Analysis of Resonance in Photonic Crystals

S. David Lively
Southern Methodist University

Photonic crystals - periodic dielectric structures - may be used to direct and contain electromagnetic waves. The ability of a crystal to affect wave propagation depends on several factors including the shape, scale and distribution of dielectrics as well as the relative wavelength of an incident wave. Simulation techniques such as FDTD may be used to model such structures and their response to different wavelengths in order to evaluate their suitability for a particular application. We present an analysis of three structures, their resonant wavelengths, and software modifications to enable efficient simulation of such structures.

Background

The Finite Difference Time Domain method is a popular, robust method to simulate and analyze the propagation of electromagnetic waves within different media and geometries. While typically applied to non-periodic structures such as dielectric slabs and complex circuits, the algorithm may also be used to simulate periodic structures.

For this document, the existing GoLightly simulator was modified to simulate periodic structures. The goals of such modifications include:

1. Compatibility with modern hardware
2. Tiling of image-based crystal unit cells
3. Multi-resolution simulation
4. Source wavelength sweep
5. Interactive monitor charting and analysis
6. Summary data export

The most-recent GoLightly implementation, created in 2015, utilized a combination of C++, CUDA[1] and OpenGL. While effective at the time it was created, several factors indicated that a reimplementation was required for this project. The available hardware – specifically, a 2019 16" MacBook pro with AMD GPU – presented several challenges.

CUDA requires access to an NVIDIA GPU. In addition, the CUDA version used by GoLightly is nearly a decade old. Updating the application to use a more modern CUDA version would have required substantial work which would not benefit the task at hand. CUDA kernels are based on the Cg shader language which is very similar to HLSL[2] used by Unity3D.

Given that available hardware consisted of a MacBook Pro with an AMD GPU, it seemed logical to port the CUDA kernels to HLSL. This is platform-agnostic system that may be cross-compiled for NVIDIA, AMD and Intel GPUs, and, through Unity3D, is available for all modern operating systems.

GoLightly's user interface requires a complex CUDA/OpenGL interoperation mechanism. MacOS support for OpenGL has been deprecated by Apple. While it is still possible to use OpenGL on MacOS, implementation of the FDTD kernels in HLSL precludes use of OpenGL. While the FDTD kernels could have been re-written in GLSL[3], the similarities between CUDA and HLSL make HLSL a more suitable platform.

---

[1] Compute Unified Device Architecture is NVIDIA's GPGPU development platform.
[2] High Level Shader Language, used by DirectX and Unity3D (rebranded as Shader Lab).
[3] GLSL is the shader programming language used for Vulkan and OpenGL.

Modern game development platforms such as Unity3D are well-suited for the creation of cross-platform GPU-centric applications with rich user interfaces. Several factors informed the decision to reimplement GoLightly in Unity3D.

- Cross-compilation support of HLSL to Metal, the preferred graphics API on MacOS
- Extensive library of user interface software components
- Rapid development iteration cycle
- Support for MacOS, Linux and Windows

The decision to reimplement GoLightly in Unity3D yielded many benefits including reduced software complexity. The C++ GoLightly implementation contained a substantial amount of code purely to manage the user interface. Unity3D provides much of this functionality by default, leading to the exclusion of many thousands of lines of source code.

Software Modifications

Once the Unity3D implementation reached feature parity with the C++ version, some modifications were required to facilitate support for periodic structures. The prior version relied on the scale-invariant nature of Maxwell's equations that describe the behavior of electromagnetic waves.

For instance, the FDTD algorithm defines the size of a Yee cell as $1/10^{th}$ of the wavelength of the highest frequency source in the simulation. All physical constants such as **u0**, **eps0**, and **c** are defined as 1. While these assumptions simplify some calculations in FDTD, they present issues when dealing with structures defined in physical units. Since analysis of the dielectric lattices requires a frequency sweep to identify resonant wavelengths, it is convenient to define the Yee grid in terms of feature size rather than wavelength[4].

To that end, we define the normal wavelength of the simulation in terms of the smallest feature size in the simulation. For the first structure to be analyzed[5], the smallest feature is a rod with diameter 480nm. This indicates that a spatial resolution of 48nm is sufficient to avoid aliasing and numerical artifacts. The frequency scanning portion of the simulation may then use wavelengths as low as 10*48nm= 480nm. (For each of the lattice simulations to be performed, the minimum wavelength of interest is on the order of 600nm.)

```
calculateSimulationParameters() {
    nmPerCell = rodDiameterNm / cellDivisor;
    rodSpacingCells = rodSpacingNm / nmPerCell;
    rodDiameterCells = rodDiameterNm / nmPerCell;
    lambdaNorm = lambdaNm / (10 * nmPerCell);
}
```

Figure 1 Calculation of FDTD parameters from physical model parameters. cellDivisor is >= 10

Having determined a rational way to convert physical units to the unitless parameter space used by GoLightly, a few other modifications were required.

GoLightly uses an image-based model definition format to eliminate any dependence on complex CAD applications or scripting. While that method is well-suited for describing complex waveguide geometries, investigation of the lattices of interest requires flexibility not easily represented by the image-based model definition. Specifically, it was required to repeatedly adjust the Yee grid resolution when exploring potential solutions. This required the creation of many versions of the lattice unit cell tile images at different resolutions.

[4] Choosing a minimum wavelength that is proportional the smallest feature of interest would also have been valid.

[5] A square lattice of rods with spacing of 1.2μm, diameter 480nm.

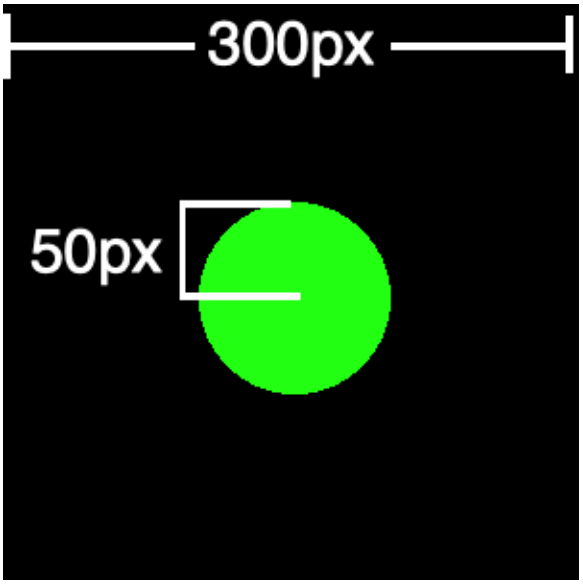

Figure 2 Square rod lattice unit cell texture with exterior dimension 300px and rod radius of 50px

For the square rod lattice, a unit cell tile image (Figure 2) was created with dimensions 300x300px with a green circle of diameter 100px. These values were chosen to provide sufficient resolution to avoid aliasing in the rod geometry. The 100% green value indicates the maximum dielectric value in the tile.

```
for (var j = 0; j < domainHeight; ++j) {
    var texelYOffset = (j % tileTexture.height) *
tile.width;
    for (var i = 0; i < domainWidth; ++i) {
        var texelAddr = texelYOffset + (i %
tileTexture.width);
        var color = texels[texelAddr];
        var cbAddr = j * domainWidth + i;
        if (color.g > 0)
            cb[cbAddr] = dielectric;
        else
            cb[cbAddr] = air;
    }
}
```

Figure 3 Replicating an image-based tile over the simulation domain

After generating several versions of this tile in a raster image editor, a Unity3D script was created to generate the unit cells which were then exported to the PNG.

Generating a tile image, exporting it to a PNG image, reloading and decoding that image and using it to populate the dielectric material array used by the FDTD kernels is an inefficient process. Since the routines to generate a tile had already been created, we chose to use them to directly populate the FDTD dielectric array. While this is a departure from GoLightly's design ethos[6], it simplified this use case.

```
var cellCount = new int2(
    domainWidth / rodSpacingCells + 1,
    domainHeight / rodSpacingCells + 1
    );
for (var j = 0; j < cellCount.y; ++j) {
    for (var i = 0; i < cellCount.x; ++i) {
        if (excludeCell[i, j]) continue;

        var x = offset + i * rodSpacingCells;
        var y = offset + j * rodSpacingCells;
        var center = new int2(x, y);
        makeCylinder(center, radius, domainSize,
domainHeight, dielectric, cb);
    }
}
```

Figure 4 Code to programmatically generate a rod array

Once the model generation system was complete, a software component was created to run the simulation with different conditions in a loop. Specifically, the source wavelength was incremented in a loop with user-defined band and resolution. Results were collected by summing the RMS power of each location within a series of FDTD monitors over 4000 time steps[7]. After each run, the E, H, PML and monitor buffers are cleared to effectively reset the simulation. The peak value and peak RMS for each monitor are captured to a CSV file for later analysis.

[6] If the application were meant to handle general simulation cases, an image-based tile format would be advantageous.

[7] 4000 steps was experimentally determined to be the approximate minimum number of frames required for the monitor values to reach steady state.

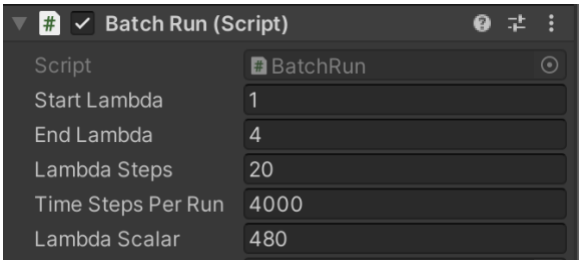

Figure 5 Wavelength Sweep Parameters

```
void Update() {
    if (simulation.timeStep >= timeStepsPerRun &&
!simulation.isPaused) {
        captureResult();
        _currentLambda += _lambdaDelta;
        if (_currentLambda <= endLambda) {
simulation.SetSingleSourceWavelengthAndReset(_cur
rentLambda);
        }
        else
        {
            simulation.isPaused = true;
            saveResultsToFile();
        }
    }
}

void captureResult() {
var rmsSum = 0f;
var valSum = 0f;
foreach(var monitor in _monitors) {
    rmsSum += monitor.rmsMaxValue;
    valSum += monitor.maxValue;
}
var line = $"{_currentLambda},{_currentLambda *
lambdaScalar},{timeStepsPerRun},{rmsSum},{valSum}
";
_results.AppendLine(line);
}
```

Figure 6 Code to sweep source wavelength and capture monitor results

The additional requirement of user-defined domain resolution[8] necessitated the implementation of a Model Provider component responsible for defining the domain size, Yee cell size, and other parameters as indicated in Figure 7.

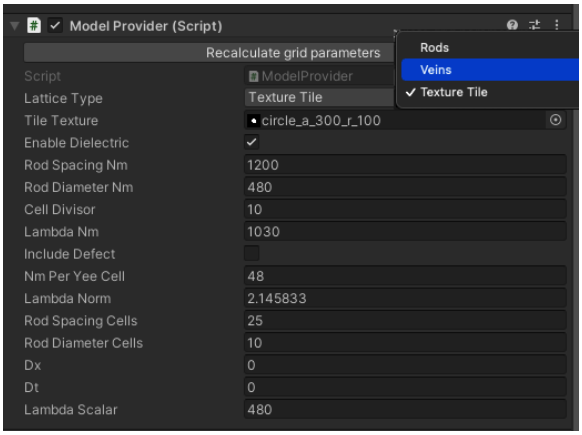

Figure 7 Model Definition Parameters

Figure 7 shows the model input parameters for the rod array lattice simulation. The Lattice Type lets the user choose between programmatic generation of the rod array or vein array, or to tile a unit cell image texture. The "Lambda Norm" parameter indicates the minimum relative source wavelength, defined as the ratio of lambda (nm) to 10 * nm per cell. The Cell Divisor parameter indicates the minimum source wavelength in terms of Yee cells. For the rod array experiment, the "Include Defect" parameter indicates whether the rod removal defect should be included in the resulting dielectric array. A more complete implementation would allow for defect locations to be defined in an image or configuration file.

[8] The resolution of the domain is defined as nm per Yee cell along each axis.

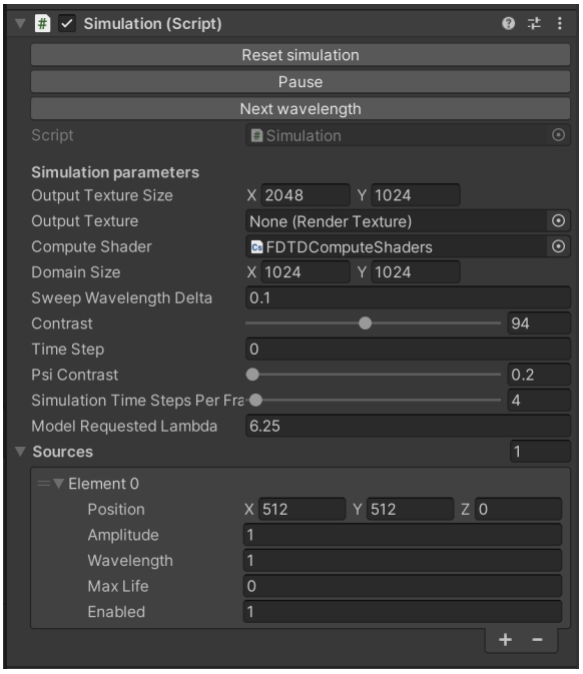

Figure 8 Simulation Configuration

The Simulation component (Figure 8) provides a configuration interface to control parameters such as the domain size in Yee cells, the variant of the FDTD compute shaders to be used, the current integer time step (updated while a simulation executes) and the number of FDTD frames to be calculated between user interface updates. Generating an image from an electric or magnetic field array requires substantial compute resources and is done infrequently to reduce execution time while maintaining an interactive frame rate.

In its previous incarnation, GoLightly also stored source and monitor locations within the PNG model using the red and blue color channels to indicate source wavelength and a monitor ID. The transition to a tile-based system necessitated a new approach. Source and monitor locations are now defined directly within the Unity3D UI. The Simulation component contains a list of sources (Figure 9).

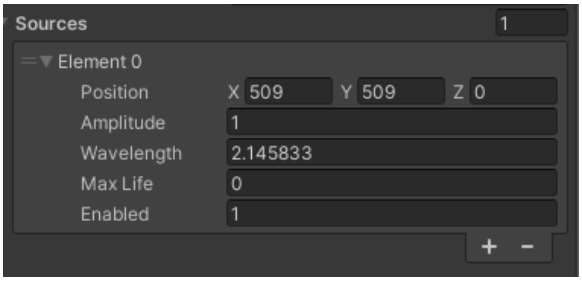

Figure 9 Source configuration interface

Monitors are generated via a dedicated Monitor Provider script.

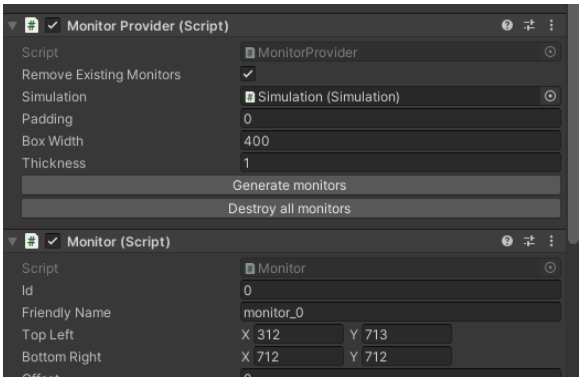

Figure 10 Monitor Provider interface

The Monitor Provider generates a symmetrical array of four monitors centered about a source location, with user-defined dimensions, thickness and spacing (Figure 11).

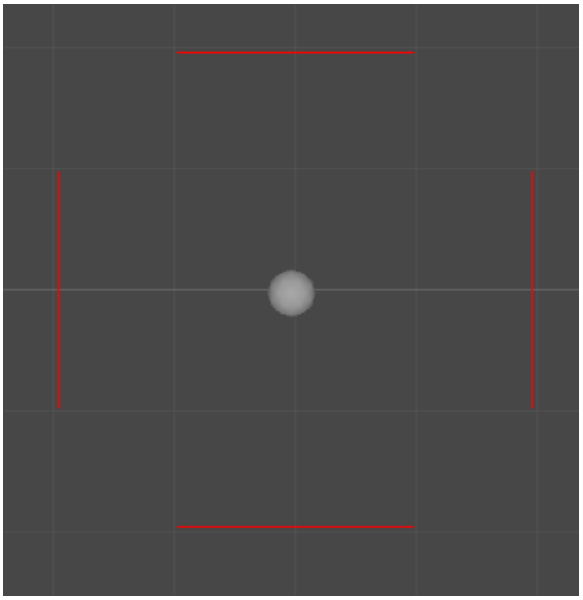

Figure 11 Monitor Array in Unity3D Scene View. The sphere at the center indicates the source position.

```
var center = simulation.domainSize / 2;
var halfWidth = boxWidth / 2;
var west = -halfWidth;
var east = halfWidth;
var north = halfWidth;
var south = -halfWidth;

var rects = new int[] {
    //top
    west+padding,north+thickness,east-
padding,north,
    // right
    east,north-
padding,east+thickness,south+padding,
    // bottom
    west+padding, south-thickness, east-padding,
south,
    // left
    west-thickness,north-padding, west,
south+padding,
};

var monitors = new List<Monitor>();
for (var i = 0; i < rects.Length; i += 4) {
    var monitor =
this.gameObject.AddComponent<Monitor>();
    monitor.topLeft = center + new
Vector2Int(rects[i], rects[i + 1]);
    monitor.bottomRight = center + new
Vector2Int(rects[i + 2], rects[i + 3]);
}
```

Figure 12 Generating a square array of monitors

At runtime, the Simulation component determines the linear domain offset of each cell that is contained within a monitor. Those values are placed in a linear array and uploaded to the GPU. At each frame update, a compute shader updates a buffer containing current Ez values for each monitor location.

The CSUpdateMonitors compute shader is invoked once for each Yee cell that is covered by a monitor, indicated by the index variable. That index is used to look up a value in the monitorAddresses array which contains the linear domain offset of that location. That location is used to find the Ez value corresponding to the monitor location.

```
[numthreads(64,1,1)]
void CSUpdateMonitors(uint3 id :
SV_DispatchThreadID) {
    int index = id.x;

    if (index >= numMonitorAddresses)
        return;

    int offset = monitorAddresses[index];
    // monitorValues[index] = pow(ez[offset],2);
    monitorValues[index] = ez[offset];
}
```

Figure 13 Updating Monitor values from Ez field

Experiments

For this project, three crystal definitions were analyzed at various wavelengths in order to determine resonance. These are detailed below.

Experiment 1: Rod array

A square lattice of rods in air with spacing a=1.2 µm, radius r=0.24µm and relative epsilon of 8.9.

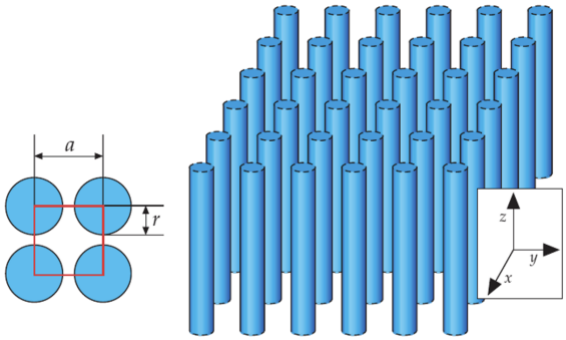

Figure 14 A square lattice of cylindrical rods. a=1.2um. r=0.24um.

A Yee cell size of 48nm[9] was chosen to minimize aliasing and ensure that rods occupied enough cells to accurately represent their interaction with the minimum wavelength defined as 480nm. Sweeping the input wavelength from 480nm to 1800nm in steps of 72nm yielded the results shown in Figure 5.

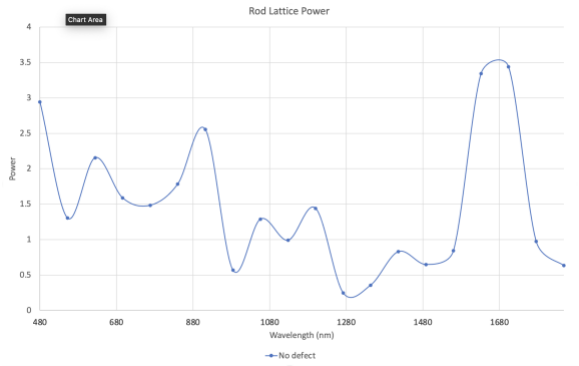

Figure 15 RMS output of a square lattice of rods

In this case, the area of interest is the minimum monitor sum indicating maximum confinement or resonance. Experimentally, this was determined to be a source wavelength of 1266nm[10].

Figure 15 shows the TMz E-field distribution.

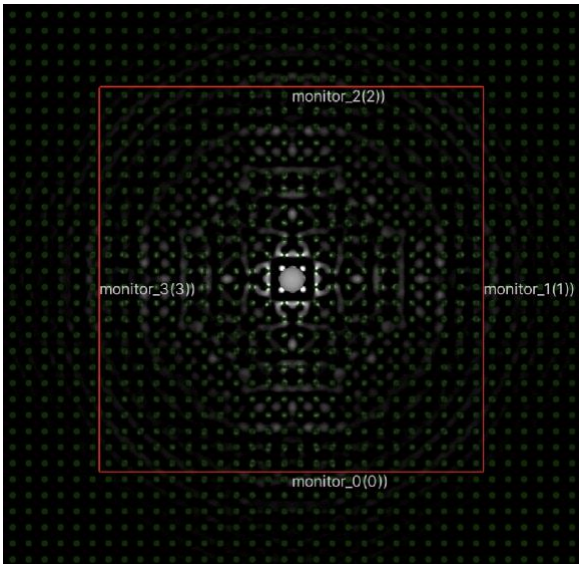

Figure 16 Ez field for square rod array with lambda 1266nm

The bright area at the center of the field plot indicates a high-magnitude Ez value. The wave remains confined within the rod array.

Experiment 2: Rod array with defect

For comparison, the same experiment was run with a missing rod adjacent to the source position (Figure 17).

[9] The Yee cell size is 10% of the rod diameter, or 48nm.

[10] Given the wavelength sweep resolution of 72nm, this value could be 1266 +- 36nm.

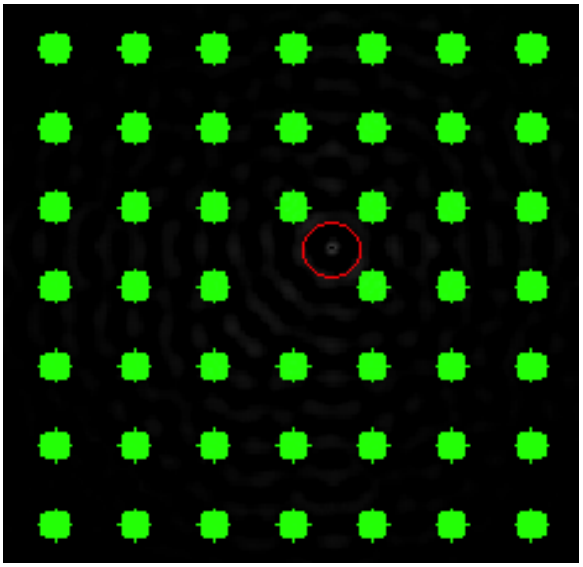

Figure 17 Rod lattice with missing rod defect. Source position is indicated in red.

Note the missing rod at the center of the field. The red circle indicates the source position. The same lattice was tested against the resonant wavelength of 1266nm. The resulting Ez field is portrayed in Figure 18.

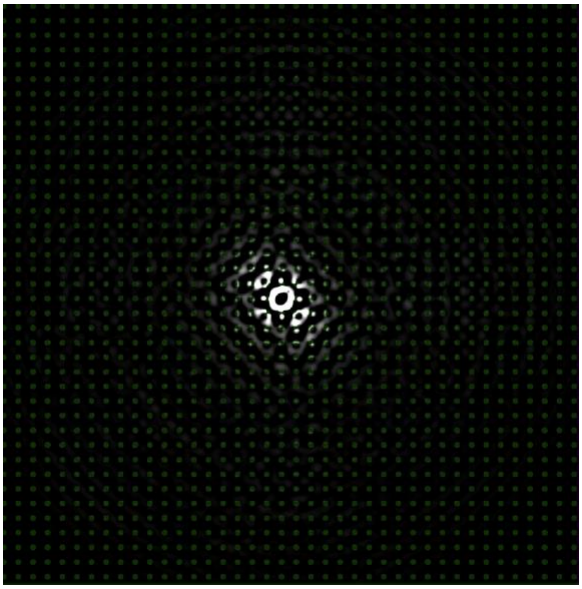

Figure 18 TMz Ez field - Square lattice after 4000 frames with lambda 1266nm

Note the apparent 45º rotation of the mode. This is due to the source location being slightly offset relative to the missing rod. This defect creates a resonant cavity which allows the wave to reflect within the empty space. The crystal still contains the wave just as in the version without the defect, but allows for some interesting things to happen in the interior.

An interesting side-effect of the offset source position is an oscillation between modes[11] (Figure 19).

[11] A short video of the oscillation is available at https://www.youtube.com/watch?v=g6s6uwqU-ds

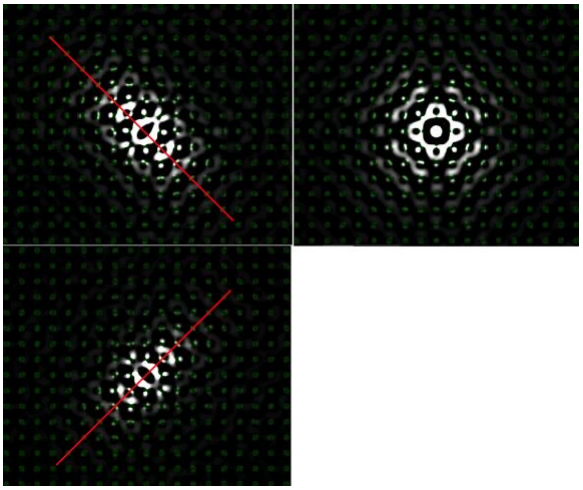

Figure 19 Resonance modes in rod array with defect

The peak RMS output of the perfect lattice and lattice with defect are shown in Figure 20.

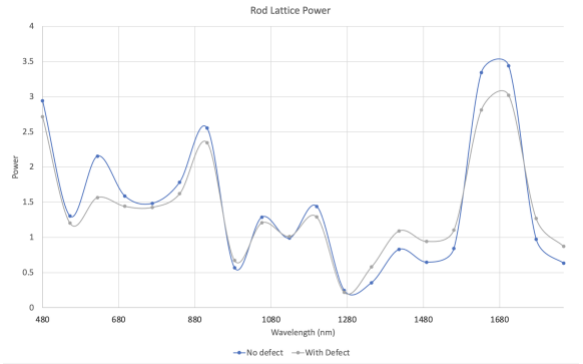

Figure 20 Rod lattice power output - defect vs non-defect

Note that the models have approximately equal resonant wavelengths, the most interesting of which occurs at approximately 1266nm. Differences in the peak power at different wavelengths may be accounted for by the directionality of the source waves. This is due to the offset source position relative to the rod lattice. In the defect-free experiment, the source was perfectly centered within a square of four rods. In the defect experiment, a single rod was removed, leaving the source slightly off-centered relative to the lattice.

These experiments indicate that the simulator correctly models wave confinement and may be used to identify resonant wavelengths given a lattice definition and a range of wavelengths.

Experiment 3: Square lattice of veins

For the final experiment, a square lattice of veins with spacing a = 1.2µm, width r=0.2a, and relative epsilon = 8.9.

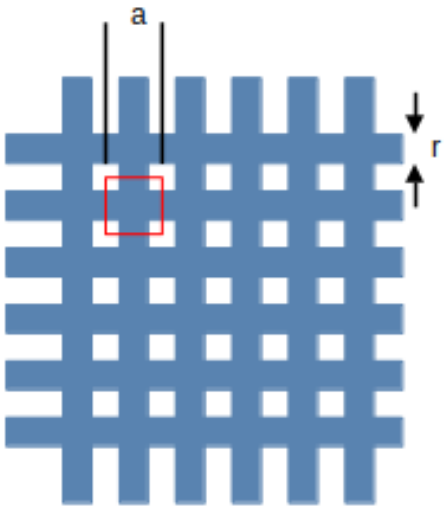

Figure 21 Square lattice of veins. a=1.2µm, r=0.24µm

This required modification of the Model Provider script to generate a unit cell consisting of a waffle-like pattern as indicated by the red square in Figure 21.

```
for (var j = 0; j < domainHeight; ++j) {
    var y = (j + offset) % domainHeight;
    var inVeinY = (y % cellWidth) <= veinWidth;
    for (var i = 0; i < domainWidth; ++i) {
        var x = (i + offset) % domainWidth;
        var inVeinX = (x % cellWidth) <=
veinWidth;
        if (inVeinX || inVeinY) {
            var addr = j * domainWidth + i;
            cb[addr] = dielectric;
        }
```

```
    }
}
```

Figure 22 Generating a rectangular lattice of veins

A wavelength sweep from 48nm to 1800nm with a step resolution of 48nm[12] is shown in Figure 23.

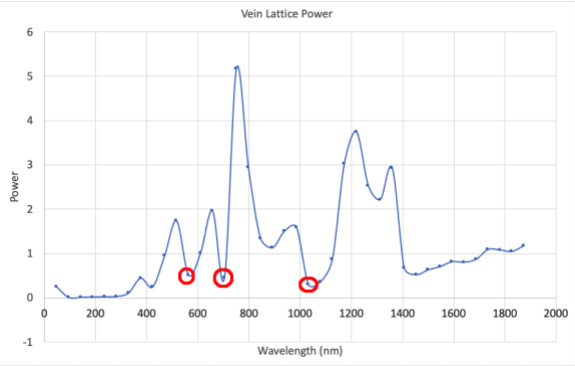

Figure 23 Vein Lattice Resonant Frequencies

Wavelengths less that the vein thickness are effectively blocked. Other resonant frequencies include 560nm, 703nm and 1030nm. Figure 24 shows the Ez field at 1030nm after 4000 time steps.

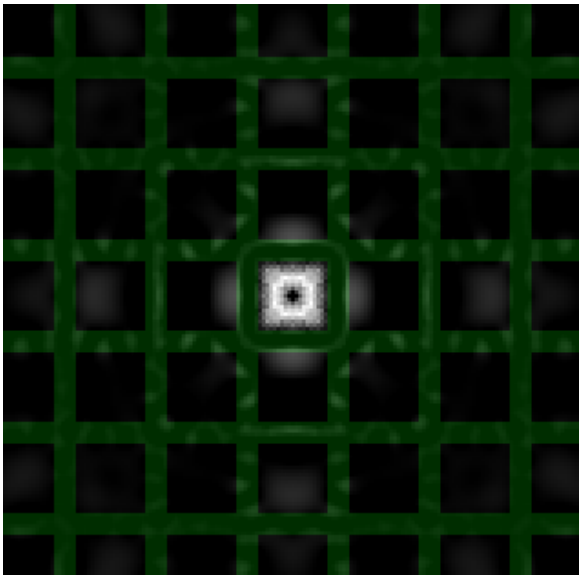

Figure 24 Lattice Resonance at 1030nm

Conclusion

The GoLightly FDTD simulator is capable of modeling photonic crystals. However, some modifications were required:

- Support for tiling image-based models
- Programmatic generation of models
- Flexible, configurable monitor and source definitions.
- Use of physical units rather than the unitless FDTD values

Most modifications were to support a tile-based dielectric rather than a monolithic model definition that covers the entire domain. While this was a significant departure from the previous incarnation of the software, it resulted in a more capable and user-friendly tool.

[12] Experimentation indicates that the square lattice inhibits propagation of smaller wavelengths.