Rectangular Patch Antenna Array Design Supplement

Lewis Collum

Contents

I. Library

A. Unit Test Runner

```
python -m unittest source/$name.py 2>&1
echo
```

B. conversion

```
import numpy
from numpy import pi, sin, cos, arccos, arctan2, sqrt
from collections import namedtuple

c = 2.99792458E8

def frequencyToWavelength(frequency):
    return c/frequency

def wavelengthToFrequency(wavelength):
    return c/wavelength

def waveNumber(wavelength):
    return 2*pi / wavelength
```

```
SphericalCoordinates = namedtuple('SphericalCoordinates', ['r', 'pitch', 'yaw'])
CartesianCoordinates = namedtuple('CartesianCoordinates', ['x', 'y', 'z'])

def sphericalToCartesian(r, yaw, pitch):
    x = r * cos(yaw) * sin(pitch)
    y = r * sin(yaw) * sin(pitch)
    z = r * cos(pitch)
    return CartesianCoordinates(x, y, z)

def sphericalVectorToCartesian(r, yaw, pitch):
    x = r[0] * cos(yaw) * sin(pitch)
    y = r[1] * sin(yaw) * sin(pitch)
    z = r[2] * cos(pitch)
    return CartesianCoordinates(x, y, z)

def cartesianToSpherical(x, y, z):
    r = sqrt(x**2 + y**2 + z**2)
    pitch = arccos(z / r)
    yaw = arctan2(y, x)
    return SphericalCoordinates(r, pitch, yaw)
```

C. *test_conversion*

```
import unittest
import conversion as conv
import numpy
from numpy import pi, sqrt

class TestConversion(unittest.TestCase):
    def test_cartesianToSpherical_xOnly(self):
        actual = conv.cartesianToSpherical(1, 0, 0)
        expected = conv.SphericalCoordinates(r = 1, pitch = pi/2, yaw = 0)

        numpy.testing.assert_array_almost_equal(actual, expected)


    def test_cartesianToSpherical_yOnly(self):
        actual = conv.cartesianToSpherical(0, 1, 0)
        expected = conv.SphericalCoordinates(r = 1, pitch = pi/2, yaw = pi/2)

        numpy.testing.assert_array_almost_equal(actual, expected)


    def test_cartesianToSpherical_zOnly(self):
        actual = conv.cartesianToSpherical(0, 0, 1)
        expected = conv.SphericalCoordinates(r = 1, pitch = 0, yaw = 0)

        numpy.testing.assert_array_almost_equal(actual, expected)


    def test_cartesianToSpherical_each(self):
        actual = conv.cartesianToSpherical(sqrt(2)/2, -sqrt(2)/2, 0)
        expected = conv.SphericalCoordinates(r = 1, pitch = pi/2, yaw = -pi/4)

        numpy.testing.assert_array_almost_equal(actual, expected)


    def test_waveNumber_14Ghz(self):
        frequency = 14E9
        wavelength = conv.frequencyToWavelength(frequency)

        actual = conv.waveNumber(wavelength)
        expected = 2*pi / (conv.c/frequency)

        numpy.testing.assert_almost_equal(actual, expected)


if __name__ == '__main__':
    unittest.main()
```

D. *patch*

```
import numpy
from numpy import pi, sin, cos, sqrt
from collections import namedtuple
from . import conversion as conv
from scipy.integrate import simps

Patch = namedtuple('Patch', ['width', 'height', 'length', 'effectiveLength', 'waveNumber'])
Material = namedtuple('Material', ['relativePermittivity', 'height'])

def rolloff(radians, factor):
    degrees = numpy.rad2deg(radians)
    F1 = 1 / (((factor*(abs(degrees) - 90))**2) + 0.001)
    return 1 / (F1 + 1)

class PatchRadiation:
    def __init__(self, patch, resolution):
        self.patch = patch
        self.resolution = resolution
        offset = pi/resolution
        self.pitch = numpy.linspace(offset, pi, resolution, endpoint=False)
        self.yaw = numpy.linspace(-pi/2, pi/2, resolution, endpoint=False)
```

```
    def hPlane(self):
        x = self.patch.waveNumber*self.patch.height/2 * sin(self.pitch)
        z = self.patch.waveNumber*self.patch.width/2 * cos(self.pitch)
        magnitudes = sin(self.pitch) * sin(x)/x * sin(z)/z
        return conv.SphericalCoordinates(
            r = magnitudes,
            yaw = 0,
            pitch = self.pitch)

    def ePlane(self):
        x = self.patch.waveNumber*self.patch.height/2 * cos(self.yaw)
        y = self.patch.waveNumber*self.patch.effectiveLength/2 * sin(self.yaw)
        magnitudes = cos(y) * sin(x)/x
        return conv.SphericalCoordinates(
            r = magnitudes,
            yaw = self.yaw,
            pitch = 90)

    def totalAsSpherical(self):
        ePlane, hPlane = numpy.meshgrid(self.ePlane().r, self.hPlane().r)
        yaw, pitch = numpy.meshgrid(self.yaw, self.pitch)
        radiation = ePlane * hPlane * rolloff(self.yaw, 0.3)
        return conv.SphericalCoordinates(
            r = numpy.nan_to_num(radiation),
            yaw = yaw,
            pitch = pitch)

    def directivity(self):
        total, pitch, yaw = self.totalAsSpherical()
        total = total/total.max()
        average = simps(simps(total, pitch[:, 0]), yaw[0])
        directivity = 4*numpy.pi/average
        return directivity


class PatchBuilder:
    def __init__(self, material):
        self.eR = material.relativePermittivity
        self.height = material.height

    def buildForFrequency(self, frequency):
        self.setDimensionsFromFrequency(frequency)
        return Patch(
            width = self.width,
            length = self.lengthWithoutFringe,
            height = self.height,
            effectiveLength = self.lengthWithFringe,
            waveNumber = conv.waveNumber(self.wavelength))


    def setDimensionsFromFrequency(self, frequency):
        self.wavelength = conv.frequencyToWavelength(frequency)
        self.width = self.widthFromWavelength(self.wavelength)
        widthToThicknessRatio = self.width/self.height

        eEff = self.effectiveRelativePermittivity(widthToThicknessRatio)
        self.lengthWithFringe = self.lengthEffective(self.wavelength, eEff)

        fringeExtension = self.fringeExtension(eEff, widthToThicknessRatio)
        self.lengthWithoutFringe = self.lengthWithFringe - 2*fringeExtension


    def widthFromWavelength(self, wavelength):
        return wavelength/2 * sqrt(2/(self.eR + 1))


    def effectiveRelativePermittivity(self, widthToThicknessRatio):
        linear = (self.eR + 1)/2
        coefficient = (self.eR - 1)/2
        nonlinear = (1 + 12/widthToThicknessRatio)**(-1/2)
        return linear + coefficient*nonlinear


    def lengthEffective(self, wavelength, eEff):
        return wavelength/2/sqrt(eEff)


    def fringeExtension(self, eEff, widthToThicknessRatio):
        numerator = (eEff + 0.3)*(widthToThicknessRatio + 0.264)
        denominator = (eEff - 0.258)*(widthToThicknessRatio + 0.8)
        extension = 0.412 * self.height * numerator/denominator
        return extension

    @property
    def heightEffective(self):
        return self.height*sqrt(self.eR)
```

E. *test_patch*

```
import unittest
from . import patch as pt
import numpy
from numpy import pi, sqrt
from . import conversion as conv
```

```
class TestPatchBuilder(unittest.TestCase):
    def setUp(self):
        rtDuroid5880 = pt.Material(
            relativePermittivity = 2.2,
            height = 0.1588E-2)
        patchBuilder = pt.PatchBuilder(rtDuroid5880)

        self.frequency = 10E9
        self.patch = patchBuilder.buildForFrequency(self.frequency)
        self.radiation = pt.PatchRadiation(self.patch, resolution = 501)
        self.plotter = pt.RadiationPlotter(self.radiation)

    def test_rtDuroid5880Build_matchesBalanisTextbook(self):
        actual = self.patch
        expected = pt.Patch(
            width = 1.186E-2,
            height = 0.1588E-2,
            length = 0.906E-2,
            effectiveLength = 1.068E-2,
            waveNumber = conv.waveNumber(conv.frequencyToWavelength(10E9)))

        numpy.testing.assert_array_almost_equal(actual, expected, 3)

if __name__ == '__main__':
    unittest.main()
```

F. *radiation_plotter*

```
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D
import numpy
from numpy import pi

from . import conversion as conv

class RadiationPlotter:
    def __init__(self, patchRadiation):
        self.radiation = patchRadiation

    def plotHPlane(self, fileName = None):
        hPlane = self.radiation.hPlane()
        radius = self.dB(hPlane.r)
        nanMask = ~numpy.isnan(radius)
        figure = pyplot.figure(figsize=(4,3))
        axes = figure.add_subplot(111, projection='polar')
        axes.plot(hPlane.pitch[nanMask], radius[nanMask])
        axes.set_thetamax(180)
        axes.set_xticks([0, pi/2, pi])
        axes.set_yticks(numpy.linspace(numpy.round(numpy.min(radius[nanMask])), 0, 3))
        axes.set_xlabel(r'$\theta$')
        axes.set_ylabel('dB')
        figure.tight_layout()
        self.showOrSave(fileName)

    def plotEPlane(self, fileName = None):
        ePlane = self.radiation.ePlane()
        figure = pyplot.figure(figsize=(4,3))
        axes = figure.add_subplot(111, projection='polar')
        axes.plot(ePlane.yaw, self.dB(ePlane.r))
        axes.set_thetamin(-90)
        axes.set_thetamax(90)
        axes.set_xlabel(r'$\phi$')
        axes.set_ylabel('dB')
        figure.tight_layout()
        self.showOrSave(fileName)

    def plotTotal(self, fileName = None):
        total = self.radiation.totalAsSpherical()
        normalized = total.r/total.r.max()
        x, y, z = conv.sphericalToCartesian(
            r = normalized,
            yaw = total.yaw,
            pitch = total.pitch)
        figure = pyplot.figure(figsize=(5,4))
        axes = figure.add_subplot(111, projection='3d')
        axes.plot_surface(z, y, x, rcount=100, ccount=100)
        axes.set_xlabel('z (along width)')
        axes.set_ylabel('y (along length)')
        axes.set_zlabel('x (along height)')
        axes.set_zlim(0, 1)
        axes.set_ylim(-0.5, 0.5)
        axes.set_xlim(-0.5, 0.5)

        figure.tight_layout()
        self.showOrSave(fileName)

    def dB(self, value):
        return 10 * numpy.log10(value)

    def showOrSave(self, fileName):
        if fileName:
            pyplot.savefig(fileName)
            pyplot.clf()
```

```
        else:
            pyplot.show()
```

G. test_radiation_plotter

```
from source import patch as pt
from source import plotter

rtDuroid5880 = pt.Material(
    relativePermittivity = 2.2,
    height = 0.1588E-2)
patchBuilder = pt.PatchBuilder(rtDuroid5880)

frequency = 10E9
patch = patchBuilder.buildForFrequency(frequency)
radiation = pt.PatchRadiation(patch, resolution = 501)
plotter = plotter.RadiationPlotter(radiation)

plotter.plotHPlane()#'figure/patch_hPlane.png')
plotter.plotEPlane()#'figure/patch_ePlane.png')
plotter.plotTotal()#'figure/patch_total.png')
```

H. array

```
from collections import namedtuple
import numpy
from numpy import sin, cos
from scipy.integrate import simps

from . import conversion as conv
from . import patch as pt

class ArrayRadiation:
    def __init__(self, radiation, wavelength, elements):
        self.radiation, self.pitch, self.yaw = radiation
        self.elements = elements
        self.wavelength = wavelength
        self.waveNumber = conv.waveNumber(wavelength = self.wavelength)

    def totalAsSpherical(self):
        return conv.SphericalCoordinates(
            r = numpy.abs(self.arrayFactor()) * self.radiation,
            yaw = self.yaw,
            pitch = self.pitch)

    def arrayFactor(self):
        elementSum = 0
        for element in self.elements:
            relativePhase = self.relativePhase(element)
            elementSum += numpy.exp(1j*relativePhase)
        return elementSum.real

    def relativePhase(self, element):
        z, y, x = conv.sphericalVectorToCartesian(
            r = element,
            yaw = self.yaw,
            pitch = self.pitch)

        return self.waveNumber * (x + y + z)

    def directivity(self):
        total, pitch, yaw = self.totalAsSpherical()
        total = total/total.max()
        average = simps(simps(total, pitch[:, 0]), yaw[0])
        directivity = 4*numpy.pi/average
        return directivity


Element = namedtuple('Element', ['x', 'y', 'z'])
def spacedPositions(shape, spacing):
    offsets = -spacing*(numpy.asarray(shape)-1)/2
    positions = []
    for x in range(shape[0]):
        for y in range(shape[1]):
            for z in range(shape[2]):
                positions.append(Element(
                    x = x*spacing + offsets[0],
                    y = y*spacing + offsets[1],
                    z = z*spacing + offsets[2]))
    return positions
```

Abstract

We re-create an example design for a patch antenna, given by Balanis [1], to simulate an array of patches using an RT/duroid 5880 substrate. We include common metrics, such as directivity. We also include illustrations for the far-zone radiation pattern of both a single patch and array of patches.

II. Single Patch Antenna: Physical Design

A. Design Context

We start by designing a single patch, which we will use to create an antenna array (in section IV).

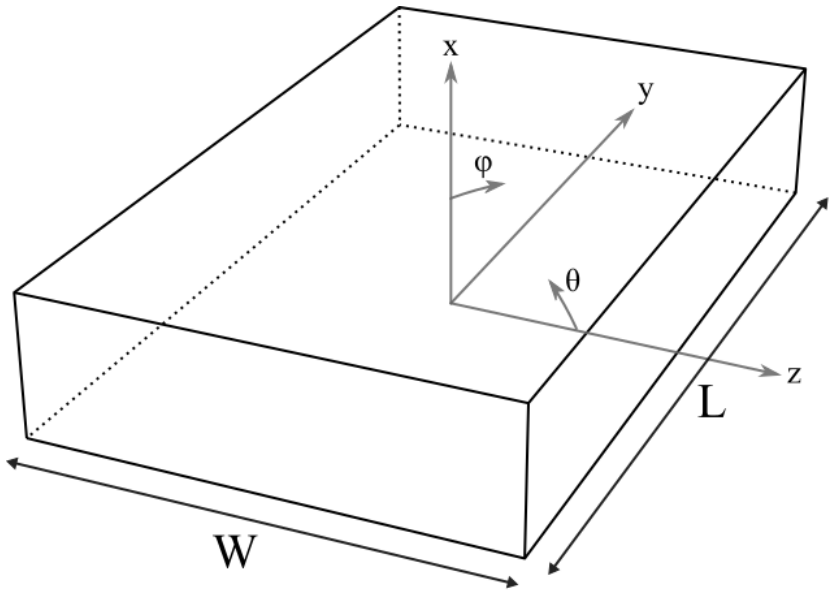

Fig. 1. Patch Model with spherical (θ and ϕ) and cartesian coordinates.

For the patch substrate, we are using RT/duroid 5880, which has a relative permittivty of $\epsilon_r = 2.2$. The height of the material is designed to be $h = 0.1588\,\mathrm{cm}$. Our patch is intended to resonate at $f_r = 10\,\mathrm{GHz}$.

With these three parameters, we can find the patch width and height. Then, we can illustrate the radiation pattern for the patch.

B. Fringing Effects

Fringing fields at the lengths of the patch makes the patch appear to have a greater length than it actually does. This is important since the effective dimensions of the patch affect the resonant frequency. If the physical length of the patch is L, then the effective length, L_{eff}, can be written as

$$L_{eff} = L + 2 \cdot \Delta L, \tag{1}$$

where ΔL is the additional length on one end of the patch.

The additional length can be related to the width of the patch, W and the effective relative permittivity of the dieletric substrate, ϵ_{eff}, as [1]

$$\frac{\Delta L}{h} = 0.412 \frac{(\epsilon_{eff} + 0.3)\left(\frac{W}{h} + 0.264\right)}{(\epsilon_{eff} - 0.258)\left(\frac{W}{h} + 0.8\right)}. \tag{2}$$

The effective relative permittivity, ϵ_{eff}, is given as [1]

$$\epsilon_{eff} = \frac{\epsilon_r + 1}{2} + \frac{\epsilon_r - 1}{2}\left(1 + 12 \cdot \frac{h}{W}\right)^{-1/2}. \tag{3}$$

C. Physical Width

A model for patch width is [1]

$$W = \frac{\lambda_r}{2}\sqrt{\frac{2}{\epsilon_r + 1}} \tag{4}$$

where λ is the wavelength at the resonance frequency, f_r.

$$\lambda = \frac{c}{10\,\mathrm{GHz}} = 3.00\,\mathrm{cm}$$

The patch's physical width, from eq. 4, is

$$\boxed{W = 1.185\,\mathrm{cm}}$$

D. Effective Length

A model for patch length (without accounting for fringing effects) is, [1]

$$L_{eff} = \frac{\lambda}{2\sqrt{\epsilon_{eff}}} \tag{5}$$

Using eq. 3, we find the effective relative permittivity to be

$$\begin{aligned}\epsilon_{eff} &= \frac{2.2+1}{2} + \frac{2.2-1}{2}\left(1 + 12 \cdot \frac{0.1588\,\text{cm}}{1.185\,\text{cm}}\right) \\ &= 1.97.\end{aligned}$$

Thus,

$$\boxed{L_{eff} = 1.07\,\text{cm}}$$

E. Physical Length

The physical length is the effective length minus the extensions lengths caused by fringing fields, as per eq. 1. By eq. 2, the extension length on one side, ΔL, is

$$\Delta L = 0.0825\,\text{cm}$$

So,

$$\begin{aligned}L &= L_{eff} - 2\Delta L \\ &= 1.07\,\text{cm} - 2 \cdot 0.0825\,\text{cm} \\ &= \boxed{0.905\,\text{cm}}\end{aligned}$$

F. Summary

The width, height, and effective length, are used in following calculations as the effective dimensions of the patch. The actual length of the patch is used for building the antenna and impedance matching, but it is not used to analyze the radiation pattern of the patch.

III. Single Patch Antenna: Analysis

A. E-Plane ($\theta = 90°, -90° \leq \phi \leq 90°$)

The radiation intensity in the E-plane can be modeled as [1]

$$F_\phi = \cos(y_\phi) \cdot \frac{\sin(x_\phi)}{x_\phi} \tag{6a}$$

$$x_\phi = \frac{k_0 h}{2}\cos(\phi) \tag{6b}$$

$$y_\phi = \frac{k_0 L_{eff}}{2}\sin(\phi), \tag{6c}$$

where k_0 is the wave-number,

$$k_0 = \frac{2\pi}{\lambda} = 209.6.$$

```
from source import patch as pt
from source import plotter as pl
from source import conversion as conv

rtDuroid5880 = pt.Material(
    relativePermittivity = 2.2,
    height = 0.1588E-2)
patchBuilder = pt.PatchBuilder(rtDuroid5880)

frequency = 10E9
wavelength = conv.frequencyToWavelength(frequency)
patch = patchBuilder.buildForFrequency(frequency)
radiation = pt.PatchRadiation(patch, resolution = 100)
```

```
plotter = pl.RadiationPlotter(radiation)
plotter.plotEPlane(f'figure/patch_ePlane.png')
```

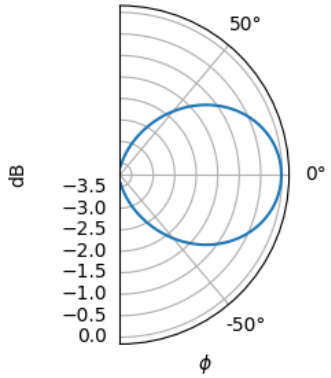

Fig. 2. E-Plane radiation pattern of the patch under design.

B. *H-Plane* ($\phi = 0°, 0° \leq \theta \leq 180°$)

The radiation intensity in the H-plane can be modeled as [1]

$$F_\theta = \sin(\theta) \cdot \frac{\sin(x_\theta)}{x_\theta} \cdot \frac{\sin(z_\theta)}{z_\theta} \tag{7a}$$

$$x_\theta = \frac{k_0 h}{2} \sin(\theta) \tag{7b}$$

$$z_\theta = \frac{k_0 W}{2} \cos(\theta), \tag{7c}$$

using wave-number, k_0, from section III-A.

```
from source import patch as pt
from source import plotter as pl
from source import conversion as conv

rtDuroid5880 = pt.Material(
    relativePermittivity = 2.2,
    height = 0.1588E-2)
patchBuilder = pt.PatchBuilder(rtDuroid5880)

frequency = 10E9
wavelength = conv.frequencyToWavelength(frequency)
patch = patchBuilder.buildForFrequency(frequency)
radiation = pt.PatchRadiation(patch, resolution = 100)

plotter = pl.RadiationPlotter(radiation)
plotter.plotHPlane(f'figure/patch_hPlane.png')
```

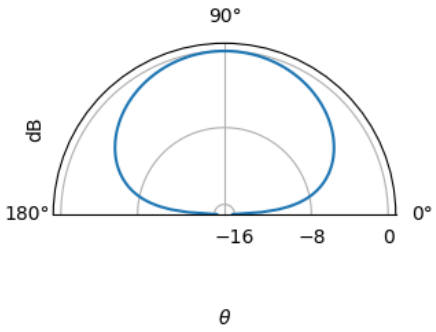

Fig. 3. H-Plane radiation pattern of the patch under design.

C. Far-Zone Total Radiation Intensity

Since the two plane models are normalized, and since the planes are orthogonal, the total radiation intensity is approximately the product of the two plane models.

That is,

$$F(\phi, \theta) = F_\phi \cdot F_\theta. \tag{8}$$

```
from source import patch as pt
from source import plotter as pl
from source import conversion as conv

rtDuroid5880 = pt.Material(
    relativePermittivity = 2.2,
    height = 0.1588E-2)
patchBuilder = pt.PatchBuilder(rtDuroid5880)

frequency = 10E9
wavelength = conv.frequencyToWavelength(frequency)
patch = patchBuilder.buildForFrequency(frequency)
radiation = pt.PatchRadiation(patch, resolution = 100)

plotter = pl.RadiationPlotter(radiation)
plotter.plotTotal(f'figure/patch_total.png')
```

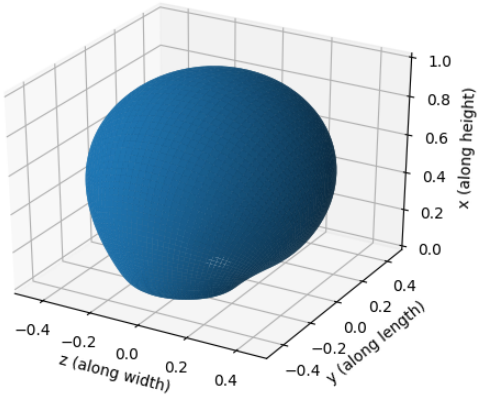

Fig. 4. Normalized radiation pattern of the patch under design using eq. 8.

D. Directivity

Directivity of the main lobe (in figure 4) is the maximum radiation intensity over the average intensity, [2]

$$D = \frac{F_{max}}{F_{average}} = \frac{1}{\frac{1}{4\pi}\Omega_p} \tag{9}$$

where Ω_p is the pattern solid angle. Note that for a *normalized* radiation intensity, the maximum, F_{max}, will be equal to 1, by definition.

The pattern solid angle is defined as [2]

$$\Omega_p = \iint_{4\pi} F(\phi, \theta)\, d\theta\, d\phi. \tag{10}$$

Since we have the electric field intensity stored as a 2-dimensional array, we use simpson's rule to calculate the double integral.

This yields a directivity of,

```
import numpy
from source import patch as pt
from scipy.integrate import simps

rtDuroid5880 = pt.Material(
    relativePermittivity = 2.2,
    height = 0.1588E-2)
patchBuilder = pt.PatchBuilder(rtDuroid5880)

patch = patchBuilder.buildForFrequency(10E9)
radiation = pt.PatchRadiation(patch, resolution = 100)
directivity = radiation.directivity()
```

```
print((f'\[D = {directivity:.2f}\]'))
```

$$D = 3.17$$

IV. Patch Grid

A. Array Factor

The array factor is a function that incorporates the positions of the patches in the array. It optionally includes weights and phase offsets. We use a simplified model which only accounts for position of elements, and assumes no element phase offset, a uniform element amplitude of one, and a uniform wave number, written as

$$AF = \sum_{i=1}^{N} e^{-jk_0|r_i|} \tag{11}$$

where $k_0|r_i|$ is the relative phase at patch `i` located at $r = (x_i, y_i, z_i)$.
The relative phase at each patch, $k_0|r_i|$, describes the phase variation for the position of the element, as

$$k_0|r_i| = \sin\theta\cos\phi z + \sin\theta\sin\phi y + \cos\theta x. \tag{12}$$

Importantly, z and x would be flipped in eq. 12, if we used z to represent the "up" axis.
We can plot the array factor for a 3x3 patch array, to visualize the characteristics of the pattern.

```
import numpy
from numpy import pi
from source import plotter as pl
from source import conversion as conv
from source import array as pa

frequency = 10E9
wavelength = conv.frequencyToWavelength(frequency)

yaw, pitch = numpy.meshgrid(numpy.linspace(-pi/2, pi/2), numpy.linspace(0, pi))
radiation = conv.SphericalCoordinates(r = 1, yaw = yaw, pitch = pitch)

elements = pa.spacedPositions((1,3,3), wavelength/2)
array = pa.ArrayRadiation(radiation, wavelength, elements)
plotter = pl.RadiationPlotter(array)
plotter.plotTotal('figure/patchArray_3x3_af.png')
```

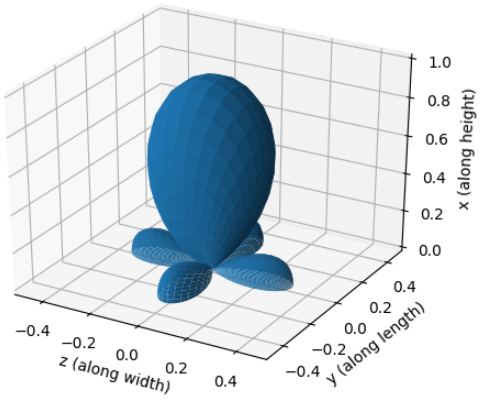

Fig. 5. 3x3 Array Factor

B. Radiation Pattern

For an array, the total radiation intensity is the normalized product of the array factor and the radiation intensity for a single patch [1]. That is,

$$F(\phi, \theta) = AF \cdot F(\phi, \theta)_0 \tag{13}$$

We use this relationship to illustrate a 2x2, 3x3, and 4x4 patch array.

```
from source import patch as pt
from source import plotter as pl
from source import conversion as conv
```

```
from source import array as pa

rtDuroid5880 = pt.Material(
    relativePermittivity = 2.2,
    height = 0.1588E-2)
patchBuilder = pt.PatchBuilder(rtDuroid5880)

frequency = 10E9
wavelength = conv.frequencyToWavelength(frequency)
patch = patchBuilder.buildForFrequency(frequency)
radiation = pt.PatchRadiation(patch, resolution = 100)
for i in range(1,5):
    elements = pa.spacedPositions((1,i,i), wavelength/2)
    array = pa.ArrayRadiation(radiation.totalAsSpherical(), wavelength, elements)
    plotter = pl.RadiationPlotter(array)
    plotter.plotTotal(f'figure/patchArray_{i}x{i}.png')
```

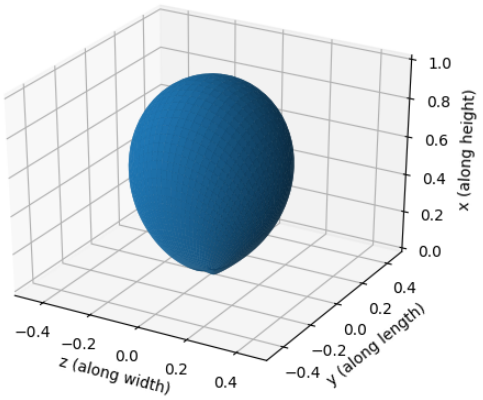

Fig. 6. 2x2 Patch Array

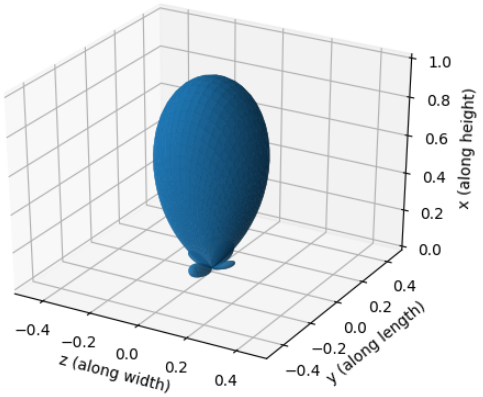

Fig. 7. 3x3 Patch Array

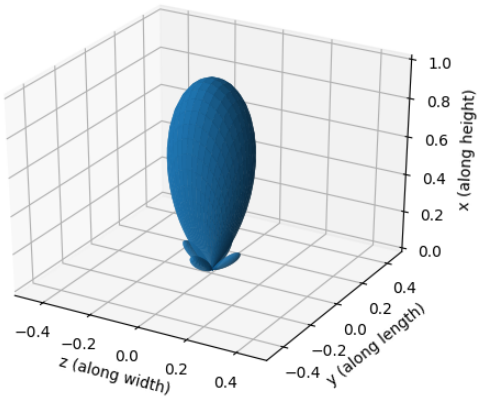

Fig. 8. 4x4 Patch Array

C. Directivity

Using eq. 13 for total array radiation and plugging the result into the solid angle equation (eq. 10), we find the directivity (eq. 9) for each patch array.

This yields patch array directivities of,

```
import numpy
from source import patch as pt
from source import plotter as pl
from source import conversion as conv
from source import array as pa

rtDuroid5880 = pt.Material(
    relativePermittivity = 2.2,
    height = 0.1588E-2)
patchBuilder = pt.PatchBuilder(rtDuroid5880)

frequency = 10E9
wavelength = conv.frequencyToWavelength(frequency)
patch = patchBuilder.buildForFrequency(frequency)
radiation = pt.PatchRadiation(patch, resolution = 100)

x = numpy.arange(2,5)
for i in x:
    elements = pa.spacedPositions((1,i,i), wavelength/2)
    array = pa.ArrayRadiation(radiation.totalAsSpherical(), wavelength, elements)
    endPunctuation = '.' if i == x[-1] else ','
    print(f'\[D_{{{i}x{i}}} = {array.directivity():.2f}{endPunctuation}\]')
```

$$D_{2x2} = 7.30,$$

$$D_{3x3} = 12.53,$$

$$D_{4x4} = 19.61.$$

In theory, the more patches we add to the array, the higher the directivity (and gain) becomes. As a note, gain is

$$G = \epsilon D,$$

where ϵ is the antenna efficiency. We assume an efficiency of 100% for this analysis, so we obtain no additional by calculating gain.

V. Conclusion

We extended the theory for designing a single patch, to begin designing an antenna array. While there are still many design factors not considered, such as efficiency and impedance matching, this design acts as a first step towards including those factors.

References

[1] C. A. Balanis, *Antenna Theory: Analysis and Design*. John Wiley Sons, second ed., 1997.
[2] F. T. Ulaby and U. Ravaioli, *Fundamentals of Applied Electromagnetics*. Pearson, seventh ed., 2015.