# On-The-Fly Mesh Generation for a High Performance Physical Optics Radar Backscattering Simulator

**Sebastian Hegler, Ronny Hahnel, and Dirk Plettemeier**

Communications Laboratory
Dresden University of Technology, Dresden, 01127, Germany
sebastian.hegler@tu-dresden.de, ronny.hahnel@tu-dresden.de, dirk.plettemeier@tu-dresden.de

***Abstract*** — In this paper, we present a radar backscattering simulator based on the method of physical optics (PO). Our simulation tool closely intertwines the tessellation of the simulation geometry with the physical optics method kernel, which enables on-the-fly refinement of input model data while still yielding high precision and computational performance. The algorithms for the physical optics method as well as the parallelization scheme will be presented. Also, performance comparisons will be shown and explained, both in regard to accuracy of the results and computation time.

***Index Terms*** — Remote sensing, vector and parallel computation.

## I. INTRODUCTION

Radar clutter is loosely defined as "the part of the received signal that is undesired." For ground penetrating radar systems (aboard satellites or other space craft, for example), this definition applies to the part of the signal that is backscattered from the surface of the sounded object. Separating surface clutter from the received signal is a tremendous aid for the correct interpretation of radar images.

In order to calculate the backscattered signal of known terrains, we developed a simulation tool based on the method of physical optics. The development of this tool was driven by the need to simulate huge objects, i.e., entire moons or planets, and the possibility to choose arbitrary radiation patterns for the sender and receiver. It is implemented in C++, using OpenMP[1] for parallelization.

*Submitted On: Oct. 21, 2011*
*Accepted On: Jan. 31, 2012*

The following section is dedicated to introducing the method of physical optics. In Section III, our simulation tool is presented. Section IV gives a description of the generation of on-the-fly meshes. In Section V, results of different simulation scenarios will be compared. The final section is dedicated to drawing conclusions and outlining further work.

## II. METHOD OF PHYSICAL OPTICS

### A. Method overview

The method of physical optics is a method to calculate the electromagnetic field backscattered from the surface of an object. As this method neglects electromagnetic coupling, its demands for computation time and main memory are modest compared to other methods, for example the method of moments. Assuming far field conditions further simplifies the equations; this assumption is justified for our use cases.

The formulae given in this section describe the method of physical optics for dielectric bodies, assuming far field conditions, a homogeneous dielectric permittivity $\varepsilon_r$ and permeability $\mu_r$.

### B. Surface current densities

Given a far field radiation pattern (as calculated by antenna simulation tools, for example, or as an analytical formulation where possible), the incident electric and magnetic far field for a given point in free space can be calculated. These are denoted by $\mathbf{E}_i$ and $\mathbf{H}_i$. For a dielectric surface with a known homogeneous dielectric permittivity $\varepsilon_r$ and permeability $\mu_r$, the equivalent magnetic and

electric surface current densities can be calculated as follows[2–4]:

$$\mathbf{J}_s^{PO} = 2\hat{\mathbf{n}} \times \left[ \left( \frac{1}{1+\zeta cos\vartheta_i} \mathbf{e}_{i\parallel} \otimes \mathbf{e}_{i\parallel} \right) \cdot \mathbf{H}_i \right] + 2\hat{\mathbf{n}} \times \left[ \left( \frac{1}{1+\frac{\zeta}{cos\vartheta_i}} \mathbf{e}_{i\perp} \otimes \mathbf{e}_{i\perp} \right) \cdot \mathbf{H}_i \right],$$

and

$$\mathbf{M}_s^{PO} = -2\hat{\mathbf{n}} \times \left[ \left( \frac{\frac{\zeta}{cos\vartheta_i}}{1+\frac{\zeta}{cos\vartheta_i}} \mathbf{e}_{i\parallel} \otimes \mathbf{e}_{i\parallel} \right) \cdot \mathbf{E}_i \right] - 2\hat{\mathbf{n}} \times \left[ \left( \frac{\zeta cos\vartheta_i}{1+\zeta cos\vartheta_i} \mathbf{e}_{i\perp} \otimes \mathbf{e}_{i\perp} \right) \cdot \mathbf{E}_i \right],$$

where

$$\zeta = \frac{Z_F}{Z_0} = \frac{\sqrt{\frac{\varepsilon_r}{\mu_r}}}{\sqrt{\frac{\varepsilon_0}{\mu_0}}},$$

$\hat{\mathbf{n}}$ is the surface unit normal, and $\vartheta_i$ is the angle of incidence. $\mathbf{e}_{i\perp}$ and $\mathbf{e}_{i\parallel}$ are the perpendicular and parallel base unit vectors of the plane of incidence, separating the incident field into perpendicular and parallel components. Thereby, for both $\mathbf{J}_s^{PO}$ and $\mathbf{M}_s^{PO}$ polarization is accounted for.

### C. Backscattered field

Using dyadic Green's function in vectorial form, the electric field based on the equivalent surface current densities can be calculated[5, 6]. Assuming far field conditions again, the equations can be simplified. The electric field caused by the equivalent magnetic surface current, denoted $\mathbf{E}_m$, can be written as follows:

$$\mathbf{E}_m(\mathbf{r}', \mathbf{M}_s^{PO}) = jk\hat{\boldsymbol{\beta}}_r \times \int_S \frac{e^{-jkr}}{4\pi r} \mathbf{M}_s^{PO}(\mathbf{r})\, ds,$$

whereas the magnetic field caused by the equivalent electric surface current densities, denoted $\mathbf{H}_e$, is calculated as follows:

$$\mathbf{H}_e(\mathbf{r}', \mathbf{J}_s^{PO}) = -jk\hat{\boldsymbol{\beta}}_r \times \int_S \frac{e^{-jkr}}{4\pi r} \mathbf{J}_s^{PO}(\mathbf{r})\, ds,$$

where $\mathbf{r}$ is the source point and $\mathbf{r}'$ is the target point. $r = ||\mathbf{r}' - \mathbf{r}||$ is the distance between these points, and $\hat{\beta}_r = r^{-1}(\mathbf{r}' - \mathbf{r})$ is the unit vector of the direction. The total field can then be written as:

$$\mathbf{E}_{total} = \mathbf{E}_m - Z_0\hat{\boldsymbol{\beta}}_r \times \mathbf{H}_e.$$

For each surface element, $\mathbf{E}_{total}$ is weighted with the correspondent antenna gain before numerical integration is performed. By calculating a frequency series and transforming it to time domain, the radar echo backscattered by the given surface is calculated.

## III. PHYSICAL OPTICS SIMULATOR

### A. Overview

In its current state, our physical optics simulator is capable of calculating the backscattered radar echo of a given input topography in monostatic as well as in bistatic mode. Antenna radiation patterns for a Hertzian dipole or a finite length dipole are provided as analytical formulations. Arbitrary antenna radiation patterns can be used, by importing far field $\vartheta$-$\varphi$ maps for $\mathbf{E}$ and $\mathbf{H}$. Post processing is performed to generate radargram images from the frequency domain simulation output using the Python programming (scripting) language[7].

### B. Software evolution

Starting as a set of MATLAB scripts, the software was first ported to Fortan90 in order to build a standalone binary application[8]. This allowed a much higher degree of parallelism, as the bottlenecks of MATLAB and its MEX interface, namely the limitation of parallel constructs to compiled MEX files and the inability to re-use memory, could be overcome this way.

For a final re-implementation, C++ was chosen as programming language, because of its advantages over the Fortran family of programming languages. These include, amongst others, a concise integration of the paradigm of object oriented programming (while refraining from making its use obligatory), stricter type-safety enforcement (while offering well-defined ways to circumvent it where necessary), support for generic programming (using C++'s keyword `template`), and complete configurability at a low level (by operator overloading), making the resulting code easier to understand, maintain and evolve.

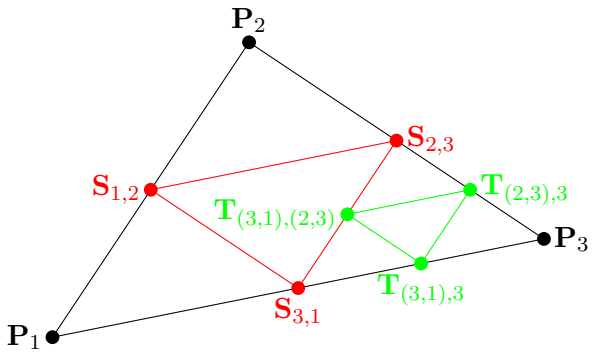


Fig. 1. Sierpinski tessellation of a triangle. Points $S$ are the edge bisection points of the original triangle. Points $T$, also edge bisection points, illustrate one further step in the refinement process.

Thanks to a change in the internal data structures and data flow layout, the allocation of memory for intermediate results could be removed, which was a limiting factor of the degree of parallelism. This resulted in a speedup of factor 2.5 (for serial execution): a test model with 235.623 triangles completes in 45s, compared to 120s for the Fortran90 version on an Intel Xeon X5365.

Parallel processing is implemented using OpenMP instructions. This is sufficient, as the problem resident set can be kept small enough to fit into the main memory of small compute nodes. The following section presents how this is achieved.

## IV. ON-THE-FLY MESH GENERATION

### A. Overview and requirements

A crucial factor for electrically large problem sets is the generation of a tessellation of the computational domain. Unfortunately, currently no tessellation software packages are available which feature a multi-threaded implementation and/or the ability to deal with huge datasets. For our typical use cases, the area to be tessellated is huge, ranging from 6100 km$^2$ in a basic case, to 1.4 million km$^2$ and beyond, usually consisting of some $10000^2$ points. Pre-calculating a tessellation for these areas would consume enormous amounts of time, and possibly exhaust main memory even on HPC platforms. For example, for the first given case, a tessellation occupies 12 GB of hard disk space, with 5 GB being the data relevant to the simulation, and the rest being adjacency information for the tessellation graph; however, this pre-generated mesh still has a resolution that is greater than the wavelength, and is therefore not fine enough to yield precise results.

The original terrain data is provided as a *Digital Elevation Model* (DEM), i.e., as a matrix of values denoting a difference in radius compared to a reference sphere or ellipsoid. Adjacency relationship is implicitly available as the point's neighborhoods. The only preprocessing stage that is necessary is to convert the implicit coordinates of the DEM to explicit Cartesian coordinates; the matrix structure, however, is retained, and therefore also the neighborhood relationship is. This data structure can then be used in the simulator to generate an on-the-fly tessellation of the topography of interest.

### B. Triangle selection strategies

For the on-the-fly meshing procedure, a triangle selection strategy has to be selected first. As the selection strategy plays is important to the performance of the code, it is implemented as a C++ template argument. The neighborhood relationship of the DEM allows four possible combinations

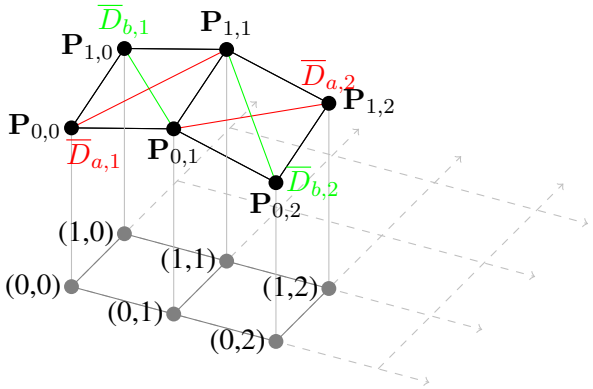


Fig. 2. Triangle construction from elevation map point set. Points $\mathbf{P}$ illustrate the heights at a given matrix coordinate. The diagonals $\overline{D}$ illustrate possible diagonals for the triangular tessellation. Four possible selection strategies can be applied: left-upper-right-lower (red), left-lower-right-upper (green), or choosing the diagonal by length (shorter or longer).

with two general cases, as illustrated in Figure 2. The first two cases are static: the same diagonal is chosen for all points, by either splitting the area into a triangle residing up left and one below right (illustrated by the red diagonals $\overline{D}_a$), or the other way round (illustrated by the green diagonals $\overline{D}_b$). The remaining cases are dynamic, by choosing the triangles based on the length of the diagonal.

After having chosen a selection strategy $\mathcal{S}$, a refinement criterion $\mathcal{C}$ needs to be defined. The default setting for $\mathcal{C} = \lambda_{min} = c_0/f_{max}$, i.e. the wavelength of the highest frequency of the sweep that is to be simulated.

### C. Parallel iteration

Given the input dataset, the selection strategy $\mathcal{S}$, and a refinement criterion $\mathcal{C}$, an iterator object, denoted $\mathcal{I}_{\mathcal{S}}(DEM, \mathcal{C})$, is generated.

Generating and using an iterator object is a natural choice when C++ is chosen as the implementation language. The class describing the iterator is implemented as a random access iterator, allowing consistent use within the idiomatics of C++. Since the iteration over the triangles provided by the input data set is the outermost loop, this is where the OpenMP instructions to parallelize the computation are placed.

This approach has several advantages. Firstly, it allows a single thread to work on adjacent memory items, allowing hardware-level pre-fetching to come into effect, thereby reducing memory latency. Using OpenMP's dynamic scheduling with a moderate chunk size, load imbalance caused by triangles that are discarded from the calculation can easily be accounted for. Experience shows that setting the chunk size to 5000 to 10000 works well, balancing scheduling overhead versus parallel computation time. Secondly, as an added benefit, this implementation is very easy to use as a (header-only) library function, since the user only has to provide an iterator class to feed the geometry to the simulator core. Parallelization and infrastructure is already taken care of.

### D. Mesh refinement

Using the algorithm to calculate the back-scattered echo for a triangle $T$ from Section II, and inspired by [9] a tessellation using Sierpinski triangles was implemented in order to increase precision where the original tessellation does not meet the necessary numerical criterion $\mathcal{C}$ for precise simulation results. The tessellation itself is implemented as a recursive function. The following piece of pseudo-code describes the function intertwined with the PO method. An illustration of the tessellation is shown in Figure 1.

Given a triangle $T : (\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3)$ generated by $\mathcal{I}_{\mathcal{S}}(DEM, \mathcal{C})$:

Call decomposition function $\mathcal{D}(\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3)$:

1) Are all edge lengths $(l_1, l_2, l_3)$ of $T < \mathcal{C}$?
   yes: Calculate back-scattered signal of $T$, return
2) Calculate edge bisection points $(\mathbf{S}_{1,2}, \mathbf{S}_{2,3}, \mathbf{S}_{3,1})$ from $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$
3) Calculate $\mathcal{D}(\mathbf{P}_1, \mathbf{S}_{1,2}, \mathbf{S}_{3,1})$
4) Calculate $\mathcal{D}(\mathbf{S}_{1,2}, \mathbf{P}_2, \mathbf{S}_{2,3})$
5) Calculate $\mathcal{D}(\mathbf{S}_{3,1}, \mathbf{S}_{2,3}, \mathbf{P}_3)$
6) Calculate $\mathcal{D}(\mathbf{S}_{1,2}, \mathbf{S}_{2,3}, \mathbf{S}_{3,1})$

### E. Evaluation

This implementation has several advantages. Firstly, generation and storage of a surface tessellation can be completely omitted, removing the primary computational bottleneck of the simulator tool chain. The resident memory set size of the simulator is reduced to only the pre-processed input elevation map plus the memory to store the results, which is another advantage. Therefore, thirdly, this allows for a much higher degree of parallelism, as the simulator now iterates over triangles which are generated on-the-fly, using only the neighboring points of the current point. Although this incurs a small penalty in total computation time, this allows simulation of very large radar targets.

## V. COMPARISON OF RESULTS

This section is dedicated to a comparison of the simulator in regard to computational performance and precision. To this end, two very different scenarios are used. For both cases, the input data set is provided as a DEM.

Test case A represents a very large area with generally smooth slopes, where the distance between points is large. The total number of triangles

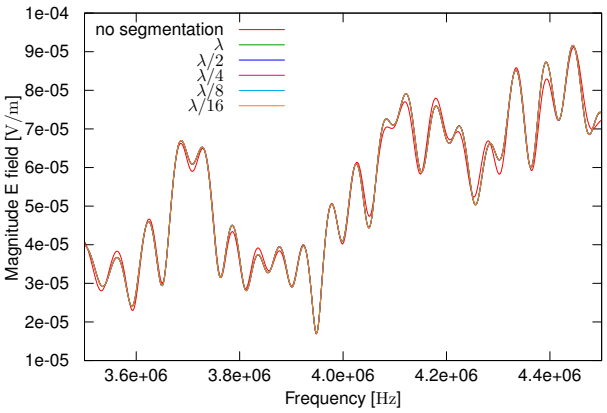


Fig. 3. Frequency domain signal for test case A, with varying segmentation.

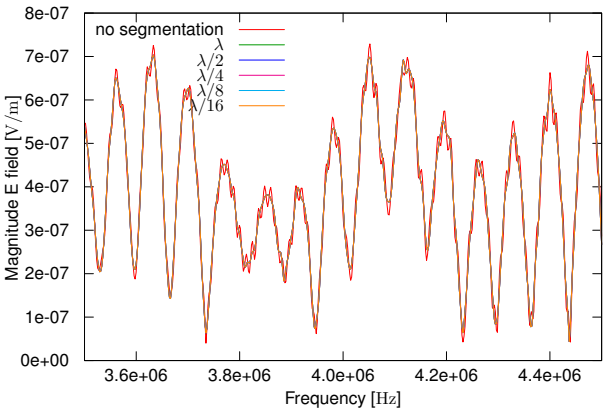


Fig. 5. Frequency domain signal for test case B, with varying segmentation.

in this dataset can be calculated from the number of points of the DEM: $12000^2$ points yield 287952002 initial triangles.

Test case B represents a closed body, which is smaller than the radar foot print of the simulated antenna system for the given simulation distance. The Cartesian distance between two points at the object's equator is smaller than the average distance for test case A, while it is in the range of a few meters at the poles. Given a DEM of 14000 by 7000 points, this results in 195958002 initial triangles.

All computation time measurements were done with the `time(1)` utility. The simulations were run on a 4x4 core Xeon E7430 system using the Linux operating system. The compiler used was gcc-4.5[10], compilation flags where chosen to yield best performance, even when code size would grow, also using the compiler's "unsafe" math optimizations (these optimizations may violate the IEEE 754[11] standard by changing rounding behavior and floating point unit error reporting; also see [12]).

### A. Precision of the simulation

As can be seen in Figures 3 and 5, for both test case A and test case B the curves of the simulations with segmentation disabled diverge notably. However, for segmentations using values smaller or equal to the wavelength, no increase in precision is gained.

When comparing the time domain signal for test case A shown in Figure 4, it can be seen that the segmentation plays only a little role in the simulation signal results, as the raw data already yields a good result.

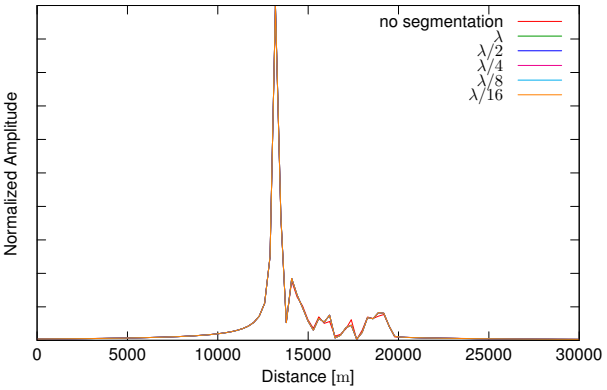


Fig. 4. Time domain signal for test case A, with varying segmentation.

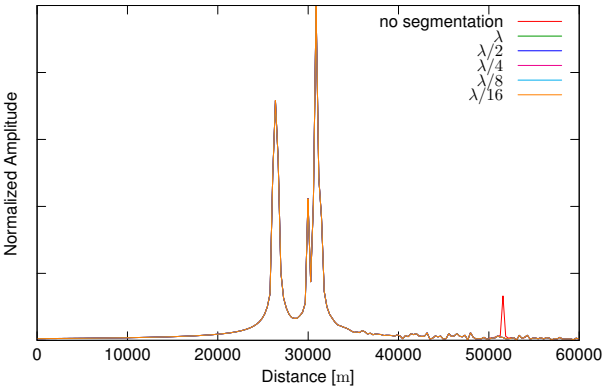


Fig. 6. Time domain signal for test case B, with varying segmentation.

Table 1: Simulation time for test case A, with varying segmentation

| Segmentation | $\lambda_{rel}$ | $t_{user}$ [s] | Increase of $t_{user}$ |
|---|---|---|---|
| none | n/a | 178.22 | n/a |
| $\lambda$ | 66.6 | 2389.40 | 13.407 |
| $\lambda/2$ | 33.3 | 9545.24 | 3.994 |
| $\lambda/4$ | 16.6 | 37964.96 | 3.977 |
| $\lambda/8$ | 8.3 | 151567.09 | 3.992 |
| $\lambda/16$ | 4.1 | 605593.99 | 3.995 |

Table 2: Simulation time for test case B, with varying segmentation

| Segmentation | $\lambda_{rel}$ | $t_{user}$ [s] | Increase of $t_{user}$ |
|---|---|---|---|
| none | n/a | 21215.08 | n/a |
| $\lambda$ | 66.6 | 21212.09 | 0.999 |
| $\lambda/2$ | 33.3 | 21222.50 | 1.000 |
| $\lambda/4$ | 16.6 | 21249.54 | 1.001 |
| $\lambda/8$ | 8.3 | 22705.27 | 1.068 |
| $\lambda/16$ | 4.1 | 75746.32 | 3.336 |

However, for test case B, the distortion introduced by the insufficiently fine discretization causes phantom echoes, as shown in Figure 6. This can be explained by the small size of the simulated object: for a closed surface which fits into the radar foot print and many triangles with edges that are bigger than the wavelength, side lobes are introduced with increasing angle of incidence. This can be seen in Figure 5. With mesh refinement enabled, the side lobes disappear, as do, in consequence, the phantom echoes.

### B. Computational performance

The Sierpinski refinement approach described in Section IV has a clear disadvantage: the increase of computation time by factor four for each level of refinement. This can clearly be seen in Table 1. The increase in computation time for test case A by a factor of $13.407$ between disabled segmentation and $\lambda$ refinement indicates that the original triangles are indeed very large; implying an average of $1.8725$ necessary refinement stages.

Test case B shows that for all refinement levels, except for $\lambda/16$, the simulations run in nearly the same computation time. For $\lambda/16$, an increase of factor $3.3$ was measured, indicating that only for this case, a further segmentation was necessary for a majority of the first level triangles. The minuscule decrease of computation time between the first two lines was probably caused by different loads on the compute host.

The differences between the initial and refined computation times for cases A and B can be explained by the simulated geometries. For case A, about 80% of the first level triangles are discarded, since they are not within the radar footprint and therefore do not contribute to the result. For case B however, about 50% of the first level triangles need to be considered for the simulation, resulting in a higher total computation time.

## VI. CONCLUSIONS AND FURTHER WORK

A simulation tool to calculate the backscattered radar echo of large dielectric surfaces was successfully implemented. Switching to C++ as the implementation language resulted in an immense gain of computational performance, and also in degrees of freedom in regard to programming and further development of the code base. The code base is now easier to maintain, and extensions are easier to implement. Core code features are available as template classes, which allow easy component reuse and compile-time configuration, without sacrificing computational performance. For example, the iterator class was adapted for a different geometry format. The only required changes were the loading routines for the different format, and the iterator itself, boiling down to 20 lines of code.

As a task for further investigation, integrating the method described in [13] as another simulation kernel is considered, as it may yield considerable savings in computation time.

## REFERENCES

[1] OpenMP.org – The OpenMP API specification for parallel programming. [Online]. Available: http://openmp.org/wp/

[2] C. Balanis, *Advanced Engineering Electromagnetics*. Wiley, 1989.

[3] E. Yamashita, *Analysis Methods for Electromagnetic Wave Problems*, ser. The Artech House antenna library. Artech House, no. 2, 1996.

[4] U. Jakobus, *Intelligente Kombination verschiedener numerischer Berech-*

*nungsverfahren zur effizienten Analyse elektromagnetischer Streuprobleme unter besonderer Berücksichtigung der Parallelverarbeitung*. Shaker, 1999.

[5] L. Felsen and N. Marcuvitz, *Radiation and Scattering of Waves*, ser. IEEE Press series on electromagnetic waves. IEEE Press, 1994.

[6] L. Díaz and T. Milligan, *Antenna Engineering using Physical Optics: Practical CAD Techniques and Software*, ser. Artech House antenna library. Artech House, no. 1, 1996.

[7] Python Programming Language – Official Website. [Online]. Available: http://www.python.org

[8] S. Hegler, R. Hahnel, and D. Plettemeier, "Implementation of a High Performance Numerical Simulator for Radar Surface Echoes," in *Proceedings of The 2nd International Multi-Conference on Engineering and Technological Innovation: IMETI 2009*. International Institute of Informatics and Systemics, 2009.

[9] M. Bader and C. Zenger, "Efficient Storage and Processing of Adaptive Triangular Grids Using Sierpinski Curves," in *Computational Science – ICCS 2006*, ser. Lecture Notes in Computer Science, V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, Eds. Springer Berlin / Heidelberg, vol. 3991, pp. 673–680, 2006.

[10] GCC, the GNU Compiler Collection. [Online]. Available: http://gcc.gnu.org/

[11] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*, Aug. 2008.

[12] GCC 4.5.3 Manual. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/

[13] J.-F. Nouvel, A. Herique, W. Kofman, and A. Safaeinili, "Radar Signal Simulation: Surface Modeling with the Facet Method," in *Radio Science*, vol. 39, 2004.