis_functions_triangleMinus);
158     [~, i_vis_pos, ~] = intersect(tp, tri_id);
159     [~, i_vis_neg, ~] = intersect(tm, tri_id);
160     tp(i_vis_pos) = -1;
161     vis_pos = (tp(itp)==-1);
162     tp(i_vis_neg) = -1;
163     vis_neg = (tp(itm)==-1);
164     visible = vis_pos & vis_neg;
165     visible = visible(1:Npo);
166
167     %Test which side of each basis function is visible
168     nDotRay = dot(Solver_setup.triangle_normal_vector(Solver_setup.
    rwg_basis_functions_trianglePlus(1:Npo), :), ray.dir(Solver_setup.
    rwg_basis_functions_trianglePlus(1:Npo), :), 2);
169     delta = -nDotRay./abs(nDotRay).*visible;
170     BF_side = [delta > 0.017, delta < -0.017];
171 end
172
173
174 po.Isol = Isol(:, 1).*BF_side(:, 1);
175 po.Isol = po.Isol + Isol(:, 2).*BF_side(:, 2);
176 po.totsolTime = toc;
177
178 message_fc(Const, sprintf('Finished P0 solver in %f sec.', po.totsolTime)
    );

```

# Appendix D

## *CalcRCS.m*

```
1 function [RCS] = calcRCS(Const, Solver_setup, theta_grid, phi_grid,  
    xVectors)  
2  
3 %calcRCS  
4 %   Usage:  
5 %       [RCS] = calcRCS(Const, Solver_setup, theta_grid, phi_grid,  
    xVectors)  
6 %   Input Arguments:  
7 %       Const  
8 %           A global struct, containing general data  
9 %       Solver_setup  
10 %           Solver specific struct, e.g. frequency range, basis function  
    details, geometry details  
11 %       theta_grid  
12 %           List of theta incident angles  
13 %       phi_grid  
14 %           List of phi incident angles  
15 %       xVectors  
16 %           The reference solution-vector data (e.g. PO solution of FEKO  
    or SUN-EM)  
17 %  
18 %   Output Arguments:  
19 %       RCS  
20 %           Structs containing monostatic RCS solution and timing data  
21 %  
22 %   Description:  
23 %       Runs the PO solution based on the RGB basis functions parsed  
24 %       from FEKO for each incident angle and calculates the RCS  
25 %  
26 %   =====  
27 %   Written by Cullen Stewart-Burger on 27 July 2020  
28 %   Stellenbosch University  
29 %   Email: 20751028@sun.ac.za  
30  
31 narginchk(5,5);  
32 r = 100000;
```

```

33 Solver_setup.r_bounding = getBoundingRadius(Solver_setup);
34 % Calculate now the E-field value here internal
35 index = 0;
36 RCS = zeros(length(theta_grid)*length(phi_grid), 2);
37 Raytracer.setGeom(Solver_setup);
38 if(Solver_setup.num_reflections > 1)
39     Solver_setup.Visibility_matrix = selfShadow(Solver_setup);
40     memUsage = byteSize(Solver_setup.Visibility_matrix);
41 end
42 for theta_degrees = theta_grid
43     for phi_degrees = phi_grid
44
45         Solver_setup.theta = theta_degrees;
46         Solver_setup.phi = phi_degrees;
47
48         %
49         -----
50         % Run the EM solver
51         %
52         -----
53
54         index = index + 1;
55         [Solution] = runEMSolvers(Const, Solver_setup, 0, 0, xVectors);
56
57         %Change reviever position for bistatic case
58         if(Solver_setup.is_bistatic)
59             reciever_theta_degrees = theta_degrees + Solver_setup.
60             theta_bistatic;
61             reciever_phi_degrees = phi_degrees + Solver_setup.
62             phi_bistatic;
63         else
64             reciever_theta_degrees = theta_degrees;
65             reciever_phi_degrees = phi_degrees;
66         end
67
68         EfieldAtPointSpherical = calculateEfieldAtPointRWG(Const, r,
69         reciever_theta_degrees, reciever_phi_degrees, ...
70         Solver_setup, Solution.P0.Isol);
71
72         relError = calculateErrorNormPercentage(xVectors.Isol(1:
73         Solver_setup.num_metallic_edges,index), Solution.P0.Isol(:,1));
74         message_fc(Const,sprintf('Rel. error norm. compared to reference
75         sol. %f percent', relError));
76
77         % Calculate now the magnitude of the E-field vector.
78         Efield_magnitude = sqrt(abs(EfieldAtPointSpherical(1))^2 + ...

```

```
71         abs(EfieldAtPointSpherical(2))^2 + ...
72         abs(EfieldAtPointSpherical(3))^2);
73     E_theta = abs(EfieldAtPointSpherical(2));
74     E_phi = abs(EfieldAtPointSpherical(3));
75     RCS(index, 1) = 4*pi*(r.*(E_theta))^2;
76     RCS(index, 2) = 4*pi*(r.*(E_phi))^2;
77     % end
78     end%for
79 end%for
80
81 end
82
83 function r = getBoundingRadius(Solver_setup)
84 vertices = double(Solver_setup.nodes_xyz);
85 r = max(sqrt(sum(vertices.^2,2)));
86 end
```

# Appendix E

## *selfShadow.m*

```
1 function Visibility_matrix = selfShadow(Solver_setup)
2 % for closed surfaces:
3 % for each bf
4 %     for each centroid
5 %         set up ray from bf to centroid
6 %         cast all rays for given bf
7 %         check if ray intersects with corresp. triangle, tri is visible
8 %         dot ray with the triangle normal to check if the ray passed through
           the solid object (can this be done before ray casting?)
9 %         test if pos and neg triangles are visible
10 %         bf is visible if both triangles are visible
11 %
12 % for generalised surfaces, we need something more sophisticated
13 %     consider replicating all the geometry as "inner" and "outer"
           surfaces
14 %     in this case, we need to follow the process above (except RT
           which can be done once) with four seperate cases:
15 %         outside BF to outside centroids
16 %         outside BF to inside centroid
17 %         inside BF to outside centroid
18 %         inside BF to inside centroid
19
20 %set up visibility matrix
21 Visibility_matrix = zeros(Solver_setup.num_mom_basis_functions,
           Solver_setup.num_mom_basis_functions, 'uint8');
22
23 %Before running visibility tests, we need to construct an outward facing
24 %normal vector for each BF, as defined in Xiang, 2016.
25 rwg_normal_vec = zeros(Solver_setup.num_mom_basis_functions, 3);
26 for ind = 1:Solver_setup.num_mom_basis_functions
27 rwg_normal_vec(ind, :) = Solver_setup.triangle_normal_vector(
           Solver_setup.rwg_basis_functions_trianglePlus(ind), :)...
28     + Solver_setup.triangle_normal_vector(Solver_setup.
           rwg_basis_functions_triangleMinus(ind), :);
29 end
30 %note: We have not normalised the vector, but as we are only interested
```

```

    in
31 %direction this is not necessary.
32
33 %offset for moving rays off origin
34 offset = min(Solver_setup.rwg_basis_functions_length_m(Solver_setup.
    num_mom_basis_functions), [], 'all')/10;
35
36 for ind = 1:Solver_setup.num_mom_basis_functions
37     orig = Solver_setup.rwg_basis_functions_shared_edge_centre(ind, :);
38     %set up rays from BF to each centeroid
39     rays.origin = repmat(orig,[Solver_setup.num_metallic_triangles 1]);
40     rays.dir = Solver_setup.triangle_centre_point - rays.origin;
41
42     %fix rays intersecting with origin triangle by projecting the origin
43     %slightly in the direction of departure.
44     rays.origin = rays.origin + offset*(rays.dir./vecnorm(rays.dir, 2,
    2));
45
46     %Run RT to find line-of-sight visibility
47     [tri_id,~] = Raytracer.intersect_rays(rays);
48     [tp, ~, itp] = unique(Solver_setup.rwg_basis_functions_trianglePlus)
    ;
49     [tm, ~, itm] = unique(Solver_setup.rwg_basis_functions_triangleMinus
    );
50     [~, i_vis_pos, ~] = intersect(tp, tri_id);
51     [~, i_vis_neg, ~] = intersect(tm, tri_id);
52     tp(i_vis_pos) = -1;
53     vis_pos = (tp(itp)==-1);
54     tp(i_vis_neg) = -1;
55     vis_neg = (tp(itm)==-1);
56     visible = vis_pos & vis_neg;
57     visible = visible(1:Solver_setup.num_mom_basis_functions);
58     %we now have a list of all the basis functions with line-of-sight
59     %visibility from the source basis function centre point
60
61     %Next, we need to determine which sides of the BFs are facing each
    other
62     r_source_obs = Solver_setup.rwg_basis_functions_shared_edge_centre
    -...
63         repmat(orig,[Solver_setup.num_mom_basis_functions 1]);
64     source_norm = repmat(rwg_normal_vec(ind, :), [Solver_setup.
    num_mom_basis_functions 1]);
65     source_facing = dot(source_norm, r_source_obs, 2);
66     obs_facing = dot(rwg_normal_vec, r_source_obs, 2);
67
68     pp = (source_facing > 0.017) & (obs_facing < -0.017);
69     pn = (source_facing > 0.017) & (obs_facing > 0.017);

```

```
70 np = (source_facing < -0.017) & (obs_facing < -0.017);
71 nn = (source_facing < -0.017) & (obs_facing > 0.017);
72
73 Visibility_matrix(ind, pp) = visible(pp)*1;
74 Visibility_matrix(ind, pn) = visible(pn)*2;
75 Visibility_matrix(ind, np) = visible(np)*3;
76 Visibility_matrix(ind, nn) = visible(nn)*4;
77 end
78 end
```

# Appendix F

## *Raytracer.m*

```
1 classdef Raytracer
2     methods(Static)
3         function setGeom(Solver_setup)
4
5             %Change indexing to base 0 for use in C++
6             face_index = Solver_setup.triangle_vertices-1;
7
8             % Send geometry to raytracer
9             % Note: mexRaytracer creates and preserves a copy of the
10            vertex
11            %           and face index data, so we don't have to preserve
12            the
13            %           variables here.
14            vertex_coord = Solver_setup.nodes_xyz;
15            modified = datestr(now);
16            mexRaytracer('setmesh',vertex_coord,face_index, modified);
17
18        end
19        function [tri_id,dist] = intersect_rays(rays)
20            % Find ray intersections
21            mexRaytracer('castrays',[rays.origin, rays.dir]);
22            [tri_id,dist] = mexRaytracer('getresults');
23            %change indexing to base 1 for use in MATLAB
24            tri_id = tri_id+1;
25        end
26    end
27end
```



# Appendix G

## *mexRaytracer.cpp*

```
1 // *****
2 // Optix Raytracer Interface for Matlab
3 //
4 // mexRaytracer.cpp
5 //
6 // CD Stewart-Burger 16/01/2020
7 //
8 //
9 // *****
10
11 // Global defines
12 #define MAX_ARG_LEN 256
13 #define MAX_MSG_LEN 256
14
15 // Stop windows.h from defining min and max as required by optix_math.h
16 #define NOMINMAX
17
18 // Default includes
19 #include "mex.h"
20 #include "optix_world.h"
21 #include "optix_prime.h"
22 #include <optixu/optixu_math_namespace.h>
23 #include <string.h>
24
25 typedef optix::float3 float3;
26 typedef optix::int3 int3;
27
28 struct Ray
29 {
30     float3 origin;
31     float3 direction;
32 };
33
34 struct Hit
35 {
36     float t;
```

```

37     int triId;
38 };
39
40 // Platform specific includes
41 #ifdef WIN32
42 #include <process.h>
43 #else
44 #include <pthread.h>
45 #endif
46
47 // Global variables
48 RTPcontext g_context = NULL;
49 RTPresult g_code = RTP_SUCCESS;
50 RTPmodel g_model;
51 RTPbuffertype g_bufferType = RTP_BUFFER_TYPE_HOST;
52 RTPbufferdesc raysBD;
53 RTPbufferdesc hitsBD;
54 Hit *g_hits = NULL;
55 Ray *g_rays = NULL;
56 int g_numthreads = 0;
57 int3 *g_indexdata = NULL;
58 float3 *g_vertexdata = NULL;
59 unsigned int g_numrays = 0;
60 char g_time_str[MAX_ARG_LEN + 1];
61
62 bool bInitRequired = true;
63 bool bRunNonBlocking = false;
64 bool bResultsAvail = false;
65 bool bResultsRead = false;
66 bool bVerbose = false;
67
68 #ifdef WIN32
69 HANDLE g_thread;
70 #else
71 pthread_t g_thread;
72 #endif
73
74 #ifdef WIN32
75 void RunTraversal(void *)
76 #else
77 void *RunTraversal(void *)
78 #endif
79 {
80
81     RTPquery query;
82     g_code = RTP_ERROR_UNKNOWN;
83     if (rtpQueryCreate(g_model, RTP_QUERY_TYPE_CLOSEST, &query) !=

```

```

RTP_SUCCESS)
84 {
85     mexWarnMsgTxt("rtpQueryCreate failed\n");
86 }
87 else if (rtpQuerySetRays(query, raysBD) != RTP_SUCCESS)
88 {
89     mexWarnMsgTxt("rtpQuerySetRays failed\n");
90 }
91 else if (rtpQuerySetHits(query, hitsBD) != RTP_SUCCESS)
92 {
93     mexWarnMsgTxt("rtpQuerySetHits failed\n");
94 }
95 else
96 {
97     rtpModelFinish(g_model);
98     if (rtpQueryExecute(query, 0 /* hints */) != RTP_SUCCESS)
99     {
100         mexWarnMsgTxt("rtpQueryExecute failed\n");
101     }
102     else
103         g_code = RTP_SUCCESS;
104     rtpQueryDestroy(query);
105 }
106 }
107
108 void WaitForThreadToFinish(void)
109 {
110     if (bVerbose)
111         mexPrintf("Waiting for traversal thread to finish\n");
112
113     // Wait for traversal thread to finish
114 #ifdef WIN32
115     WaitForSingleObject(g_thread, INFINITE);
116 #else
117     pthread_join(g_thread, NULL);
118 #endif
119     bRunNonBlocking = false;
120     bResultsAvail = true;
121     bResultsRead = false;
122     if (g_code != RTP_SUCCESS)
123     {
124         bResultsAvail = false;
125         mexWarnMsgTxt("Non-blocking run of rtuTraversalTraverse failed\n
126 ");
127     }
128 }

```

```

129 void Initialise(void)
130 {
131     if (bVerbose)
132         mexPrintf("Initializing\n");
133     // Create traversal API & context
134     if (rtpContextCreate(RTP_CONTEXT_TYPE_CUDA, &g_context) !=
RTP_SUCCESS)
135     {
136         if (rtpContextCreate(RTP_CONTEXT_TYPE_CPU, &g_context) !=
RTP_SUCCESS)
137         {
138             mexErrMsgTxt("rtpContextCreate failed\n");
139         }
140         else
141         {
142             if (g_numthreads == 0)
143             {
144                 g_numthreads = 1;
145             }
146             if (rtpContextSetCpuThreads(g_context, g_numthreads) !=
RTP_SUCCESS)
147             {
148                 mexErrMsgTxt("rtpContextSetCpuThreads failed\n");
149             }
150             else
151                 mexPrintf("Using CPU (GPU context setup failed)\n");
152         }
153         g_bufferType = RTP_BUFFER_TYPE_HOST;
154         /*
155     }
156     else
157     {
158         if (bVerbose)
159             mexPrintf("Using GPU\n");
160         //g_bufferType = RTP_BUFFER_TYPE_CUDA_LINEAR;//change to CUDA
type
161     } */
162 }
163
164 static void CleanUp(void)
165 {
166     if (bVerbose)
167         mexPrintf("Cleaning up\n");
168
169     // Wait for previous run to finish
170     if (bRunNonBlocking)
171     {

```

```

172     WaitForThreadToFinish();
173 }
174
175 // Destroy traversal API & context
176 g_code = rtpContextDestroy(g_context);
177
178 // Make sure memory gets freed
179 if (g_vertexdata)
180 {
181     // mexPrintf("Freeing vertices\n");
182     mxFree(g_vertexdata);
183 }
184 if (g_indexdata)
185 {
186     // mexPrintf("Freeing indices\n");
187     mxFree(g_indexdata);
188 }
189 if (g_rays)
190 {
191     // mexPrintf("Freeing rays\n");
192     mxFree(g_rays);
193 }
194 if (g_hits)
195 {
196     //mexPrintf("Freeing hits\n");
197     mxFree(g_hits);
198 }
199
200 // Now its ok to throw error, if any
201 if (g_code != RTP_SUCCESS)
202 {
203     mexErrMsgTxt("rtpContextDestroy failed\n");
204 }
205 }
206
207 template <class T_src>
208 void transposeFloat3(float3 *p_dst, T_src *p_src,
209                     unsigned int R_src)
210 {
211     unsigned int r;
212     for (r = 0; r < R_src; r++)
213     {
214         p_dst[r].x = (float)p_src[r];
215         p_dst[r].y = (float)p_src[r + R_src];
216         p_dst[r].z = (float)p_src[r + 2 * R_src];
217     }
218 }

```

```

219
220 template <class T_src>
221 void transposeInt3(int3 *p_dst, T_src *p_src,
222                  unsigned int R_src)
223 {
224     unsigned int r;
225     for (r = 0; r < R_src; r++)
226     {
227         p_dst[r].x = (int)p_src[r];
228         p_dst[r].y = (int)p_src[r + R_src];
229         p_dst[r].z = (int)p_src[r + 2 * R_src];
230     }
231 }
232
233 template <class T_src>
234 void transposeRay(Ray *p_dst, T_src *p_src,
235                 unsigned int R_src)
236 {
237     unsigned int r;
238     for (r = 0; r < R_src; r++)
239     {
240         p_dst[r].origin.x = (float)p_src[r];
241         p_dst[r].origin.y = (float)p_src[r + R_src];
242         p_dst[r].origin.z = (float)p_src[r + 2 * R_src];
243         p_dst[r].direction.x = (float)p_src[r + 3 * R_src];
244         p_dst[r].direction.y = (float)p_src[r + 4 * R_src];
245         p_dst[r].direction.z = (float)p_src[r + 5 * R_src];
246     }
247 }
248
249 // Repack data from (x1 x2 x3 ... y1 y2 y3 ... z1 z2 z3 ...)
250 // to (x1 y1 z1 x2 y2 z2 ...) and convert types as required
251 template <class T_dst, class T_src>
252 void Transpose(T_dst *p_dst, T_src *p_src,
253               unsigned int R_src, unsigned int C_src)
254 {
255     unsigned int r, c;
256     for (c = 0; c < C_src; c++)
257     {
258         for (r = 0; r < R_src; r++)
259         {
260             *p_dst++ = (T_dst)p_src[(r * C_src) + c];
261         }
262     }
263 }
264
265 // Convert data type from a source to destination block

```

```

266 template <class T_dst, class T_src>
267 void Convert(T_dst *p_dst, T_src *p_src, unsigned int N_src)
268 {
269     unsigned int n;
270     for (n = 0; n < N_src; n++)
271     {
272         *p_dst++ = (T_dst)p_src[n];
273     }
274 }
275
276 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *
    prhs[])
277 {
278     char arg_str[MAX_ARG_LEN + 1];
279
280     // Check inputs
281     if (nrhs < 1)
282         mexErrMsgTxt("No arguments\n");
283     if (!mxIsChar(prhs[0]))
284         mexErrMsgTxt("First argument must be a command string\n");
285     mxGetString(prhs[0], arg_str, MAX_ARG_LEN);
286
287     // Wait for previous run to finish
288     if (bRunNonBlocking)
289     {
290         WaitForThreadToFinish();
291     }
292
293     // Handle CPU thread argument first (in case of CPU only init)
294     if (strcmp(arg_str, "cputhreads") == 0)
295     {
296         // Check number of CPU threads argument
297         if (nrhs < 2)
298             mexErrMsgTxt("Number of CPU threads not specified\n");
299         if (!mxIsNumeric(prhs[1]))
300         {
301             mexErrMsgTxt("Number of CPU threads must be an integer\n");
302         }
303         int numthreads = (int)(*mxGetPr(prhs[1]));
304         if (numthreads > 0)
305         {
306             g_numthreads = numthreads;
307         }
308         else
309         {
310             mexErrMsgTxt("Invalid number of CPU threads\n");
311         }

```

```

312
313     if (!bInitRequired)
314     {
315         // Set number of CPU threads to use when GPU is not
available
316         // and ray traversal reverts to CPU implementation
317         if (rtpContextSetCpuThreads(g_context, g_numthreads) !=
RTP_SUCCESS)
318         {
319             mexErrMsgTxt("rtpContextSetCpuThreads failed\n");
320         }
321     }
322 }
323
324 // Handle initialisation
325 if (bInitRequired)
326 {
327     mexAtExit(CleanUp);
328     Initialise();
329     bInitRequired = 0;
330 }
331
332 // Process arguments
333 if (strcmp(arg_str, "cputhreads") == 0)
334 {
335     // Detect argument as valid
336 }
337 else if (strcmp(arg_str, "setmesh") == 0)
338 {
339     // Check triangle mesh inputs
340     if (nrhs < 2)
341         mexErrMsgTxt("No vertex data specified\n");
342     if (!(mxIsSingle(prhs[1]) || mxIsDouble(prhs[1])))
343     {
344         mexErrMsgTxt("Vertex data must be a single or double matrix\
n");
345     }
346     if (mxGetNumberOfDimensions(prhs[1]) != 2)
347     {
348         mexErrMsgTxt("Vertex data must be a 2D matrix\n");
349     }
350     if ((mxGetM(prhs[1]) != 3) && (mxGetN(prhs[1]) != 3))
351     {
352         mexErrMsgTxt("Vertex data must be (x y z)*n_vertices\n");
353     }
354     unsigned int n_vertices = (unsigned int)mxGetNumberOfElements(
prhs[1]) / 3;

```



```

355     if (n_vertices == 0)
356     {
357         mexErrMsgTxt("Vertex data empty\n");
358     }
359
360     if (nrhs < 3)
361         mexErrMsgTxt("No index data specified\n");
362     if (!(mxIsUint32(prhs[2]) || mxIsDouble(prhs[2])))
363     {
364         mexErrMsgTxt("Index data must be a uint32 or double index
matrix\n");
365     }
366     if (mxGetNumberOfDimensions(prhs[2]) != 2)
367     {
368         mexErrMsgTxt("Index data must be a 2D matrix\n");
369     }
370     if ((mxGetM(prhs[2]) != 3) && (mxGetN(prhs[2]) != 3))
371     {
372         mexErrMsgTxt("Index data must be (i1 i2 i3)*n_indices\n");
373     }
374     unsigned int n_indices = (unsigned int)mxGetNumberOfElements(
prhs[2]) / 3;
375     if (n_indices == 0)
376     {
377         mexErrMsgTxt("Index data empty\n");
378     }
379
380     // Compare the incoming geometry 'last modified' to the existing
geometry
381     if (nrhs < 4)
382         mexErrMsgTxt("No Mesh time\n");
383     if (!mxIsChar(prhs[3]))
384         mexErrMsgTxt("Mesh time must be a string\n");
385     char time_str[MAX_ARG_LEN + 1];
386     mxGetString(prhs[3], time_str, MAX_ARG_LEN);
387     if (strcmp(g_time_str, time_str) == 0)
388     {
389         return;
390     }
391     strcpy(g_time_str, time_str);
392
393     // Copy vertex data
394     if (g_vertexdata)
395     {
396         mxFree(g_vertexdata);
397     }
398     g_vertexdata = (float3 *)mxMalloc(n_vertices * sizeof(float3));

```

```

399     if (g_vertexdata)
400     {
401         mexMakeMemoryPersistent(g_vertexdata);
402     }
403     else
404     {
405         mexErrMsgTxt("Unable to allocate memory for vertex data\n");
406     }
407     {
408         // Tranpose and/or convert vertex data as required
409         float3 *p_dst = g_vertexdata;
410         if (mxIsSingle(prhs[1]))
411         {
412             float *p_src = (float *)mxGetData(prhs[1]);
413             transposeFloat3<float>(p_dst, p_src, n_vertices);
414         }
415         else
416         {
417             double *p_src = mxGetPr(prhs[1]);
418             transposeFloat3<double>(p_dst, p_src, n_vertices);
419         }
420     }
421
422     // Copy index data
423     if (g_indexdata)
424     {
425         mxFree(g_indexdata);
426     }
427     g_indexdata = (int3 *)mxMalloc(n_indices * sizeof(int3));
428     if (g_indexdata)
429     {
430         mexMakeMemoryPersistent(g_indexdata);
431     }
432     else
433     {
434         mexErrMsgTxt("Unable to allocate memory for index data\n");
435     }
436     {
437         // Tranpose and/or convert index data as required
438         int3 *p_dst = g_indexdata;
439         if (mxIsUint32(prhs[2]))
440         {
441             unsigned int *p_src = (unsigned int *)mxGetData(prhs[2])
;
442             transposeInt3<unsigned int>(p_dst, p_src, n_indices);
443         }
444         else

```

```

445         {
446             // Convert and interleave data
447             double *p_src = mxGetPr(prhs[2]);
448             transposeInt3<double>(p_dst, p_src, n_indices);
449         }
450     }
451
452     RTPbufferdesc verticesBD;
453     if (rtpBufferDescCreate(g_context,
454 RTP_BUFFER_FORMAT_VERTEX_FLOAT3, RTP_BUFFER_TYPE_HOST, g_vertexdata,
455 &verticesBD) != RTP_SUCCESS)
456     {
457         mexErrMsgTxt("rtpBufferDescCreate failed for vertexdata\n");
458     }
459     else if (rtpBufferDescSetRange(verticesBD, 0, n_vertices) !=
460 RTP_SUCCESS)
461     {
462         mexErrMsgTxt("rtpBufferDescSetRange failed for vertexdata\n"
463 );
464     }
465
466     RTPbufferdesc indicesBD;
467     if (rtpBufferDescCreate(g_context,
468 RTP_BUFFER_FORMAT_INDICES_INT3, RTP_BUFFER_TYPE_HOST, g_indexdata, &
469 indicesBD) != RTP_SUCCESS)
470     {
471         mexErrMsgTxt("rtpBufferDescCreate failed\n");
472     }
473     else if (rtpBufferDescSetRange(indicesBD, 0, n_indices) !=
474 RTP_SUCCESS)
475     {
476         mexErrMsgTxt("rtpBufferDescSetRange failed for indexdata\n")
477 ;
478     }
479
480     // Set triangle mesh data
481     if (g_model)
482         rtpModelFinish(g_model);
483     if (rtpModelCreate(g_context, &g_model) != RTP_SUCCESS)
484     {
485         mexErrMsgTxt("rtpModelCreate failed\n");
486     }
487     if (rtpModelSetTriangles(g_model, indicesBD, verticesBD) !=
488 RTP_SUCCESS)
489     {
490         mexErrMsgTxt("rtpModelSetTriangles failed\n");
491     }
492

```

```

483     //clock_t start = clock();
484     //Model update is currently done in blocking mode. Asynchronous
mode is possible.
485     if (rtpModelUpdate(g_model, RTP_MODEL_HINT_ASYNC) != RTP_SUCCESS
)
486     {
487         mexErrMsgTxt("rtpModelUpdate failed\n");
488     }
489
490     //clock_t stop = clock();
491     //mexPrintf("%lf", (double)(stop-start));
492 }
493 else if (strcmp(arg_str, "castrays") == 0)
494 {
495     // Check ray data inputs
496     if (nrhs < 2)
497         mexErrMsgTxt("Ray data missing\n");
498     if (!(mxIsSingle(prhs[1]) || mxIsDouble(prhs[1])))
499     {
500         mexErrMsgTxt("Ray data must be a single or double matrix\n")
;
501     }
502     if (mxGetNumberOfDimensions(prhs[1]) != 2)
503     {
504         mexErrMsgTxt("Ray data must be a 2D matrix\n");
505     }
506     if ((mxGetM(prhs[1]) != 6) && (mxGetN(prhs[1]) != 6))
507     {
508         mexErrMsgTxt("Ray data must be (px py pz kx ky kz)*n_rays\n"
);
509     }
510     if (bResultsAvail && !bResultsRead)
511     {
512         mexWarnMsgTxt("Discarding previous results\n");
513     }
514     g_numrays = (unsigned int)mxGetNumberOfElements(prhs[1]) / 6;
515     bResultsAvail = false;
516     bResultsRead = false;
517     if(g_numrays == 0) return;
518
519     // Check blocking mode input
520     if (nrhs == 3)
521     {
522         char mode_str[MAX_ARG_LEN + 1];
523
524         if (!mxIsChar(prhs[2]))
525             mexErrMsgTxt("Blocking mode must be specified with a

```

```

string\n");
526     mxGetString(prhs[2], mode_str, MAX_ARG_LEN);
527     if (strcmp(mode_str, "noblock") == 0)
528     {
529         bRunNonBlocking = true;
530     }
531 }
532
533 // Copy ray data
534 {
535     // Map ray buffer
536     if (g_rays)
537     {
538         rtpBufferDescDestroy(raysBD);
539         mxFree(g_rays);
540     }
541     g_rays = (Ray *)mxMalloc(g_numrays * sizeof(Ray));
542     if (g_rays)
543     {
544         mexMakeMemoryPersistent(g_rays);
545     }
546     else
547     {
548         mexErrMsgTxt("Unable to allocate memory for rays\n");
549     }
550
551     // Tranpose and/or convert ray data as required
552     if (mxIsSingle(prhs[1]))
553     {
554         float *p_src = (float *)mxGetData(prhs[1]);
555         transposeRay<float>(g_rays, p_src, g_numrays);
556     }
557     else
558     {
559         double *p_src = mxGetPr(prhs[1]);
560
561         transposeRay<double>(g_rays, p_src, g_numrays);
562     }
563
564     if (rtpBufferDescCreate(g_context,
RTP_BUFFER_FORMAT_RAY_ORIGIN_DIRECTION, g_bufferType, g_rays, &raysBD
) != RTP_SUCCESS)
565     {
566         mexErrMsgTxt("rtpBufferDescCreate failed for rays\n");
567     }
568     else if (rtpBufferDescSetRange(raysBD, 0, g_numrays) !=
RTP_SUCCESS)

```

```

569         {
570             mexErrMsgTxt("rtpBufferDescSetRange failed for rays\n");
571         }
572     }
573
574     // Create buffer for returned hit descriptions
575     if (g_hits)
576     {
577         rtpBufferDescDestroy(hitsBD);
578         mxFree(g_hits);
579     }
580     g_hits = (Hit *)mxMalloc(g_numrays * sizeof(Hit));
581     if (g_hits)
582     {
583         mexMakeMemoryPersistent(g_hits);
584     }
585     else
586     {
587         mexErrMsgTxt("Unable to allocate memory for hits\n");
588     }
589     if (rtpBufferDescCreate(g_context, RTP_BUFFER_FORMAT_HIT_T_TRIID
, g_bufferType, g_hits, &hitsBD) != RTP_SUCCESS)
590     {
591         mexErrMsgTxt("rtpBufferDescCreate failed for hits\n");
592     }
593     else if (rtpBufferDescSetRange(hitsBD, 0, g_numrays) !=
RTP_SUCCESS)
594     {
595         mexErrMsgTxt("rtpBufferDescSetRange failed for vertexdata\n"
);
596     }
597
598     // Run traversal to cast rays
599     if (bRunNonBlocking)
600     {
601         // Run traversal in separate thread (non-block)
602 #ifdef WIN32
603         g_thread = (HANDLE)_beginthread(RunTraversal, 0, NULL);
604 #else
605         pthread_create(&g_thread, NULL, &RunTraversal, NULL);
606 #endif
607     }
608     else
609     {
610         // Run traversal now and wait (block)
611         RunTraversal(NULL);
612         bResultsAvail = true;

```

```

613         bResultsRead = false;
614         if (g_code != RTP_SUCCESS)
615         {
616             bResultsAvail = false;
617             mexErrMsgTxt("Traversal failed\n");
618         }
619     }
620 }
621 else if (strcmp(arg_str, "getresults") == 0)
622 {
623     // Check outputs
624     if (nlhs < 1)
625         mexErrMsgTxt("Primitive ID output missing\n");
626     if (nlhs < 2)
627         mexErrMsgTxt("Intersection distance output missing\n");
628
629     // Check results
630     if (!bResultsAvail){
631         mexWarnMsgTxt("No results available\n");
632     }
633     plhs[0] = mxCreateNumericMatrix(g_numrays, 1, mxINT32_CLASS,
mxREAL);
634     int *p_prim_id = (int *)mxGetData(plhs[0]);
635
636     plhs[1] = mxCreateNumericMatrix(g_numrays, 1, mxSINGLE_CLASS,
mxREAL);
637     float *p_dist = (float *)mxGetData(plhs[1]);
638
639     float m = 1.0f-pow(2, -23);
640     for (unsigned int n = 0; n < g_numrays; n++)
641     {
642         *p_prim_id++ = g_hits[n].triId;
643         *p_dist++ = g_hits[n].t*m;
644     }
645     bResultsRead = true;
646 }
647 else if (strcmp(arg_str, "getsurfnorm") == 0)
648 {
649     // Check output
650     if (nlhs < 1)
651         mexErrMsgTxt("Surface normal output missing\n");
652
653     // Check results
654     if (!bResultsAvail)
655         mexErrMsgTxt("No results available\n");
656
657     // Create output

```

```

658     plhs[0] = mxCreateNumericMatrix(g_numrays, 3, mxSINGLE_CLASS,
mxREAL);
659     float *p_surfnorm = (float *)mxGetData(plhs[0]);
660
661     float3 *norms = (float3 *)mxMalloc(g_numrays * sizeof(float3));
662     if (norms)
663     {
664         //mexMakeMemoryPersistent(norms);
665     }
666     else
667     {
668         mexErrMsgTxt("Unable to allocate memory for norms\n");
669     }
670
671     for (size_t i = 0; i < g_numrays; i++)
672     {
673         if (g_hits[i].t < 0.0f)
674         {
675             //no hit
676             norms[i].x = 0;
677             norms[i].y = 0;
678             norms[i].z = 0;
679         }
680         else
681         {
682             int3 tri = g_indexdata[g_hits[i].triId];
683             float3 v0 = g_vertexdata[tri.x];
684             float3 v1 = g_vertexdata[tri.y];
685             float3 v2 = g_vertexdata[tri.z];
686             float3 e0 = v1 - v0;
687             float3 e1 = v2 - v0;
688             norms[i] = normalize(cross(e0, e1));
689         }
690     }
691
692     // Repack data from (nx1 ny1 nz1 nx2 ny2 nz2 ...)
693     // to (nx1 nx2 nx3 ... ny1 ny2 ny3 ... nz1 nz2 nz3 ...)
694     for (unsigned int n = 0; n < (g_numrays); n++)
695     {
696         *p_surfnorm++ = norms[n].x;
697     }
698     for (unsigned int n = 0; n < (g_numrays); n++)
699     {
700         *p_surfnorm++ = norms[n].y;
701     }
702     for (unsigned int n = 0; n < (g_numrays); n++)
703     {

```



```
704         *p_surfnorm++ = norms[n].z;
705     }
706
707     mxFree(norms);
708 } /**/
709 else if (strcmp(arg_str, "verbose") == 0)
710 {
711     // Set verbose option
712     bVerbose = true;
713 }
714 else
715 {
716     char err_msg[MAX_MSG_LEN];
717     sprintf(err_msg, "Unknown argument '%s'\n", arg_str);
718     mexErrMsgTxt(err_msg);
719 }
720 }
721 // EOF
```