PSSFSS—An Open-source Code for Analysis of Polarization and Frequency Selective Surfaces

Peter S. Simon

Retired Antenna Engineer
Camarillo, California, USA
peter_simon@ieee.org

Abstract **–** The open-source code PSSFSS for analysis and design of polarization selective surfaces (PSSs), and frequency selective surfaces (FSSs) is presented, beginning with an introduction to the Julia programming language in which the code is written. Analysis methods and algorithms used in PSSFSS are described, highlighting features of Julia that make it attractive for developing this type of application. Usage examples illustrate the code's ease of use, speed, and accuracy.

Index Terms **–** computer code, frequency selective surface, FSS, Julia programming language, open-source, polarization selective surface, radome, reflectarray.

I. INTRODUCTION

PSSFSS [1] is a free, open-source code for analysis and design of polarization selective surfaces (PSSs), frequency selective surfaces (FSSs), radomes, reflectarray elements, and similar devices. It is written in Julia [2], a recently developed language for high-performance technical computing. Its speed, accuracy, and ease of use make PSSFSS useful and accessible to both students and working engineers. Section II of this paper briefly introduces the Julia programming language. Section III describes the PSSFSS program, including information on its deployment as a standard Julia package. It then goes on to outline the steps in the analytical formulation used in PSSFSS. Some of the key steps are detailed and their realization in Julia code is presented. In the process, a few of the features of Julia that make it well-suited for this type of application are highlighted. Section IV provides several usage examples, including timing and convergence studies, with comparisons to exact solutions or results of other simulation tools. Finally, conclusions are presented in Section V.

II. THE JULIA LANGUAGE

Julia [2] is a recently developed, free and open-source computer programming language intended for high-performance technical computing. A modern, dynamic, high-level language, Julia was specifically created to address perceived shortcomings of existing languages like R, C, C++, Fortran, Matlab, and Python. Notably, the developers of Julia were awarded the 2019 James H. Wilkinson Prize for Numerical Software. Although Julia can be used interactively where it "feels" like an interpreted language, it transparently compiles code to machine language prior to execution ("JIT" or just in time compilation), resulting in execution speed competitive with Fortran or C. Indeed, in 2017 Julia joined C, C++, and Fortran as the only languages to have achieved petaflop performance [3]. Support for parallelism (both distributed and threaded) is built into the language, as is support for multidimensional arrays of any dimensions and types. The syntax for performing linear algebra is similar to Matlab's. Installation of Julia is a simple process on Windows, Linux, or Macintosh computers. The language fully supports Unicode, so that symbols such as `ϵᵣ`, `μᵣ`, and `tanδ` can be used for Julia variable names, if desired.

Most of Julia's core code and its standard libraries are written in Julia itself, making it easy for users to read, understand, and contribute improvements. User-written packages employing custom types can be just as performant as functions and standard libraries supplied with the language, since all are written in Julia. In this way, Julia solves the so-called "two-language problem" wherein packages for dynamic languages like Python or Matlab that require maximum performance must either be partially written in C or Fortran, or be restricted to a compilable subset of the full language.

Additional comments about specific features of Julia that were found to be useful in developing PSSFSS are included below in Section III.

III. DESCRIPTION OF PSSFSS

A. Packaging

PSSFSS has been released as a registered Julia package. As such it can be installed via a single command at the Julia REPL[1]. Doing so causes Julia to download the

[1]"REPL" is an acronym for Read-Evaluate-Print Loop, the interactive terminal environment where the user types inputs that are then evaluated and printed back by Julia.

Submitted On: April 26, 2023
Accepted On: March 27, 2024

https://doi.org/10.13052/2024.ACES.J.390207

1054-4887 © ACES

Git repository [1] containing the PSSFSS source code, along with repositories of dependent packages. Similarly, updates published to the PSSFSS Github repository can be automatically retrieved by the user with another simple Julia update command.

Besides its on-line user manual [4], PSSFSS includes a detailed, 86 page theory document [5] that derives from first principles the formulas and algorithms implemented in the code. The code is heavily commented with references to specific sections and equation numbers in the theory document to promote code transparency.

B. Analysis methods and algorithms used

1. Overview

PSSFSS solves for the electric and/or magnetic surface currents on planar FSS/PSS sheets located within a stratified medium consisting of any number of dielectric layers. Currents are represented using modified Rao-Wilton-Glisson (RWG) basis functions [6] and are determined via a periodic moment method (PMM) solution of the mixed-potential integral equation (MPIE) in the space domain. The potential Green's functions are computed using a wide-band expansion [7] that, for normal incidence, permits the most expensive part of matrix assembly to be performed once only, regardless of the number of analysis frequencies performed. Multiple FSS/PSS sheets are accommodated by cascading their generalized scattering matrices (GSMs). The number of Floquet modes retained in the GSMs is calculated automatically such that excluded modes must encounter at least 30 dB of attenuation between neighboring sheets. The following subsections describe some of these analysis steps in more detail, and describe their Julia implementation.

2. Surface representation

As in other surface-based moment method (MoM) formulations, in PSSFSS a conducting surface (in the case of a capacitive FSS) or aperture (in the case of inductive FSS) must be triangulated as a preliminary step in forming the RWG basis functions. Triangulation is accomplished using the `Triangulate` package, a convenient Julia interface to the well-known, planar mesh generator `Triangle` [8]. Polygon and triangle vertex locations in the plane are stored in one-dimensional arrays, each element of which is of type `SVector{2,Float64}` as defined in the `StaticArrays` [9] package. Here the type parameters `2` and `Float64` mean that each planar vertex is represented by a length-2 "static vector" consisting of two 64-bit floating point values (the x and y coordinates of the point). Since the length of an `SVector` is encoded statically into its type, this length is known to the Julia compiler, allowing optimizations at compile time such as CPU register or stack allocation (rather than slower heap allocation), and efficient packing in memory and retrieval from arrays of these objects. For example, an array of n `SVectors` requires exactly the same amount of memory as is required for a contiguous array of $2n$ 64-bit floating point numbers. Furthermore, linear algebra operations on `StaticArray` objects of small orders are specialized via unrolling and other optimizations to be many times faster than those on an ordinary, heap-allocated arrays.

Besides speed of computation, static vectors (and matrices) inherit all the convenient array syntax of the language. By way of illustration, given a pair of `SVectors` `r` and `s`, the expression `3r - 2s` will perform the indicated scalar multiplications[2] of each vector followed by a vector difference. The result will be another `SVector`. With the `LinearAlgebra` standard library package, many additional operations are defined. E.g., we may compute the dot product of the two vectors by writing `r ⋅ s`. The centered dot is the infix form of the two-argument `dot` function and is typed at the Julia REPL (or within an appropriately configured editor) as `\cdot` (LaTeX notation) followed by a `Tab` character.

3. Green's function calculation

Discretization of the MPIEs via MoM in the spatial domain requires calculation of the potential Green's functions for currents flowing in a stratified medium, subject to quasi-periodic (Floquet) boundary conditions in the lateral directions. For efficient numerical evaluation, each potential Green's function is represented as a sum of modal series and quasi-static, spatial series, as derived in detail in [5]. As an example, consider an electric surface current flowing in the interface plane z_s, located between dielectric layers s and $s+1$ of the stratified media. The $\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}$ component of magnetic vector potential dyadic Green's function evaluated at $\boldsymbol{r} = \boldsymbol{\rho} + z_s\hat{\boldsymbol{z}} = (x, y, z_s)$ due to a point current source at $\boldsymbol{r}' = \boldsymbol{\rho}' + z_s\hat{\boldsymbol{z}} = (x', y', z_s)$ is

$$G^A_{xx}(\boldsymbol{\rho}-\boldsymbol{\rho}', z_s, z_s) = \tilde{\mu}\{\Sigma_{M1}(\boldsymbol{\rho}-\boldsymbol{\rho}') + + \frac{u}{4\pi}\left[\Sigma_{S1}(\boldsymbol{\rho}-\boldsymbol{\rho}') + c_3\Sigma_{S2}(\boldsymbol{\rho}-\boldsymbol{\rho}')\right]\}, \quad (1)$$

where $u > 0$ is an appropriately chosen smoothing factor with units of wavenumber, $\tilde{\mu} = \mu_s\mu_{s+1}/(\mu_s + \mu_{s+1})$, μ_n is the permeability of dielectric region n, ω is the radian frequency, and c_3 is a constant that depends on frequency, u, and the electrical parameters of regions s and $s+1$. Σ_{S1} and Σ_{S2} are quasi-static, spatial series, and Σ_{M1} is a modal series. All series are double sums $\sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty}$ which is abbreviated as $\sum_{m,n}$ hereafter.

[2]A numeric literal prepended to a variable or parenthesized expression implies multiplication.

Spatial series

The spatial series are

$$\Sigma_{S1}(\boldsymbol{\rho}-\boldsymbol{\rho}') = \sum_{m,n} \frac{e^{-u\rho_{mn}}}{u\rho_{mn}} e^{-j(m\psi_1+n\psi_2)}, \qquad (2)$$

$$\Sigma_{S2}(\boldsymbol{\rho}-\boldsymbol{\rho}') = \sum_{m,n} e^{-u\rho_{mn}} e^{-j(m\psi_1+n\psi_2)}, \qquad (3)$$

where $\rho_{mn} = \|\boldsymbol{\rho}-\boldsymbol{\rho}'-m\boldsymbol{s}_1-n\boldsymbol{s}_2\|$, $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are the direct lattice vectors of the periodic metalization pattern, and ψ_1 and ψ_2 are the incremental phase shifts associated with the Floquet boundary conditions in the $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ directions, respectively. It is evident that these rapidly (exponentially) convergent series are independent of frequency whenever the incremental phase shifts are. This occurs for normally incident illumination of the FSS ($\psi_1 = \psi_2 = 0$) or when simulating an FSS placed within a rectangular waveguide. When either of these is the case, and a swept-frequency analysis is desired, the contributions of the spatial series can be calculated a single time and reused at each new analysis frequency, achieving substantial execution time savings. It should also be noted that Σ_{S1} contains the known $1/\|\boldsymbol{\rho}-\boldsymbol{\rho}'\|$ static singularity in the $(m,n) = (0,0)$ term, permitting singularity subtraction and closed-form integration [10] as part of the MoM procedure.

It is of interest to see how these series are evaluated in Julia. The following code fragment is taken from the PSSFSS package:

```
# Sum over the r'th ring:
for (m, n) in Ring(r)
    uρₘₙ = norm(uρ⃗₀₀ - (m * us⃗₁ + n * us⃗₂))
    phase = -(m*ψ₁ + n*ψ₂)
    ...
end
```

The `for` loop iterates over the `(m,n)` tuples of integer indices comprising the `r`th concentric ring of the summation lattice. For example, `Ring(0)` generates the single tuple `(0,0)`, while `Ring(1)` iterates over the eight tuples `(1, -1)`, `(1, 0)`, `(1, 1)`, `(-1, -1)`, `(-1, 0)`, `(-1, 1)`, `(0, -1)`, and `(0, 1)`. In the code base `Ring` is a `struct` defined simply as

```
struct Ring
    r::Int
end
```

where the double colon is a type declaration for field `r`. The capability to iterate on `Ring(r)` as in the `for` loop above comes from defining a method for the base Julia `iterate` function that accepts two inputs: a variable of type `Ring`, and the current state of the iterator. It provides as outputs the next iterate and updated state. Each tuple is then efficiently generated iteratively in turn (i.e., lazily) in the `for` loop, avoiding unnecessary heap allocations. The above `for` loop is actually embedded in an outer loop over the ring index `r`, where convergence is checked after summing the contributions over each ring—the natural and rigorously correct way to form partial sums of such series. Thus the `Ring` type, straightforwardly implemented using Julia's standard iteration protocol, enhances the simplicity and readability of the Green's function code.

Calculation of the scalar variable `uρₘₙ` within the body of the `for` loop above shows how closely Julia code can resemble the mathematics that it is implementing. Variables `uρ⃗₀₀`, `us⃗₁`, and `us⃗₂` are all of type `SVector{2,Float64}`, representing in this case dimensionless 2-vectors. Again, such variable names incorporating Unicode symbols are easily typed using LaTeX notation. The expression passed as the argument to the `norm` function above will be evaluated prior to the call, resulting in a new `SVector`, stored either in the CPU stack or CPU registers (without any heap allocation). The function `norm` computes the 2-norm of a vector or matrix, and is exported by the `LinearAlgebra` package. However, a method specifically for arguments of type `SVector{2,Float64}` is provided at compile time by the `StaticArrays` package. Both the construction of the argument and calculation of its norm in the specialized method avoid (via unrolling) the loops that would otherwise be needed for dealing with standard arrays.

Modal Series

The modal series is

$$\Sigma_{M1}(\boldsymbol{\rho}-\boldsymbol{\rho}') = \sum_{m,n} f_{(m,n)} e^{-j\boldsymbol{\beta}_{mn}\cdot(\boldsymbol{\rho}-\boldsymbol{\rho}')}, \qquad (4)$$

where

$$f_{(m,n)} = \frac{1}{2A}\left[\frac{2V_{\mathrm{i}}^{\mathrm{TE}}(\boldsymbol{\beta}_{mn})}{j\omega\tilde{\mu}} - \frac{1}{\kappa_{mn}} - \frac{c_3}{\kappa_{mn}^3}\right]. \qquad (5)$$

Here A is the area of the unit cell, $V_{\mathrm{i}}^{\mathrm{TE}}$ is the spectral transmission line Green's function as defined in [11], $\boldsymbol{\beta}_{mn} = \boldsymbol{\beta}_{00} + m\boldsymbol{\beta}_1 + n\boldsymbol{\beta}_2$, $\boldsymbol{\beta}_{00} = (\psi_1\boldsymbol{\beta}_1 + \psi_2\boldsymbol{\beta}_2)/(2\pi)$, $\kappa_{mn} = \sqrt{\boldsymbol{\beta}_{mn}\cdot\boldsymbol{\beta}_{mn} + u^2}$, and $\boldsymbol{\beta}_1 = (2\pi/A)\boldsymbol{s}_2 \times \hat{\boldsymbol{z}}$ and $\boldsymbol{\beta}_2 = (2\pi)/A\hat{\boldsymbol{z}} \times \boldsymbol{s}_1$ are the reciprocal lattice vectors. The modal series is very smooth as a function of $\boldsymbol{\rho}-\boldsymbol{\rho}'$ and rapidly converging, since the summand decays as β_{mn}^{-5}. Using the change of variables $\boldsymbol{\rho}-\boldsymbol{\rho}' = \xi_1\boldsymbol{s}_1 + \xi_2\boldsymbol{s}_2$ and using the fact that

$$\xi_i = \frac{1}{2\pi}\boldsymbol{\beta}_i\cdot(\boldsymbol{\rho}-\boldsymbol{\rho}'), \quad i = 1,2 \qquad (6)$$

the series can be rewritten in the form

$$\Sigma_{M1}(\xi_1\boldsymbol{s}_1 + \xi_2\boldsymbol{s}_2) = e^{-j(\xi_1\psi_1+\xi_2\psi_2)} \sum_{m,n} f_{(m,n)} e^{-j2\pi(m\xi_1+n\xi_2)}. \qquad (7)$$

If we restrict evaluation to a discrete, regular grid of ξ_1 and ξ_2 points, and assume that the summand is negligibly small outside some maximal ring index, it is straightforward to recast the series above into the form of a two-dimensional discrete Fourier transform, which is efficiently evaluated using the fast Fourier transform (FFT).

Low-order bivariate polynomial interpolation is used to allow evaluation of the modal series at arbitrary points within the grid.

In the Julia implementation of the modal series, it is convenient when tabulating $f_{(m,n)}$ to use an array whose indices can range over both positive and negative integers. This is accomplished by use of the `OffsetArrays` package, which provides Julia with arrays that have arbitrary indices, similar to those found in some other programming languages like Fortran. Consider the following code fragment, taken from the modal series code:

```
parent = zeros(ComplexF64, 2mg + 1, 2mg + 1)
table1g = OffsetArray(parent, -mg:mg, -mg:mg)
```

The `zeros` call returns a two-dimensional array of 64-bit, complex, floating-point numbers initialized to zero. The indices range from `1` to `2*mg + 1` (native Julia arrays use 1-based indexing). The `OffsetArray` call returns an array sharing the same memory as `parent`, but whose indices range from `-mg` to `mg`. There is no performance penalty associated with indexing into `table1g` versus `parent`. Both `StaticArrays` and `OffsetArrays` were originally contributed by the Julia user community, and both are written entirely in Julia, showing the power and flexibility inherent in the language.

Tabulation of $f_{(m,n)}$ is performed in a parallel (multi-threaded) loop. Here is a fragment from the relevant code:

```
# Fill the tables:
@threads for r in (mmax_oldo2+1):mmaxo2
    ringsum1 = zero(eltype(table1g))
    ....
    for (m, n) in Ring(r)
        β⃗ₘₙ = β⃗₀₀ + m * β⃗₁ + n * β⃗₂
        β² = β⃗ₘₙ ⋅ β⃗ₘₙ # magnitude squared
        ...
    end
    ...
    table1g[m,n] += ringsum1
    ...
end
```

The outer loop over ring index is parallelized (threaded) by a call to `@threads` of the built-in threading library `Threads`. The `@` symbol denotes a Julia *macro*, which can arbitrarily transform Julia source prior to compilation. The two lines shown in the inner loop over summation ring indices, where $\boldsymbol{\beta}_{mn}$ and $\|\boldsymbol{\beta}_{mn}\|^2$ are computed, exemplify once again how the use of Unicode symbols in variable names can enhance clarity when translating a mathematical formulation to Julia code. `β⃗₀₀`, `β⃗₁`, and `β⃗₂` are all defined previously in the code and have type `SVector{2,Float64}`. The assignment causes `β⃗ₘₙ` to also have the same type without requiring any type declaration, an example of type inference performed by the Julia compiler.

Following tabulation, the FFT is performed in-place by a call to the function `FFT!` from the `FFTW` package. The latter provides convenient Julia bindings to the popular FFTW library [12].

4. GSM representation and cascading

A GSM is represented in the Julia code by the `GSM` type, defined as

```
struct GSM
    s11::Matrix{ComplexF64}
    s12::Matrix{ComplexF64}
    s21::Matrix{ComplexF64}
    s22::Matrix{ComplexF64}
end
```

The `s11`, `s12`, `s21`, and `s22` fields store the four partitions of the full GSM. In Julia, an array is indexed using square brackets, e.g. `s[1,2]` to select the element of array `s` in the first row and second column. The Julia compiler converts this syntax into the call `getindex(s, 1, 2)`. By extending the `getindex` function with a new method for the `GSM` type, one can enable indexing into a variable of this type:

```
function Base.getindex(gsm::GSM, i, j)
    (i, j) == (1, 1) && (return gsm.s11)
    (i, j) == (1, 2) && (return gsm.s12)
    (i, j) == (2, 1) && (return gsm.s21)
    (i, j) == (2, 2) && (return gsm.s22)
    throw(BoundsError(gsm, (i, j)))
end
```

Here the short-circuit logical "and" operator `&&` is used as a concise if-then construct, a common Julia idiom. Given `g::GSM`, the above definition allows, e.g., `g[1,2]` to be used in lieu of `g.s12`. Since Julia functions use "pass-by-sharing" semantics, such indexing does not cause any unwanted copying of the GSM partitions.

Cascading of GSMs is accomplished by use of the Redheffer star product [13–14]. If A and B are scattering matrices for two linear networks, the scattering matrix of the cascaded network is given by the (scattering) star product $A \star B$. If A and B are the block matrices

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \tag{8}$$

then their star product is

$$A \star B = \begin{bmatrix} (A \star B)_{11} & (A \star B)_{12} \\ (A \star B)_{21} & (A \star B)_{22} \end{bmatrix}, \tag{9}$$

where

$$(A \star B)_{11} = A_{11} + A_{12}B_{11}(I - A_{22}B_{11})^{-1}A_{21}, \tag{10a}$$

$$(A \star B)_{12} = A_{12}(I - B_{11}A_{22})^{-1}B_{12}, \tag{10b}$$

$$(A \star B)_{21} = B_{21}(I - A_{22}B_{11})^{-1}A_{21}, \tag{10c}$$

$$(A \star B)_{22} = B_{22} + B_{21}A_{22}(I - B_{11}A_{22})^{-1}B_{12}, \tag{10d}$$

and I is the identity matrix. The star product is implemented in PSSFSS in the `cascade` function:

```
function cascade(a::GSM, b::GSM)
  n2a = size(a[2,2], 2)
  n1b = size(b[1,1], 1)
  n1b ≠ n2a && error("Non-conformable GSMs")
```

```
    gprod1 = (I - a[2,2] * b[1,1]) \ a[2,1]
    s21 = b[2,1] * gprod1
    s11 = a[1,1] + (a[1,2] * b[1,1] * gprod1)
    gprod2 = (I - b[1,1] * a[2,2]) \ b[1,2]
    s12 = a[1,2] * gprod2
    s22 = b[2,2] + (b[2,1] * a[2,2] * gprod2)
    return GSM(s11, s12, s21, s22)
end
```

The backslash operator `\` and the symbol `I` used above are from the `LinearAlgebra` package. `I` is an object that represents an identity matrix of any order. The backslash, as in Matlab, performs matrix division using a polyalgorithm that depends on the array types on both sides. For the dense, square matrices occuring on the left-hand sides here, it calls a LAPACK solver based on LU factorization. By including the function definition

```
⋆(a,b) = cascade(a,b)
```

it is now possible to compute the cascade of `GSMs` `s` and `t` and store the result in `c` by simply typing `c = s ⋆ t`.

The GSM of a dielectric layer has the form $\left[\begin{smallmatrix} 0 & D \\ D & 0 \end{smallmatrix}\right]$, where D is a diagonal matrix. Therefore, when cascading some general GSM with that of a dielectric layer, the calculations needed are significantly simpler than the general case described by Eqs. (10). PSSFSS defines the `Layer` type for dielectric layers, along with several additional methods of the `cascade` function, with signatures `cascade(a::GSM, b::Layer)` and `cascade(a::Layer, b::GSM)`, wherein the appropriate, simplified calculations are encoded. This is an example of *multiple dispatch*, one of the fundamental paradigms of Julia programming. The correct method to be called for a function is selected from the available methods at compile time based on the types of *all* of the call site arguments. This is in contrast to standard object-oriented programming, where the selected method depends only on the first argument (single dispatch). Multiple dispatch is clearly more powerful, and it can be argued that it is a better fit to the needs of scientific computing. For example, given conformable arrays `A`, `B`, and `C`, the expression `A * B * C` is transformed by the Julia compiler into the call `*(A, B, C)`, a three-argument method for the `*` (multiplication) function. Within that method, the number of multiplications needed for `(A * B) * C` is compared to that for `A * (B * C)` to choose which of these to execute. Users can add their own methods as needed for their custom types.

5. Fast sweep algorithm

The default method for frequency sweeps is an extremely robust, diagonal rational function interpolation using the Stoer-Bulirsch [15] ("fast sweep") algorithm. Applied to the final, composite GSM of the structure being analyzed, it eliminates the need for a full PMM solution at each frequency (a "discrete sweep"), often producing speedups of 10× to 20×. The algorithm starts by analyzing the structure at five equally spaced frequencies (interpolation "knots") across the requested bandwidth. The composite GSM of the full structure is interpolated at the remaining frequencies via two different orders of rational function, and an error estimate is computed from their difference at each frequency. The next knot is selected as that frequency with the largest error estimate. A full analysis is performed at this knot, and new interpolations are performed incorporating the new data. The process continues until the error estimate remains less than -80 dB for three consecutive iterations. The strict termination criterion employed in PSSFSS for this algorithm makes it absolutely reliable, in the author's experience.

Additional details of the theory and implementation for PSSFSS can be found in [5].

C. Program features

Currently supported element types include rectangular strips, meanderlines, loaded and filled crosses, sinuous and manji crosses, Jerusalem crosses, split rings, and polyrings. The latter are able to model concentric rectangular loops or polygonal rings. All element types are fully parameterized for easy specification and optimization. The user manual [4] includes a gallery of supported element types.

Available output quantities include scattering parameters (magnitudes, phases, or complex) using a TE/TM, Ludwig 3, or LHCP/RHCP polarization basis, axial ratio, and others.

D. Program usage

PSSFSS is run in a Julia script. The geometry to be analyzed is specified as a vector of two or more dielectric `Layers` and zero or more `RWGSheets`. The latter define the FSS/PSS sheets and are instantiated by calling constructors such as `meander`, `strip`, `polyring`, etc. After also specifying the desired scan angles (or unit cell incremental phase shifts) and frequencies to be analyzed, a call to the `analyze` function performs the analysis. Outputs are requested using a tiny domain-specific language implemented by the `@outputs` macro. As an example, providing `@outputs s21db(L,R) ar22db(v)` as the second argument to the `extract_results` function will produce a matrix whose columns contain 1) transmission amplitude in dB to LHCP pol exiting port 2 due to a RHCP wave incident at port 1, and 2) reflected axial ratio in dB exiting port 2 due to an incident Ludwig 3 vertical wave at port 2. The outputs can be further post-processed and/or plotted in the same interactive or batch Julia session.

IV. EXAMPLE RESULTS

A. Symmetric strip grid

This example consists of a symmetric strip grating, i.e., a grating where the strip width is half the unit cell

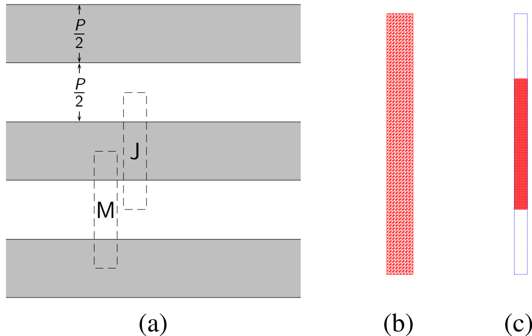


Fig. 1. Symmetric strip grid: (a) Gray areas represent metalization, Dashed lines show two possible choices for unit cell location, (b) Triangulation used for both J and M choice of unit cell, and (c) Triangulation with unit cell boundary also shown.

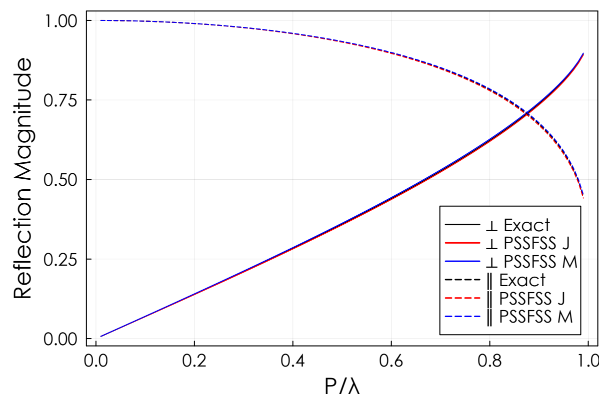


Fig. 2. Reflection magnitude for symmetric strip grid.

period P, as shown in part (a) of Fig. 1. The grating lies in the $z = 0$ plane with free space on both sides. The shaded areas represent metalization. The dashed lines show two possible choices for the unit cell location: "J" for a formulation in terms of electric surface currents, and "M" for magnetic surface currents. The same triangulation (parts (b) and (c) of the figure) can be used for either choice. The length of the unit cell in the x direction is arbitrary, here chosen to be one tenth of the strip width. The triangulation shown uses 8 edges in the x direction and 80 in the y direction for a total of 640 squares bisected into 1280 triangles. The PSSFSS analysis covered 99 frequencies where period normalized to wavelength (P/λ) varied from 0.01 to 0.99, requiring about 9 seconds of execution time. Normal incidence reflection coefficient magnitudes and phases computed by PSSFSS are compared with the exact solution ([16], [17, Prob. 10.6]) for the electric field parallel (∥) and perpendicular (⊥) to the direction of the grid (the y axis) in Figs. 2 and 3. The maximum magnitude error for either polarization is about

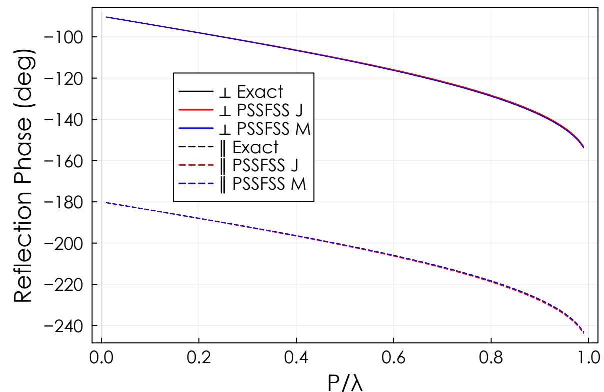


Fig. 3. Reflection phase for symmetric strip grid.

0.006 at the highest frequency. Maximum phase error is about 0.35°, also at the highest frequency. In all cases the solutions for the M and J unit cells bracket the exact solution. Presentation of the Julia script needed to run this example is omitted here; it can be found in the online manual [4].

B. Over-sized Jerusalem cross slot

This example, taken from [18] and illustrated in the inset of Fig. 4, constitutes a severe test of the fast sweep algorithm. The unit cell is a 2.5 cm square. The FSS is etched on a 0.02 cm thick dielectric slab with $\epsilon_r = 2.0 - j0.1$. The incidence angle is $(\theta, \phi) = (45°, 1°)$ so that higher-order free-space Floquet modes begin propagating at 7.025, 12.292, 12.542, 14.051, 16.665, and 17.060 GHz. Figure 4 shows how the computed transmission coefficient for TM polarization converges as the discretization is made finer. All of the traces overlay for frequencies below about 17 GHz. The fast sweep results for the final case are also shown, and they are indistinguishable from the discrete sweep results. The maximum fast sweep error for any of the 381 frequencies plotted is less than 0.0006 dB. The "glitches" in the traces between 16 GHz and 18 GHz are resonances associated with blazing frequencies. Swept frequency timing for discrete and fast sweeps are shown in Table 1. This analysis and all others reported in this paper were performed on a 3 GHz Core i7-9700 CPU. As shown in the table, fast sweeps for this example exhibited a speedup of about 4×. Greater speedups are usually obtained, but the many modes passing out of cut-off in this band make this case especially difficult. Here, about 90 of the 381 requested analysis frequencies required a full PMM solve in the fast sweep. The worst convergence of the model is obtained at the highest frequency, 20 GHz. The final column of Table 1 shows how the computed transmission coefficient at this frequency changes with number of triangles in the discretization. Applying Richardson extrapolation

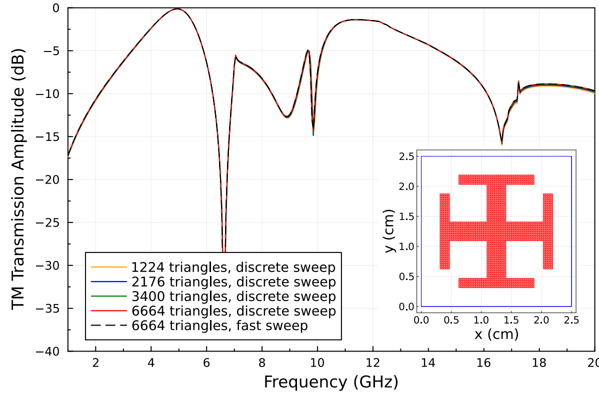


Fig. 4. TM transmission coefficient for the over-sized Jerusalem cross slot from [18]. The inset triangulation contains 3400 triangles; a blue dashed line demarks the 2.5 cm unit cell.

Table 1: Timing and convergence for the Jerusalem cross

Number of Triangles	Sweep Time (s) Discrete	Fast	Fast Sweep Speedup	T at 20 GHz (dB)
1224	199.3	58.3	3.4	-9.97
2176	627.5	155.2	4.0	-9.83
3400	1732.1	420.1	4.1	-9.75
6664	9182.0	2096.8	4.4	-9.66

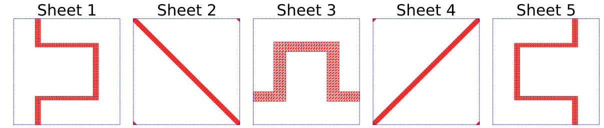
[19] to this series results in a predicted converged value of -9.54 dB, about 0.1 dB away from the 6664 triangle result. Listing 1 contains the code used to generate the fast sweep PSSFSS data and geometry plot for the 3400 triangle case.

C. 5-Sheet meanderline/strip CPSS

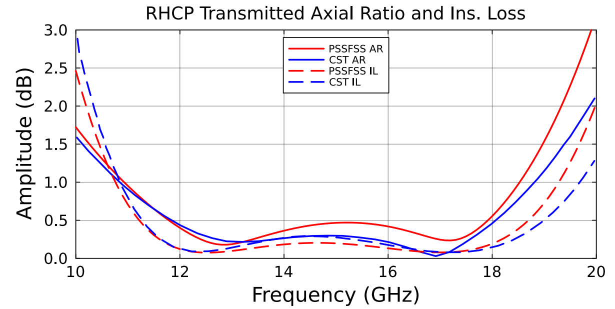
This example, from [20], consists of a 5-sheet, sequentially rotated, circular polarization selective surface (CPSS). The top images in Fig. 5 are the individual sheet triangulations. Each sheet shares the same unit cell, a 5.2 mm square. Sheets are etched on dielectric substrates separated from their neighbors by foam layers. The structure consists of 9 dielectric layers and 5 FSS sheets. Below the sheet triangulations is a plot of computed insertion loss and transmitted axial ratio (AR) for a right-hand circular polarization (RHCP) wave normally incident on the structure, both computed by PSSFSS and CST, the latter digitized from plots in [20]. The differences in the two models' results are attributed to finite metalization thickness used in the CST model, a feature not yet supported by PSSFSS. Analysis of the 5-sheet composite structure at 101 frequencies required only 20 s for PSSFSS, compared to the 17 min and 1 hr reported in [20] for CST using a coarse and fine mesh, respectively. The speed of PSSFSS for this case is due to multiple factors: 1) analysis is fastest at normal incidence, 2) meanderlines and strips are triangulated using structured meshes which are exploited by PSSFSS to avoid redundant calculations, 3) the elements used herein fill only a small portion of the unit cell and are not resonant, reducing the number of triangles required, and 4) the smoothly varying GSM over this analysis bandwidth required only 13 full solutions out of 101 analysis frequencies in the fast sweep algorithm.

Listing 1 Julia script used to produce the triangulation plot and compute the fast sweep performance shown in Fig. 4.

```
using PSSFSS, Plots
P = 2.5 # unit cell period
sheet = jerusalemcross(class='M', ntri=3400, P=P,
                       w=P/8, L1=3P/4, L2=P/8,
                       A=P/2, B=P/16, units=cm)
pl = plot(sheet, unitcell=true, linecolor=:red,
          size=(600,600))
strata = [Layer()
          sheet
          Layer(ϵᵣ=2.0, tanδ=0.05, width=0.02cm)
          Layer()]
freqs = 1:0.05:20; steering = (ϕ=1, θ=45)
reslt = analyze(strata, freqs, steering)
T_TM = extract_result(reslt, @outputs s21db(tm,tm))
```



(a)



(b)

Fig. 5. 5-Sheet CPSS from [20]: (a) All 5 sequentially rotated sheets share the same 5.2 mm unit cell and (b) RHCP → RHCP axial ratio and insertion loss computed by PSSFSS and CST Microwave Studio.

Listing 2 is a fragment of the log file created by PSSFSS when analyzing the CPSS. It recapitulates the layup of the structure and displays the number of Floquet modes assigned to each dielectric layer in the final

column. The layers labeled 1 and 11 are the semi-infinite vacuum regions on either side of the FSS structure. Note that the thin substrates adjacent to the FSS sheets, labeled as layers 2, 4, 6, 8, and 10, are assigned zero modes. This is because a very large number of modes would be required to cascade GSMs for very thin layers adjacent to FSS sheets. Instead, PSSFSS defines a so-called "GSM Block" consisting of the sheet and any neighboring thin layers. A single, composite GSM is computed for this GSM Block using the stratified media Green's functions. For the remaining foam spacer layers, it is observed that 10 modes are used in the 3.81 mm layers 3 and 9, while 18 modes are used in the thinner 2.61 mm layers 5 and 7.

Listing 2 Portion of the log file produced by PSSFSS when analyzing the CPSS of Fig. 5.

```
Layer Width units  epsr   tandel   mur  mtandel modes
----- ----------- ------ ------ ------ ------ -----
    1   0.000   mm   1.00 0.0000   1.00 0.0000     2
    2   0.127   mm   2.17 0.0009   1.00 0.0000     0
================= Sheet   1 =====================
    3   3.810   mm   1.04 0.0017   1.00 0.0000    10
    4   0.127   mm   2.17 0.0009   1.00 0.0000     0
================= Sheet   2 =====================
    5   2.610   mm   1.04 0.0017   1.00 0.0000    18
================= Sheet   3 =====================
    6   0.127   mm   2.17 0.0009   1.00 0.0000     0
    7   2.610   mm   1.04 0.0017   1.00 0.0000    18
    8   0.127   mm   2.17 0.0009   1.00 0.0000     0
================= Sheet   4 =====================
    9   3.810   mm   1.04 0.0017   1.00 0.0000    10
================= Sheet   5 =====================
   10   0.127   mm   2.17 0.0009   1.00 0.0000     0
   11   0.000   mm   1.00 0.0000   1.00 0.0000     2
```

Of course, this use of multi-mode GSM cascading is only possible because all sheets in this structure share the same periodicity. For structures with sheets of different periodicity, PSSFSS employs the usual approximation where only the principal TE and TM Floquet modes are included in the GSMs used for cascading.

D. Loaded cross slots bandpass filter

This example, originally from [21, Fig. 7.9], was also used in [22]. The geometry, shown in Fig. 6, consists of two identical loaded cross slot-type elements separated by a 6 mm layer of dielectric constant 1.9. Outboard of each sheet is a 1.1 cm layer of dielectric constant 1.3. The closely spaced sheets provide another good test of the generalized scattering matrix cascading formulation implemented in PSSFSS. Analysis of this structure with PSSFSS at 381 frequencies over a 20:1 bandwidth was performed for a few different mesh discretizations. A total of 18 Floquet modes in the center dielectric layer was selected by the code to accurately model interactions between the two closely spaced FSS sheets. Computed transmission amplitudes from PSSFSS and HFSS are compared in Fig. 7, where good agreement is observed over a very wide dynamic range of nearly 80 dB.

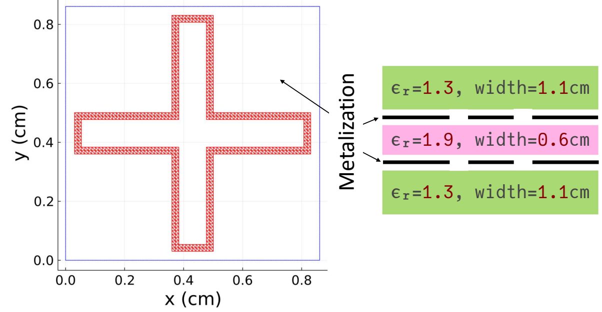


Fig. 6. Bandpass filter geometry consisting of two identical loaded cross slots in a three-layer, symmetrical, dielectric sandwich structure.

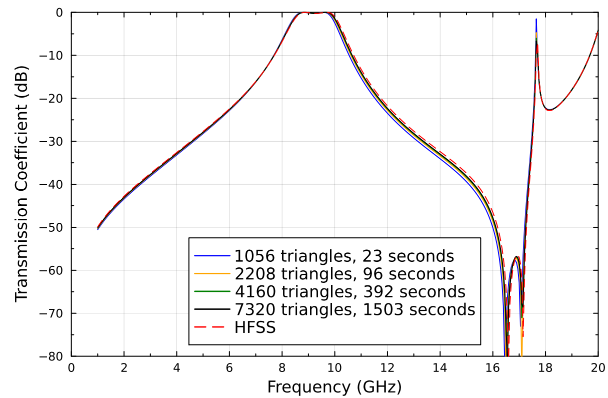


Fig. 7. Normal incidence transmission amplitude versus frequency for the bandpass filter.

Timings for the different discretizations are given in the figure legend.

E. Other

The complete scripts for all the above cases along with numerous other usage examples can be found in [4], including reflectarray elements, rabsorbers, and a fully worked-out design optimization for a CPSS using an evolutionary algorithm.

V. CONCLUSIONS

An open-source code PSSFSS for analysis of polarization and frequency selective surfaces has been described. PSSFSS can be obtained, installed, and used easily and without cost. It is written in the relatively new Julia programming language; therefore, some of the features of Julia that were especially convenient and useful in developing PSSFSS were highlighted. PSSFSS employs multi-threading and several novel algorithms to enhance computational efficiency. Examples were provided that demonstrate the speed, ease of use, and accuracy of the code.

REFERENCES

[1] P. S. Simon, "PSSFSS — Analysis of polarization and frequency selective surfaces in Julia," [Online]. Available: https://github.com/simonp0420/PSSFSS.jl

[2] *The Julia Programming Language*. [Online]. Available: https://julialang.org

[3] JuliaHub. *Parallel Supercomputing for Astronomy*. [Online]. Available: https://juliahub.com/case-studies/celeste

[4] P. S. Simon. *PSSFSS User Manual*. [Online]. Available: https://simonp0420.github.io/PSSFSS.jl/stable/manual

[5] P. S. Simon, "PSSFSS theory documentation," Tech. Report, May 2021. [Online]. Available: https://github.com/simonp0420/PSSFSS.jl/blob/main/docs/TheoryDocs/theorydoc.pdf

[6] P. S. Simon, "Modified RWG basis functions for analysis of periodic structures," *2002 IEEE MTT-S Int. Microwave Symp. Dig.*, vol. 3, pp. 2029-2032, Seattle, WA, June 2002.

[7] P. S. Simon, "Efficient Green's function formulation for analysis of frequency selective surfaces in stratified media," *Dig. 2001 IEEE AP-S Int. Symp.*, vol. 4, pp. 374-377, Boston, MA, July 2001.

[8] J. R. Shewchuk, "Triangle: Engineering a 2D quality mesh generator and delaunay triangulator," *Applied Computational Geometry: Towards Geometric Engineering*, ser. Lecture Notes in Computer Science, M. C. Lin and D. Manocha, Eds. Springer-Verlag, vol. 1148, pp. 203-222, May 1996.

[9] StaticArrays—statically sized arrays for Julia. [Online]. Available: https://github.com/JuliaArrays/StaticArrays.jl

[10] D. R. Wilton, S. M. Rao, A. W. Glisson, D. H. Schaubert, O. M. Al-Bundak, and C. M. Butler, "Potential integrals for uniform and linear source distributions on polygonal and polyhedral domains," *IEEE Trans. Antennas Propagat.*, vol. AP-32, no. 3, pp. 276-281, Mar. 1984.

[11] K. A. Michalski and J. R. Mosig, "Multilayered media Green's functions in integral equation formulations," *IEEE Trans. Antennas Propagat.*, vol. 45, no. 3, pp. 508-519, Mar. 1997.

[12] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216-231, 2005.

[13] R. Redheffer, "Inequalities for a matrix riccati equation," *Journal of Mathematics and Mechanics*, pp. 349-367, 1959.

[14] R. C. Rumpf, "Improved formulation of scattering matrices for semi-analytical methods that is consistent with convention," *Progress In Electromagnetics Research B*, vol. 35, pp. 241-261, 2011.

[15] Y. Ding, K.-L. Wu, and D. G. Fang, "A broadband adaptive-frequency-sampling approach for microwave-circuit em simulation exploiting Stoer-Bulirsch algorithm," *IEEE Trans. Microwave Theory Tech.*, vol. 51, no. 3, pp. 928-934, 2003.

[16] L. A. Weinstein, *The Theory of Diffraction and the Factorization Method: Generalized Wiener-Hopf Technique*. Golem Press, 1969.

[17] R. E. Collin, *Field Theory of Guided Waves*, 2nd ed. IEEE Press, 1991.

[18] X. Wang and D. H. Werner, "Improved model-based parameter estimation approach for accelerated periodic method of moments solutions with application to the analysis of convoluted frequency selected surfaces and metamaterials," *IEEE Trans. Antennas Propag.*, vol. 58, no. 1, pp. 122-131, 2010.

[19] G. Dahlquist and Å. Björck, *Numerical Methods in Scientific Computing, Volume I*. SIAM, 2008.

[20] A. Ericsson and D. Sjöberg, "Design and analysis of a multilayer meander line circular polarization selective structure," *IEEE Trans. Antennas Propag.*, vol. 65, no. 8, pp. 4089-4101, 2017.

[21] B. A. Munk, *Frequency Selective Surfaces: Theory and Design*. John Wiley & Sons, Inc., 2000.

[22] L. Li, D. H. Werner, J. A. Bossard, and T. S. Mayer, "A model-based parameter estimation technique for wide-band interpolation of periodic moment method impedance matrices with application to genetic algorithm optimization of frequency selective surfaces," *IEEE Trans. Antennas Propag.*, vol. 54, no. 3, pp. 908-924, 2006.

Peter S. Simon is a retired antenna engineer and a Life Senior Member of the IEEE. He received the B.S. and M.S. degrees in electrical engineering from the University of Illinois at Urbana/Champaign and the Ph.D. degree in electrical and computer engineering from the University of California at Santa Barbara. He served in the United States Navy as both an enlisted man and later as a commissioned officer. During his engineering career he worked at Raytheon Electromagnetic Systems Division (now Raytheon Intelligence & Space), Goleta, CA, in positions involving antenna design and computational electromagnetics, where he ultimately attained the rank of Principal Engineer. Following his stint at Raytheon, he joined Space Systems/Loral (now Maxar Technologies Space Systems), Palo Alto, CA, where his work again involved computational electromagnetics, analysis, and design applied to reflector antennas, feed horns, fre-

quency and polarization selective surfaces, and feed networks for commercial satellite applications. In 2014 he was selected among the first group of SSL employees to be awarded the title of Distinguished Engineer. He is the author or coauthor of five patents and several papers relating to antennas. Since his retirement he has been involved in developing open-source software using the Julia programming language.