WaveCalc

Ryan Goetz [1]

June 13, 2019

[1] rgoetz@phys.ufl.edu

Contents

Chapter 1

The Wave Calculator

1.1 Motivating a Wave Tool

Much of the utility, depth, and breadth of experimental optics is predicated upon the wealth of electromagnetic wave interactions with optical materials. From materials we build lenses, mirrors, prisms, polarizers, retarders, rotators, isolators, gratings, modulators, resonators, and all sorts of things between. Even in optical experiments wherein the signal of interest is generated by a laser field's propagation through vacuum, there will undoubtedly be a plethora of these aforementioned optical components whose role is to condition the field for its scientific purpose.

Unfortunately, if the reader has only been exposed to optical phenomena as taught in the standard electromagnetism courses, they are likely only familiar with a small subset of wave behavior in materials. Particularly, common core coursework will often restrict itself to isotropic media, only very rarely stepping out into the muddier waters of anisotropy, and even then only considering highly sanitized cases. As an example, most students and optics researchers will be well-versed in the basic operating principle behind a half-wave plate. Two orthogonal polarizations in the medium undergo a differential phase shift of π over the length of the material, and so the effect on any incoming field whose linear polarization is not entirely in one of these special directions is that the polarization has been rotated upon exit. But what happens when the waveplate moves from the paper to the laboratory, where it must be fixed in a physical mount and aligned to a laser field (likely by hand)? How does the output field change as the waveplate is pitched and yawed about the beam axis? Most people in the field of optics will understand that such misalignments give rise to ellipticity of the output field, but how much? How do we quantify the coupling between ellipticity and misalignment? And are there more effects to which we are unaware? Technically, anyone who has exposure to Maxwell's equations has the framework to answer these questions, however the path to the answers is not so short and for many has never been tread. In large part, this lack of exposure to the fuller picture of electromagnetism motivated the creation of the WaveCalc tool.

In isotropic media, electromagnetic wave behavior is both well-understood and straightforward to calculate. The propagation of waves within the medium is nearly identical to the case of wave solutions in free space. Electric fields oscillate in the plane of their phase-fronts. Rules governing reflection and transmission of waves across media boundaries can be written in greatly simplified forms, far removed from the vector and tensor calculus of Maxwell's equations. What results are familiar relationships such as Snell's Law, the Law of Reflection, or the Fresnel equations. However, when we extend the analysis beyond isotropic media, we quickly discover that these well-known and reliable rules are merely special cases of more complicated–yet more fundamental–results.

In designing WaveCalc, we sought to construct a tool capable of examining electromagnetic wave behavior in the simplest case: plane waves in linear, non-permeable, dielectric media. Though not

physically realizable, plane waves are reasonable models of laser fields for many applications. Similarly, many optical media of interest can be approximated as linear dielectrics; material in which there is no free charge and the electric displacement responds linearly to an incident electric field. Even in nonlinear media, the nonlinear terms are often small enough that the contribution from the laser's own electric field is negligible. We do not extend out analysis beyond dielectrics, as media in which there are free charges tend to quickly attenuate electromagnetic waves, and therefore are usually less interesting as optical materials.

Chapter 2 discusses the theoretical background for calculating plane wave behavior in linear dielectric materials; beginning from Maxwell's equations to enumerate the plane-wave modes of a medium, and ultimately characterizing the interaction of a wave with a media boundary. Chapter 3 then addresses how the results of Chapter 2 might be implemented algorithmically to incorporate into a calculation tool. Chapters 4 and beyond introduce the `wavecalc` Python package, and explain how it can be used to perform calculations of interest.

1.2 `wavecalc`

The implementations outlined in Chapter 3 have been coded in Python and packaged as `wavecalc`. This package provides tools for easily calculating plane wave behavior in dielectric media and at the interface of media. Both as attributes, function arguments, and core functions, WaveCalc relies heavily on objects from the `numpy` package; and so this package is a dependency for `wavecalc`.

In this text we will use example code to illustrate the properties and functionality of the `wavecalc` package. In doing so, we use the convention that any line beginning with `>>` is input code, and any line following is an output of the input code:

```
>> input code
output of code
```

With the exception of the line above, all example code in this text should be executable by the reader if done in the order presented. Lines that are neither input code nor following input code are often intended to give an understanding of syntax, but are not intended to be passed to an interpreter.

Technically, in Python everything is an object, however it makes sense to distinguish between the Python objects which are representative of physical entities and the Python objects which are representative of physical processes. The former we refer to as objects, and the latter as functions. Chapter 4 introduces these WaveCalc objects, Chapter 5 explains the methods associated with these objects, and Chapter 6 describes the primary set of WaveCalc functions. In Chapter 7, we will demonstrate the utility of WaveCalc with some examples. Chapter 8 is included as an interesting investigation into the algebra of some WaveCalc operations, but is not necessary to understand the usage of `wavecalc`.

To work with WaveCalc and follow along with the example code, the user will need to import the `wavecalc` package into their Python interpreter. And because a great number of WaveCalc objects have `numpy` objects as attributes, it is recommended to also import `numpy` whenever planning to use `wavecalc`:

```
>> import wavecalc as wc
>> import numpy
Importing wavecalc.classes as obj
Importing wavecalc.functions as fun
```

The reader will notice that we have chosen to alias `wavecalc` as `wc`. We do so in the interest of brevity, and hopefully not at the expense of clarity for the reader.

.

Chapter 2

Plane Waves in Dielectric Materials

In this chapter we lay the groundwork for the physical theory behind the machinations of WaveCalc. In Section 2.1 we will enumerate the plane-wave modes permitted in a dielectric medium. In Section 2.2 we will examine these modes to identify special directions in the medium, referred to as optic axes. In Section 2.3 we will show how boundary conditions can be applied to relate the wave vector on either side of a material boundary. Section 2.4 addresses boundary conditions on the electric field, and how they may be used to compute reflected and transmitted fields.

2.1 Wave Propagation in a Medium and the Booker Quartic

Maxwell's equations in a dielectric material (no free charges or currents) read:

$$\begin{aligned} \nabla \cdot \mathbf{D} = 0 \qquad & \nabla \cdot \mathbf{B} = 0 \\ \nabla \times \mathbf{E} = -\tfrac{\partial \mathbf{B}}{\partial t} \qquad & \nabla \times \mathbf{H} = \tfrac{\partial \mathbf{D}}{\partial t} \end{aligned} \tag{2.1}$$

For a linear medium the displacement vector is related to the electric field by the permittivity tensor:

$$\mathbf{D} = \varepsilon_0 \overleftrightarrow{\boldsymbol{\varepsilon}} \cdot \mathbf{E} \tag{2.2}$$

here we have chosen the convention such that $\overleftrightarrow{\boldsymbol{\varepsilon}}$ is unitless. In the simple case which we consider, $\overleftrightarrow{\boldsymbol{\varepsilon}}$ is both spatially and temporally independent. Further, we suppose that the medium is non-permeable, that is $\mu_0 \mathbf{H} = \mathbf{B}$. Under these assumptions, we can eliminate the magnetic fields and write:

$$\nabla \times \nabla \times \mathbf{E} = -\mu_0 \frac{\partial^2 \mathbf{D}}{\partial t^2} \tag{2.3}$$

As we are interested in wave solutions to Maxwell's equations, our ansatz is a plane-wave electric field:

$$\mathbf{E} = \mathbf{E}_0 e^{i(\mathbf{k}\cdot\mathbf{r} - \omega t)} \tag{2.4}$$

where $\mathbf{E}_0$ is a spatially and temporally invariant vector which encodes the amplitude and polarization of the wave, $\mathbf{k}$ is called the wave vector and determines how the phase of the wave evolves through space, and ω is a scalar which determines how the phase evolves over time. We have made the common conventional choice to represent the electric field with a complex-valued vector. This helps to simplify the algebra and calculus, however the reader should keep in mind that the physically meaningful quantity is the real part of this complex vector.

Using (2.2) and (2.4) along with the identity $\nabla \times \nabla \times \mathbf{E} = \nabla(\nabla \cdot \mathbf{E}) - \nabla^2 \mathbf{E}$, we can rewrite (2.3) as an algebraic equation:

$$-(\mathbf{k} \cdot \mathbf{E})\mathbf{k} + k^2 \mathbf{E} = \mu_0 \varepsilon_0 \omega^2 \overleftrightarrow{\varepsilon} \cdot \mathbf{E} \tag{2.5}$$

where $k^2 = \mathbf{k} \cdot \mathbf{k}$. We have arrived at what the author considers to be the most disappointing of all electromagnetic equations. Invariably, (2.5) shows up in textbooks throughout all time and space, though in its present form its utility is limited; and for our cause it will be limited to providing a solitary insight. In the event that the medium is vacuum, $\overleftrightarrow{\varepsilon}$ becomes the identity, and the Maxwell equation $\nabla \cdot \mathbf{D} = 0$ causes the first term on the left-hand side of (2.5) to vanish. When this happens, we see that the wave number of vacuum, denoted k_0, can be related to the angular frequency of the wave, ω, and fundamental constants by $k_0^2 = \mu_0 \varepsilon_0 \omega^2$. Evidently, all electromagnetic waves in vacuum propagate with speed $c = 1/\sqrt{\mu_0 \varepsilon_0}$, the so-called speed of light.

As far as we are concerned, the usefulness of the form of (2.5) has been exhausted. In its stead, we reconstruct (2.5) as an eigenvector equation:

$$\left(\mathbf{k} \otimes \mathbf{k} + k_0^2 \overleftrightarrow{\varepsilon}\right) \cdot \mathbf{E} = k^2 \mathbf{E} \tag{2.6}$$

where the $\mathbf{k} \otimes \mathbf{k}$ denotes the outer product of the wave vector with itself. We see that wave solutions for the electric field are eigenvectors of the operator $\mathbf{k} \otimes \mathbf{k} + k_0^2 \overleftrightarrow{\varepsilon}$ with eigenvalues k^2. We pause here for a moment, as this result is fundamental: every plane wave mode of a medium having the form given in (2.4) has a field amplitude vector which is an eigenvector of the operator on the left-hand side of (2.6). We will call this operator the Maxwell wave operator. Further, every plane wave mode must have a wave vector whose self dot product is an eigenvalue of the electric field amplitude vector under the Maxwell wave operator.

The eigenvalues of the Maxwell wave operator are the roots of the characteristic equation:

$$\left|\mathbf{k} \otimes \mathbf{k} + k_0^2 \overleftrightarrow{\varepsilon} - k^2 \overleftrightarrow{\mathbf{I}}\right| = 0 \tag{2.7}$$

which constitutes a condition on the wave vector $\mathbf{k}$. Though when writing out the terms it might initially appear to be a sixth-order equation in k, the highest order terms cancel and we are left with a quartic that can be written in Cartesian coordinates as:

$$\begin{aligned} 0 =& k_0^4 \left|\overleftrightarrow{\varepsilon}\right| - k_0^2 \left[k_x^2 \left(M_{yy} + M_{zz}\right) + k_y^2 \left(M_{xx} + M_{zz}\right) + k_z^2 \left(M_{xx} + M_{yy}\right)\right] \\ & - k_0^2 \left[k_x k_y \left(M_{xy} + M_{yx}\right) + k_y k_z \left(M_{yz} + M_{zy}\right) - k_x k_z \left(M_{xz} + M_{zx}\right)\right] \\ & + k^2 \left[k_x^2 \varepsilon_{xx} + k_y^2 \varepsilon_{yy} + k_z^2 \varepsilon_{zz} + k_x k_y \left(\varepsilon_{xy} + \varepsilon_{yx}\right) + k_y k_z \left(\varepsilon_{yz} + \varepsilon_{zy}\right) + k_x k_z \left(\varepsilon_{xz} + \varepsilon_{zx}\right)\right] \end{aligned} \tag{2.8}$$

where M_{ij} are the minors of the permittivity tensor. By some true genius or dumb luck (I don't have access to most of the publications discussing it), somebody discovered that this could be expressed in the more compact form:

$$0 = k^2 \left(\mathbf{k} \cdot \overleftrightarrow{\varepsilon} \cdot \mathbf{k}\right) + k_0^2 \mathbf{k} \cdot \left\{\text{adj}\left(\overleftrightarrow{\varepsilon}\right) - \text{tr}\left[\text{adj}\left(\overleftrightarrow{\varepsilon}\right)\right] \overleftrightarrow{\mathbf{I}}\right\} \cdot \mathbf{k} + k_0^4 \left|\overleftrightarrow{\varepsilon}\right| \tag{2.9}$$

When written in this form, (2.9) is known as the Booker quartic. I really wish I could derive this from (2.7) without brute force; it's so pretty. Though the general solution to quartic equations can be quite involved, notice that (2.9) is biquadratic (it only involves even powers of k), and so if we define:

$$\begin{aligned} A &= \hat{\mathbf{k}} \cdot \overleftrightarrow{\varepsilon} \cdot \hat{\mathbf{k}} \\ B &= k_0^2 \hat{\mathbf{k}} \cdot \left\{\text{adj}\left(\overleftrightarrow{\varepsilon}\right) - \text{tr}\left[\text{adj}\left(\overleftrightarrow{\varepsilon}\right)\right] \overleftrightarrow{\mathbf{I}}\right\} \cdot \hat{\mathbf{k}} \\ C &= k_0^4 \left|\overleftrightarrow{\varepsilon}\right| \end{aligned} \tag{2.10}$$

then the solutions of the companion equation:

$$0 = q^2\left(\hat{\mathbf{k}}\cdot\overleftrightarrow{\varepsilon}\cdot\hat{\mathbf{k}}\right) + qk_0^2\hat{\mathbf{k}}\cdot\left\{\text{adj}\left(\overleftrightarrow{\varepsilon}\right) - \text{tr}\left[\text{adj}\left(\overleftrightarrow{\varepsilon}\right)\right]\overleftrightarrow{\mathbf{I}}\right\}\cdot\hat{\mathbf{k}} + k_0^4\left|\overleftrightarrow{\varepsilon}\right| \tag{2.11}$$

given by:

$$q_\pm = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \tag{2.12}$$

lead directly to the solutions of (2.9):

$$\begin{aligned} k_{a+} &= \sqrt{q_-} & k_{a-} &= -\sqrt{q_-} \\ k_{b+} &= \sqrt{q_+} & k_{b-} &= -\sqrt{q_+} \end{aligned} \tag{2.13}$$

We should be careful not to confuse the solutions given by (2.13) with the magnitude of $\mathbf{k}$ (as well should be obvious since magnitudes are never negative). Instead, for any given unit vector $\hat{\mathbf{k}}$, the solutions to the Booker quartic yield the allowed wave vectors by $k_{(a,b)(+,-)}\hat{\mathbf{k}}$. We therefore interpret the left and right columns of (2.13) as the forward and backward propagating waves, respectively. For this reason, we might sometimes choose to drop the second subscript and simply write k_a and k_b as the wave solutions. The biquadratic nature of the Booker quartic for a fixed $\hat{\mathbf{k}}$ reflects the symmetry of forward and backward propagation.

As a direct consequence of Maxwell's equations for linear dielectrics, we have found that for any given propagation direction and frequency there are two wave vectors allowed by the medium. We refer to these two waves as the a-wave and the b-wave. Each of these waves have electric fields which are eigenvectors of the corresponding operator $\mathbf{k}\otimes\mathbf{k} + k_0^2\overleftrightarrow{\varepsilon}$. When the discriminant in (2.12) vanishes, the a- and b-waves are degenerate in their wave vectors; we call this special direction an optic axis of the medium.

2.2 Optic Axes of a Dielectric Medium: UNDER CONSTRUCTION

When the discriminant in (2.12) vanishes, the a- and b-waves are degenerate in their wave vectors; we call this special direction an optic axis of the medium. This condition as explicitly written:

$$B^2 - 4AC = 0 \tag{2.14}$$

is in general a condition on both $\hat{\mathbf{k}}$ and $\overleftrightarrow{\varepsilon}$.

2.2.1 Isotropic Media: UNDER CONSTRUCTION

By definition, in an isotropic medium the permittivity tensor must remain invariant for any choice of orthonormal basis, and therefore is a scalar multiple of the identity: $\overleftrightarrow{\varepsilon} = \varepsilon\overleftrightarrow{\mathbf{I}}$. In this case, the quartic coefficients simplify greatly:

$$\begin{aligned} A &= \varepsilon\hat{\mathbf{k}}\cdot\overleftrightarrow{\mathbf{I}}\cdot\hat{\mathbf{k}} = \varepsilon \\ B &= k_0^2\hat{\mathbf{k}}\cdot\left(\varepsilon^2\overleftrightarrow{\mathbf{I}} - 3\varepsilon^2\overleftrightarrow{\mathbf{I}}\right)\cdot\hat{\mathbf{k}} = -2\varepsilon^2k_0^2 \\ C &= \varepsilon^3k_0^4 \end{aligned} \tag{2.15}$$

from which we can see that equality in (2.14) is always satisfied. The allowed wave vectors are easily found:

$$k_\pm = \pm\sqrt{\varepsilon}k_0 \tag{2.16}$$

2.2.2 Uniaxial Media: UNDER CONSTRUCTION

$$\overleftrightarrow{\boldsymbol{\varepsilon}} = \begin{pmatrix} \varepsilon_1 & 0 & 0 \\ 0 & \varepsilon_1 & 0 \\ 0 & 0 & \varepsilon_2 \end{pmatrix} \tag{2.17}$$

$$A = \varepsilon_1 \hat{k}_x^2 + \varepsilon_1 \hat{k}_y^2 + \varepsilon_2 \hat{k}_z^2 \tag{2.18}$$

2.2.3 Biaxial Media: UNDER CONSTRUCTION

$$\overleftrightarrow{\boldsymbol{\varepsilon}} = \begin{pmatrix} \varepsilon_x & 0 & 0 \\ 0 & \varepsilon_y & 0 \\ 0 & 0 & \varepsilon_z \end{pmatrix} \tag{2.19}$$

$$\begin{aligned} A &= \varepsilon_x \hat{k}_x^2 + \varepsilon_y \hat{k}_y^2 + \varepsilon_z \hat{k}_z^2 \\ B &= -k_0^2 \left[\varepsilon_x \left(\varepsilon_y + \varepsilon_z \right) \hat{k}_x^2 + \varepsilon_y \left(\varepsilon_x + \varepsilon_z \right) \hat{k}_y^2 + \varepsilon_z \left(\varepsilon_x + \varepsilon_y \right) \hat{k}_z^2 \right] \\ C &= k_0^4 \varepsilon_x \varepsilon_y \varepsilon_z \end{aligned} \tag{2.20}$$

$$0 = \left[\varepsilon_x \left(\varepsilon_y + \varepsilon_z \right) \hat{k}_x^2 + \varepsilon_y \left(\varepsilon_x + \varepsilon_z \right) \hat{k}_y^2 + \varepsilon_z \left(\varepsilon_x + \varepsilon_y \right) \hat{k}_z^2 \right]^2 - 4 \varepsilon_x \varepsilon_y \varepsilon_z \left(\varepsilon_x \hat{k}_x^2 + \varepsilon_y \hat{k}_y^2 + \varepsilon_z \hat{k}_z^2 \right) \tag{2.21}$$

2.3 Wave Vector Boundary Conditions

As much of our interest is in characterizing wave behavior at media interfaces, we are not content to enumerate the optical modes in a dielectric: we must also understand how a wave incident on a medium reflects and transmits. When considering a boundary, whatever the material properties on either side, it must be the case that the phase of the wave is continuous at the boundary. More explicitly, the phase of the incoming wave must equal the phase of the refracted wave which must equal the phase of the reflected wave (or be 180 degrees out of phase depending on your perspective). In terms of the wave vectors of the three waves, this must mean that their components tangent to the surface normal must all be equal. Specifically, if $\hat{\mathbf{s}}$ is the normal vector to the boundary, and $\mathbf{k}_i$, $\mathbf{k}_j$ are wave vectors corresponding to the incoming or outgoing waves, then:

$$\hat{\mathbf{s}} \times (\mathbf{k}_i - \mathbf{k}_j) = 0 \tag{2.22}$$

This observation enables a convenient choice for coordinate system. Let the boundary be the x-y plane, with the incoming media, $\overleftrightarrow{\boldsymbol{\varepsilon}}_1$, corresponding to $z < 0$ and the outgoing media, $\overleftrightarrow{\boldsymbol{\varepsilon}}_2$, to $z > 0$. If the tangential component of the incoming (or reflected or refracted) wave vector is non-zero, choose the x-axis so that $\mathbf{k}_\perp = \rho \hat{\mathbf{x}}$ for some constant ρ. In the event that the tangential component is zero, the choice of x- and y- axes can be made at the reader's convenience.

With this convention, we know that all waves under consideration must have equal k_x and $k_y = 0$. For brevity, it is helpful to define the following two tensors for the media on either side of the boundary: $\overleftrightarrow{\boldsymbol{\sigma}} = \overleftrightarrow{\boldsymbol{\varepsilon}} + (\overleftrightarrow{\boldsymbol{\varepsilon}})^T$ and $\overleftrightarrow{\boldsymbol{\delta}} = \mathrm{tr}(\overleftrightarrow{\boldsymbol{\varepsilon}}) \overleftrightarrow{\mathbf{I}} - \overleftrightarrow{\boldsymbol{\varepsilon}}$. Then, adopting notation from Pettis (mostly), we let:

$$\begin{aligned}
\Delta_j &= k_x \sigma_{xz}^{(j)} \\
\Sigma_j &= k_x^2 \delta_{yy}^{(j)} - k_0^2 \left(M_{xx}^{(j)} + M_{yy}^{(j)} \right) \\
\Psi_j &= k_x^3 \sigma_{xz}^{(j)} + k_0^2 k_x \left(M_{xz}^{(j)} + M_{zx}^{(j)} \right) \\
\Gamma_j &= k_x^4 \varepsilon_{xx} - k_0^2 k_x^2 \left(M_{yy}^{(j)} + M_{zz}^{(j)} \right) + k_0^4 \left| \overleftrightarrow{\varepsilon}_j \right|
\end{aligned} \tag{2.23}$$

Here j is the index of the medium; for the reflected case $j = 1$, and for the transmitted case $j = 2$. We can rewrite the Booker quartic from (2.9) as an equation in k_z:

$$\varepsilon_{zz} k_z^4 + \Delta_j k_z^3 + \Sigma_j k_z^2 + \Psi_j k_z + \Gamma_j = 0 \tag{2.24}$$

Unlike in the previous subsection, this quartic is not identically biquadratic, nor in general would we expect it to be: we have not fixed a propagation direction, but instead fixed the tangential component of the wave vector. In the event that $k_x = 0$, we can see from (2.23) that the odd-order terms vanish and we recover the biquadratic; because in this special case we are fixing $\mathbf{k}$ to be in the z-direction.

What we gather from (2.24) is that an incoming mode to a boundary has four corresponding modes in both the incident and transmission media. What may be less immediately obvious is that the incoming mode will excite two modes in the incident media, corresponding to waves with $k_z < 0$, and two modes in the transmission media with $k_z > 0$. Thus, a wave incident on a material boundary gives rise to four new waves. This is a departure from the more familiar case of isotropic media, where every incoming wave has one reflected and one transmitted wave. As it happens, when the materials are isotropic, the two reflected waves share a wave vector, and the same for the transmitted waves, and hence can be considered one wave each.

At this point it is worthwhile to pause and reflect on our results to this point; there is a subtlety which would be easy to miss but has important implications for our tool. In Section 2.1 we derived the Booker quartic from Maxwell's equations and a plane-wave solution ansatz, given as (2.9). This equation defines a surface, $\mathcal{S}$, of the allowed wave vector modes of a medium, and this surface has the following property: for any unit vector $\hat{\mathbf{u}}$, there are two associated scalars, κ_a, κ_b, such that $\pm\kappa_a \hat{\mathbf{u}}$ and $\pm\kappa_b \hat{\mathbf{u}}$ lie on the surface. Because the unit sphere is compact in $\mathbb{C}^3$, and the roots of the Booker quartic are continuous for continuously varying vector directions, the surface over complex 3-space is bounded. Because of this, every component of any vector $\mathbf{k} \in \mathcal{S}$ is also bounded. However our analysis above places no such restraint on the k_x component, for example. Indeed, for any value of k_x, (2.24) will yield solutions for k_z. How is this not a contradiction?

2.4 Electric Field Boundary Conditions

In the previous section we outlined how a resultant wave vector can be related to an input wave. However, this is only half of the story: to fully describe the resultant wave we need the fields as well. Without information on how the fields are reflected and transmitted at an interface, we would be incapable of exploring many common phenomenon of optically interesting dielectrics such as: double refraction, optical activity, polarization, and more.

Suppose we have two media, labeled medium 1 and medium 2, connected by a planar boundary with unit surface normal $\hat{\mathbf{s}}$. As we saw in Section 2.3, every wave incident on a boundary will result in four new waves: two reflected waves, and two transmitted waves. Thus, the electric field on one

side of the boundary is the sum of three waves, the incident and two reflected, while the field on the other side is the sum of the two transmitted waves. In general, we will only be given the incident wave, and must calculate the four other waves.

Label the reflected waves α and β, with wave vectors $\mathbf{k}_\alpha$ and $\mathbf{k}_\beta$, and label the transmitted waves γ and ν with wave vectors $\mathbf{k}_\gamma$ and $\mathbf{k}_\nu$. Name the corresponding reflected and transmitted electric field amplitude vectors $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, and $\boldsymbol{\nu}$. In this way, we can write the fields on either side of the boundary (without time dependence) as:

$$\mathbf{E}_1 = \mathbf{E}_i e^{i\mathbf{k}_i\cdot\mathbf{r}} + \boldsymbol{\alpha} e^{i\mathbf{k}_\alpha\cdot\mathbf{r}} + \boldsymbol{\beta} e^{i\mathbf{k}_\beta\cdot\mathbf{r}} \tag{2.25a}$$

$$\mathbf{E}_2 = \boldsymbol{\gamma} e^{i\mathbf{k}_\gamma\cdot\mathbf{r}} + \boldsymbol{\nu} e^{i\mathbf{k}_\nu\cdot\mathbf{r}} \tag{2.25b}$$

here of course, $\mathbf{E}_i e^{i\mathbf{k}_i\cdot\mathbf{r}}$ is the incident wave. From Section 2.1, we know the reflected and transmitted fields must be eigenvectors of their respective Maxwell wave operators:

$$\left(\mathbf{k}_\alpha \otimes \mathbf{k}_\alpha + k_0^2 \overleftrightarrow{\boldsymbol{\varepsilon}}_1\right)\cdot\boldsymbol{\alpha} = k_\alpha^2\boldsymbol{\alpha} \tag{2.26a}$$

$$\left(\mathbf{k}_\beta \otimes \mathbf{k}_\beta + k_0^2 \overleftrightarrow{\boldsymbol{\varepsilon}}_1\right)\cdot\boldsymbol{\beta} = k_\beta^2\boldsymbol{\beta} \tag{2.26b}$$

$$\left(\mathbf{k}_\gamma \otimes \mathbf{k}_\gamma + k_0^2 \overleftrightarrow{\boldsymbol{\varepsilon}}_2\right)\cdot\boldsymbol{\gamma} = k_\gamma^2\boldsymbol{\gamma} \tag{2.26c}$$

$$\left(\mathbf{k}_\nu \otimes \mathbf{k}_\nu + k_0^2 \overleftrightarrow{\boldsymbol{\varepsilon}}_2\right)\cdot\boldsymbol{\nu} = k_\nu^2\boldsymbol{\nu} \tag{2.26d}$$

Our assumption is that we are given $\mathbf{k}_i$, and so the wave vector boundary conditions allow us to calculate all four reflection and transmission wave vectors using (2.24). Thus, the operators on the left-hand side, and the eigenvalues on the right-hand side of (2.26) are known, and therefore we can also calculate the field amplitude eigenvectors for reflection and transmission. With this knowledge, it is helpful to rewrite (2.25) as:

$$\mathbf{E}_1 = \mathbf{E}_i e^{i\mathbf{k}_i\cdot\mathbf{r}} + \hat{\boldsymbol{\alpha}} R_\alpha e^{i\mathbf{k}_\alpha\cdot\mathbf{r}} + \hat{\boldsymbol{\beta}} R_\beta e^{i\mathbf{k}_\beta\cdot\mathbf{r}} \tag{2.27a}$$

$$\mathbf{E}_2 = \hat{\boldsymbol{\gamma}} T_\gamma e^{i\mathbf{k}_\gamma\cdot\mathbf{r}} + \hat{\boldsymbol{\nu}} T_\nu e^{i\mathbf{k}_\nu\cdot\mathbf{r}} \tag{2.27b}$$

where R_α and R_β are the reflection coefficients for the incident wave to the α- and β-waves respectively, and similarly T_γ and T_ν are the transmission coefficients for the incident wave to the γ- and ν-waves. The form of (2.27) makes our problem more explicit, we are attempting to solve for four unknown quantities: R_α, R_β, T_γ, and T_ν.

To solve for these four unknowns we must impose boundary conditions on the electromagnetic fields. From Maxwell's equations together with Gauss's law and Stokes' theorem, we can arrive at four conditions:

$$0 = \hat{\mathbf{s}} \times (\mathbf{E}_1 - \mathbf{E}_2) \tag{2.28a}$$

$$0 = \hat{\mathbf{s}} \cdot (\mathbf{D}_1 - \mathbf{D}_2) \tag{2.28b}$$

$$0 = \hat{\mathbf{s}} \cdot (\mathbf{B}_1 - \mathbf{B}_2) \tag{2.28c}$$

$$0 = \hat{\mathbf{s}} \times (\mathbf{H}_1 - \mathbf{H}_2) \tag{2.28d}$$

where we are to understand that these fields are all evaluated immediately at either side of the boundary; and so will be the convention for the remainder of this section. Without loss of generality, we set $\hat{\mathbf{s}}$ to be pointing from medium 1 into medium 2. While we have included all four boundary conditions for completeness, our analysis will only make use of (2.28a) and (2.28d) from here onward.

We now make the coordinate choice such that $\hat{\mathbf{s}} = \hat{\mathbf{z}}$ and the tangential component of $\mathbf{k}_i$, if non-zero, lies in the x-direction. From (2.28a), we can make use of this choice to derive two linear equations in the reflection and transmission coefficients:

$$0 = E_{ix} + R_\alpha \hat{\alpha}_x + R_\beta \hat{\beta}_x - T_\gamma \hat{\gamma}_x - T_\nu \hat{\nu}_x \tag{2.29a}$$

$$0 = E_{iy} + R_\alpha \hat{\alpha}_y + R_\beta \hat{\beta}_y - T_\gamma \hat{\gamma}_y - T_\nu \hat{\nu}_y \tag{2.29b}$$

To make use of the magnetic field boundary conditions, we must first express them in a more helpful form. In our plane-wave ansatz, we can rewrite Maxwell's electric curl equation as:

$$i\mathbf{k} \times \mathbf{E} = -\frac{\partial}{\partial t}\mathbf{B} \tag{2.30}$$

Taking the antiderivative and neglecting any constant (non-waving) terms yields an algebraic expression for the magnetic field:

$$\mathbf{k} \times \mathbf{E} = \omega \mathbf{B} \tag{2.31}$$

As we are only considering non-permeable materials, this allows us to write (2.28d) as:

$$0 = \hat{\mathbf{s}} \times (\mathbf{k}_i \times \mathbf{E}_i + \mathbf{k}_\alpha \times \boldsymbol{\alpha} + \mathbf{k}_\beta \times \boldsymbol{\beta} - \mathbf{k}_\gamma \times \boldsymbol{\gamma} - \mathbf{k}_\nu \times \boldsymbol{\nu}) \tag{2.32}$$

which in our coordinate system yields the following two linear equations:

$$0 = k_{iz} E_{iy} + R_\alpha k_{\alpha z} \hat{\alpha}_y + R_\beta k_{\beta z} \hat{\beta}_y - T_\gamma k_{\gamma z} \hat{\gamma}_y - T_\nu k_{\nu z} \hat{\nu}_y \tag{2.33a}$$

$$\begin{aligned} 0 = {} & (k_{iz} E_{ix} - k_{ix} E_{iz}) + R_\alpha \left(k_{\alpha z} \hat{\alpha}_x - k_{\alpha x} \hat{\alpha}_z\right) + R_\beta (k_{\beta z} \hat{\beta}_x - k_{\beta x} \hat{\beta}_z) \\ & - T_\gamma \left(k_{\gamma z} \hat{\gamma}_x - k_{\gamma x} \hat{\gamma}_z\right) - T_\nu \left(k_{\nu z} \hat{\nu}_x - k_{\nu x} \hat{\nu}_z\right) \end{aligned} \tag{2.33b}$$

Notice that (2.33a) is substantially more compact than (2.33b); this is because with our coordinate choice all y-components of the wave vectors vanish. We can combine (2.29) and (2.33) as one system of equations for our four unknowns:

$$\begin{pmatrix} \hat{\alpha}_x & \hat{\beta}_x & -\hat{\gamma}_x & -\hat{\nu}_x \\ \hat{\alpha}_y & \hat{\beta}_y & -\hat{\gamma}_y & -\hat{\nu}_y \\ k_{\alpha z}\hat{\alpha}_y & k_{\beta z}\hat{\beta}_y & -k_{\gamma z}\hat{\gamma}_y & -k_{\nu z}\hat{\nu}_y \\ (k_{\alpha z}\hat{\alpha}_x - k_{\alpha x}\hat{\alpha}_z) & (k_{\beta z}\hat{\beta}_x - k_{\beta x}\hat{\beta}_z) & (k_{\gamma x}\hat{\gamma}_z - k_{\gamma z}\hat{\gamma}_x) & (k_{\nu x}\hat{\nu}_z - k_{\nu z}\hat{\nu}_x) \end{pmatrix} \begin{pmatrix} R_\alpha \\ R_\beta \\ T_\gamma \\ T_\nu \end{pmatrix} = \begin{pmatrix} -E_{ix} \\ -E_{iy} \\ -k_{iz}E_{iy} \\ k_{ix}E_{iz} - k_{iz}E_{ix} \end{pmatrix} \tag{2.34}$$

Solving (2.34) allows us to completely specify the fields on either side of the boundary.

Chapter 3

Implementation

In this chapter we discuss how the results of Chapter 2 are consolidated and algorithmically implemented to produce desired outputs. The intent is not to detail each step of a calculation or function in `wavecalc`, but rather to give a reader a sense for the general framework behind the package and thus an understanding of its capabilities and usefulness.

3.1 Eigenmodes of the Medium

Perhaps the simplest question a user might want resolved is what electromagnetic wave modes are allowed in a given medium. As we have seen in Section 2.1, every direction of propagation has two (possibly degenerate) modes associated with it determined by the Booker quartic. We can therefore begin by choosing a unit direction $\hat{\mathbf{k}}$. As we are given the medium, which in our case amounts to being given the dielectric tensor $\overleftrightarrow{\boldsymbol{\varepsilon}}$, we can construct the Booker quartic coefficients given in (2.10). We then solve the quartic to obtain the four solutions $k_{(a,b)(+,-)}$. At this point, however, because we interpret k_{a-} and k_{b-} as waves traveling in the opposite direction of $\hat{\mathbf{k}}$, we can ignore these solutions and declare $k_{a+}\hat{\mathbf{k}}$ and $k_{b+}\hat{\mathbf{k}}$ as the two wave vector modes in our chosen direction.

To find the corresponding fields for each wave vector eigenmode, we solve the two eigenvector equations:

$$\left(k_{a+}^2\hat{\mathbf{k}}\otimes\hat{\mathbf{k}}+k_0^2\overleftrightarrow{\boldsymbol{\varepsilon}}\right)\cdot\hat{\mathbf{e}}_a=k_{a+}^2\hat{\mathbf{e}}_a \qquad \left(k_{b+}^2\hat{\mathbf{k}}\otimes\hat{\mathbf{k}}+k_0^2\overleftrightarrow{\boldsymbol{\varepsilon}}\right)\cdot\hat{\mathbf{e}}_b=k_{b+}^2\hat{\mathbf{e}}_b \tag{3.1}$$

Combining both of these results yields a full picture of the wave modes of the medium in the $\hat{\mathbf{k}}$ direction. To enumerate all modes would require performing this calculation over all directions in the half space (since $k_{(a,b)+}=-k_{(a,b)-}$).

3.2 The Laboratory and the Solver Frames

As we have seen in Section 2.3, the quartic equation for wave transmission or reflection at an interface greatly simplifies given an appropriate coordinate system, which we refer to as the solver frame. This coordinate system, however, is not necessarily conducive to setting up the problem of interest. As an example, it might be the case that a user finds it easier to specify all waves and surfaces of interest in the coordinates of the principal axes of the dielectric medium.

For any calculation involving an interface, it is therefore convenient to express the problem in any orthonormal basis of the user's choosing. We refer to this basis as the laboratory frame. Our job now is to identify the coordinate frame in which the calculation simplifies, called the solver frame,

and transform the calculation inputs into this frame. As was detailed in Section 2.3, the solver frame has one of its axes in the direction parallel to the surface normal; by convention we label this the z'-direction. Further, in the solver frame, the tangential component of $\mathbf{k}$ given by:

$$\mathbf{k}_\perp = \mathbf{k} - (\mathbf{k}\cdot\hat{\mathbf{s}})\,\hat{\mathbf{s}} \tag{3.2}$$

fixes the x'-direction. The y'-direction is then determined by the cross product of the two other basis vectors, and so summarily:

$$\begin{aligned}\hat{\mathbf{x}}' &= \hat{\mathbf{k}}_\perp \\ \hat{\mathbf{y}}' &= \hat{\mathbf{z}}' \times \hat{\mathbf{x}}' \\ \hat{\mathbf{z}}' &= \hat{\mathbf{s}}\end{aligned} \tag{3.3}$$

In the event that $\mathbf{k}_\perp = 0$, the $\hat{\mathbf{x}}'$ vector can be chosen by setting:

$$\begin{aligned}x_i{}' &= \left[1 + \left(\tfrac{z_i{}'}{z_j{}'}\right)^2\right]^{-1/2} \\ x_j{}' &= -\tfrac{z_i{}'}{z_j{}'}\left[1 + \left(\tfrac{z_i{}'}{z_j{}'}\right)^2\right]^{-1/2} \\ x_k{}' &= 0\end{aligned} \tag{3.4}$$

where $i \neq j \neq k$ and $z_i{}'$, $z_j{}'$ are two non-zero components of $\hat{\mathbf{z}}'$, expressed in the laboratory coordinates, or:

$$x_j{}' = 1 \qquad i \neq j \tag{3.5}$$

if only $z_i{}'$ is non-zero. The $\hat{\mathbf{y}}'$ representation can then be found the same way as in (3.3). We can then define a matrix whose columns are the solver basis vectors expressed in the laboratory coordinates, $\mathbf{U} = [\hat{\mathbf{x}}'\ \hat{\mathbf{y}}'\ \hat{\mathbf{z}}']$. To change coordinates from the laboratory frame to the solver frame is now straightforward; any vector, $\mathbf{v}$, or tensor, $\mathbf{Q}$, expressed in the laboratory frame will transform as:

$$\mathbf{v}' = \mathbf{U}^{-1}\mathbf{v} \qquad\qquad \mathbf{Q}' = \mathbf{U}^{-1}\mathbf{Q}\mathbf{U} \tag{3.6}$$

Here we should be careful: whereas the prime notation on $\hat{\mathbf{x}}'$, $\hat{\mathbf{y}}'$, and $\hat{\mathbf{z}}'$ was to denote the solver basis vectors expressed in the laboratory coordinates, the prime notation on $\mathbf{v}'$ and $\mathbf{Q}'$ denote the representation of $\mathbf{v}$ and $\mathbf{Q}$ in the solver coordinates. Once we have transformed the problem into the solver coordinates, we can calculate all quantities of interest, and then transform the solution back into the laboratory frame by inverting (3.6).

3.3 Reflection and Transmission at an Interface

We begin with the incident wave, defined in the laboratory coordinates by its wave vector $\mathbf{k}_i$ and electric field amplitude vector $\mathbf{E}_i$. The wave is propagating in a medium defined by the dielectric tensor $\overleftrightarrow{\boldsymbol{\varepsilon}}_1$, is incident on a boundary with unit normal $\hat{\mathbf{s}}$, beyond which is a second medium defined by $\overleftrightarrow{\boldsymbol{\varepsilon}}_2$. By the process described in Section 3.2, determine the coordinate transformation matrix $\mathbf{U}$ and the solver-frame incident wave vector and dielectric tensors:

$$\mathbf{k}_i{}' = \mathbf{U}^{-1}\mathbf{k}_i \tag{3.7a}$$

$$\overleftrightarrow{\boldsymbol{\varepsilon}}_{(1,2)}{}' = \mathbf{U}^{-1}\overleftrightarrow{\boldsymbol{\varepsilon}}_{(1,2)}\mathbf{U} \tag{3.7b}$$

With this incident wave vector solve (2.24) in both medium 1 and medium 2. For solutions in medium 1, keep the two which are traveling in the negative z-direction; for solutions in medium 2, keep the those traveling in the positive z-direction: these solutions constitute $\mathbf{k}_\alpha{}'$, $\mathbf{k}_\beta{}'$, $\mathbf{k}_\gamma{}'$, and $\mathbf{k}_\nu{}'$ respectively.

Still in the solver frame, solve the eigenvector equations given by (2.26) to obtain $\hat{\boldsymbol{\alpha}}'$, $\hat{\boldsymbol{\beta}}'$, $\hat{\boldsymbol{\gamma}}'$, and $\hat{\boldsymbol{\nu}}'$. We then use (2.34) to calculate the reflection and transmission coefficients: R_α, R_β, T_γ, and T_ν. Note that these are independent of our coordinate choice. In the solver frame, the reflected and transmitted fields are:

$$\mathbf{E}_r{}' = R_\alpha \hat{\boldsymbol{\alpha}}' e^{i\mathbf{k}_\alpha{}'\cdot\mathbf{r}'} + R_\beta \hat{\boldsymbol{\beta}}' e^{i\mathbf{k}_\beta{}'\cdot\mathbf{r}'} \tag{3.8a}$$

$$\mathbf{E}_t{}' = T_\gamma \hat{\boldsymbol{\gamma}}' e^{i\mathbf{k}_\gamma{}'\cdot\mathbf{r}'} + T_\nu \hat{\boldsymbol{\nu}}' e^{i\mathbf{k}_\nu{}'\cdot\mathbf{r}'} \tag{3.8b}$$

To return to the laboratory frame, we simply apply the transformation matrix to our solutions:

$$\boldsymbol{\alpha} = R_\alpha \mathbf{U}\hat{\boldsymbol{\alpha}}' \qquad \boldsymbol{\beta} = R_\beta \mathbf{U}\hat{\boldsymbol{\beta}}' \qquad \boldsymbol{\gamma} = T_\gamma \mathbf{U}\hat{\boldsymbol{\gamma}}' \qquad \boldsymbol{\nu} = T_\nu \mathbf{U}\hat{\boldsymbol{\nu}}' \tag{3.9}$$

3.4 Optical Coatings

Many (if not most) optical components are not bare materials, but rather are specially coated so as to have desired transmission and reflection properties. In practice, these coatings are thin parallel layers of other dielectric materials, each with their own unique optical properties. However, we do not wish to turn a single interface interaction into a computation over many interfaces. Instead, we approximate that the effect of the coating layers is only to artificially change the reflection and transmission coefficients from (2.34). Explicitly: the modes excited in the reflection and transmission media remain unchanged in the presence of an optical coating, only the degree to which they are excited is affected. For isotropic coating materials, this approximation is usually appropriate.

We consider two cases: perfectly transmissive coatings, called antireflective, and perfectly reflective coatings, called highly reflective. In either case the prescription is to first calculate the reflected and transmitted waves in the absence of a coating as is done in Section 3.3. Knowing the time-averaged energy density of the incident wave:

$$\langle u_i \rangle = \frac{1}{2}\mathbf{E}_i^* \cdot \mathbf{D}_i \tag{3.10}$$

we then adjust the reflection and transmission coefficients to simultaneously conserve energy and replicate the desired coating effect.

3.4.1 Antireflective Coatings

For perfectly transmissive coatings, the reflected waves should have zero amplitude, and so we artificially set:

$$R_\alpha \to 0 \qquad R_\beta \to 0 \tag{3.11}$$

All of the energy carried into the interface by the incident field is now carried out by the two transmitted fields. The energy density of these two fields together is given as:

$$\langle u_\gamma \rangle + \langle u_\nu \rangle = \frac{1}{2}\left[|T_\gamma|^2 \hat{\boldsymbol{\gamma}}^* \cdot \left(\overleftrightarrow{\boldsymbol{\varepsilon}}_2 \cdot \hat{\boldsymbol{\gamma}}\right) + |T_\nu|^2 \hat{\boldsymbol{\nu}}^* \cdot \left(\overleftrightarrow{\boldsymbol{\varepsilon}}_2 \cdot \hat{\boldsymbol{\nu}}\right)\right] \tag{3.12}$$

and so to set (3.12) equal to $\langle u_i \rangle$, we make the following substitutions for the transmission coefficients:

$$T_\gamma \to T_\gamma \sqrt{\frac{\langle u_i \rangle}{\langle u_\gamma \rangle + \langle u_\nu \rangle}} \qquad T_\nu \to T_\nu \sqrt{\frac{\langle u_i \rangle}{\langle u_\gamma \rangle + \langle u_\nu \rangle}} \tag{3.13}$$

These new coefficients guarantee that energy is conserved, and are representative for the case of a perfectly transmissive boundary.

3.4.2 Highly Reflective Coatings

The procedure for the perfectly reflective case is substantively identical to that of Section 3.4.1. We begin by setting the transmission coefficients to zero:

$$T_\gamma \to 0 \qquad T_\nu \to 0 \tag{3.14}$$

All of the energy carried into the interface by the incident wave is now carried away by the reflected waves. The combined energy density of these wave is:

$$\langle u_\alpha \rangle + \langle u_\beta \rangle = \frac{1}{2}\left[|R_\alpha|^2 \, \hat{\boldsymbol{\alpha}}^* \cdot \left(\overleftrightarrow{\boldsymbol{\varepsilon}}_1 \cdot \hat{\boldsymbol{\alpha}} \right) + |R_\beta|^2 \, \hat{\boldsymbol{\beta}}^* \cdot \left(\overleftrightarrow{\boldsymbol{\varepsilon}}_1 \cdot \hat{\boldsymbol{\beta}} \right) \right] \tag{3.15}$$

and thus to conserve energy, we make the following substitutions for the reflection coefficients:

$$R_\alpha \to R_\alpha \sqrt{\frac{\langle u_i \rangle}{\langle u_\alpha \rangle + \langle u_\beta \rangle}} \qquad R_\beta \to R_\beta \sqrt{\frac{\langle u_i \rangle}{\langle u_\alpha \rangle + \langle u_\beta \rangle}} \tag{3.16}$$

Chapter 4

WaveCalc Objects

WaveCalc objects are defined in the `classes.py` module, however upon import of `wavecalc` the `__init__.py` module will alias this class module as `obj`. There are three distinct classes of WaveCalc object: waves, surfaces, and media. In all cases, these objects can be instantiated without any arguments, though doing so will assign default attributes. If the user would prefer to instantiate an object with an attribute set to `None`, they may set the corresponding attribute variable in the object function to `False`:

```
>> example_object = wc.obj.example_class(example_attribute = False)
>> example_object.example_attribute  is  None
True
```

where the reader will remember that we have chosen to alias the `wavecalc` module as `wc` upon import. If one wishes to instantiate an object with all attributes set to `None`, they can make use of the `everything` option:

```
>> example_object = wc.obj.example_class(everything = False)
>> example_object.example_attribute  is  None
True
```

It is also possible to randomize most attributes at instantiation by setting the attribute variable to `'random'`:

```
>> example_object = wc.obj.example_class(example_attribute = 'random')
>> example_object.example_attribute
0.6363743757927883
```

This will instantiate the object with every element of the named attribute being assigned a number from the interval $[0, 1)$ according to a uniform distribution. Note that the `everything` option cannot be used to assign all attributes randomly; at present the only choice for this option that carries functional meaning is `False`.

4.1 WaveCalc Waves

The class of objects for which WaveCalc is an eponym are called waves. These are intended to serve as representations of physical electromagnetic plane waves in any calculation. WaveCalc waves are instantiated by calling the `wave` function:

```
wave(kvec=None, efield=None, medium=None, pol=None, amp=None, everything=None)
```

For our example, we will instantiate a wave simply named `wav`:

```
>> wav = wc.obj.wave()
```

We have chosen to instantiate the wave without setting any of its attributes, and so they have taken to their defaults. The `kvec` attribute is a `numpy.ndarray` intended to represent the wave vector, which provides information about the phase evolution of a corresponding physical wave. It defaults to a unit vector in the z-direction:

```
>> wav.kvec.T, type(wav.kvec), numpy.shape(wav.kvec)
(array([[0., 0., 1.]]), numpy.ndarray, (3, 1))
```

The natural units for `kvec` are those in which $k_0 = 1$. With this convention, the length of `kvec` is equivalent to the effective index of refraction that the wave experiences.

The `efield` attribute, like `kvec`, is a `numpy.ndarray` and is intended to represent the electric field amplitude vector of the associated physical wave. It defaults to a unit vector in the x-direction:

```
>> wav.efield.T, type(wav.efield), numpy.shape(wav.efield)
(array([1., 0., 0.]), numpy.ndarray, (3, 1))
```

The `medium` attribute is a `numpy.ndarray` intended to represent the permittivity tensor of the medium in which the wave is propagating. It defaults to the 3×3 identity matrix:

```
>> wav.medium, type(wav.medium), numpy.shape(wav.medium)
(array([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]]), numpy.ndarray, (3, 3))
```

As will be generally true for all WaveCalc objects, attributes may be added or changed after instantiation through whole replacement:

```
>> wav.efield = numpy.array([[1-1j,1+1j,0]]).T
>> wav.efield.T
array([[1.-1.j, 1.+1.j, 0.+0.j]])
```

or through part-wise replacement:

```
>> wav.kvec[0,0] = 1
>> wav.kvec
array([[1, 0, 1]])
```

However, take note that the default attributes never have complex datatypes. Attempting to part-wise replace a default attribute with a complex datatype will result in an error:

```
>> wav.medium[0,1] = 1j
TypeError: can't convert complex to int
```

instead, the replacement must be done as a whole replacement:

```
>> wav.medium = numpy.array([[1,1j,0],[1j,1,0],[0,0,1]])
>> wav.medium
array([[1., 1.j, 0.],[0., 1., 0.],[0., 0., 1.]])
```

Once the wave attribute has been converted to a complex datatype in this manner, part-wise replacement with other complex datatypes will work:

```
>> wav.medium[1,0] = -1j
>> wav.medium
array([[1., 1.j, 0.],[-1.j, 1., 0.],[0., 0., 1.]])
```

One should be cautious when changing the `medium` attribute of a wave: there is no guarantee that the `kvec` and `efield` will correspond to an optical mode of the medium. Thus, in general changing the `medium` attribute will result in a wave object that is not physical. As will be addressed in Section 5.1.4, this can be rectified somewhat with the `fixmode` method.

4.2 WaveCalc Surfaces

To examine wave behavior at interfaces, it is necessary to define a surface of interaction. The WaveCalc surface objects are representations of said material surfaces. These surface objects are instantiated by calling the `surface` function:

```
surface(normal=None, into=None, out=None, coat=None, everything=None)
```

For our example, we instantiate a surface named `surf`:

```
>> surf = wc.obj.surface()
```

The WaveCalc surface has one `(3,1) numpy.ndarray` object associated to it called the `normal` attribute. This is the normal vector to the physical surface which the surface object represents. Like `kvec` for wave objects, it defaults to a unit vector in the z-direction:

```
>> surf.normal
array([0., 0., 1.])
```

The other two WaveCalc attributes of surface objects are called `out` and `into`. These are the dielectric tensors associated with the medium that the surface normal points out of and into respectively, given as `(3, 3) numpy.ndarray` objects. Both of these attributes default to the identity:

```
>> surf.out
array([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]])

>> surf.into
array([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]])

>> type(surf.out)
numpy.ndarray

>> numpy.shape(surf.into)
(3, 3)
```

4.3 WaveCalc Media

At present, WaveCalc media are the simplest objects, in a sense, having only one WaveCalc specific attribute. These objects are intended to serve as representations of the physical media in calculations. WaveCalc media are instantiated by calling the `medium` function:

```
medium(epx=None, epy=None, epz=None, ep_all=None, epsilon=None, everything=None)
```

For our example, we instantiate a medium named `med`:

```
>> med = wc.obj.medium()
```

The WaveCalc attribute associated to a medium is its permittivity tensor, called `epsilon`. Like the other medium-associated attributes of wave and surfaces, the `epsilon` attribute is a `numpy.ndarray`, and defaults to the identity:

```
>> med.epsilon, type(med.epsilon), numpy.shape(med.epsilon)
(array([[1., 0., 0.],[0., 1., 0.],[0., 0., 1.]]), numpy.ndarray, (3, 3))
```

We can assign values for the diagonals of the permittivity tensor upon instantiation of a WaveCalc medium object by use of the `epx`, `epy`, and `epz` options. As an example:

```
>> med = wc.obj.medium(2.25, 2.56, 2.89)
>> med.epsilon
array([[2.25, 0., 0.],[0., 2.56, 0.],[0., 0., 2.89]])
```

Take note, however, that a medium object has no attributes called `epx`, `epy`, and `epz`:

```
>> med.epx
<bound method medium.epx of <wavecalc.classes.medium object>>
```

The output above tells us that `epx` is a method for the WaveCalc media class, as will be discussed in further detail in Section 5.1. In the event that we want to instantiate an isotropic medium, we can make use of the `ep_all` option:

```
>> med = wc.obj.medium(ep_all = 2.25)
>> med.epsilon
array([[2.25, 0., 0.],[0., 2.25, 0.],[0., 0., 2.25]])
```

Chapter 5

WaveCalc Methods

Each of the WaveCalc classes have methods which allow the user to query and change an object instance, or create new object instances altogether. These methods include those with unique names that are specific to each class and called in the usual way:

```
object.method()
```

but WaveCalc has been designed also to support certain built-in Python methods when they might have a useful and intuitive meaning. Section 5.1 discusses the former kind of method, while Section 5.2 the latter.

As a word to the reader, this chapter is considerably nonlinear. Certain examples used to explain the workings of one method will call on other methods that have yet to be explained themselves. As a consequence, it is probably best to treat this chapter as an index rather than a series of consecutive sections.

5.1 WaveCalc Object Methods

In this section we will introduce the methods which are unique to WaveCalc objects. These methods are all called in the orthodox way, and will either return information about the objects they are called on, or will change the properties of these objects.

5.1.1 `amp`

The `amp` method operates on WaveCalc wave objects. Its syntax is:

```
amp(self, amplitude=None, rel=None)
```

When called on a wave object with no other argument, it simply returns the (real) amplitude of the electric field. If the `amplitude` argument is specified as an `int` or a `float`, the method returns nothing, but instead scales the electric field of the wave to have the corresponding amplitude:

```
>> wav = wc.obj.wave()
>> wav.efield.T, wav.amp()
(array([[1., 0., 0.]]), 1.0)

>> wav.amp(5)
>> wav.efield.T, wav.amp()
(array([[5., 0., 0.]]), 5.0)
```

If the user wishes to scale the field amplitude by some factor, they can make use of the `rel` option:

```
>> wav.amp(2, rel = True)
>> wav.amp()
10.0
```

5.1.2 `clean`

The `clean` method operates on any WaveCalc object. Its syntax is:

```
clean(self, tol=None)
```

When called on a WaveCalc object, it will drop

5.1.3 `epx`, `epy`, `epz`, `ep_all`

The `epx`, `epy`, `epz`, and `ep_all` methods act on WaveCalc medium objects. All share a similar syntax:

```
epx(self, epsilon_xx=None)
```

When given an argument as either a `int`, `float`, or `complex`, these methods act similarly to their counterpart arguments of the `wavecalc.classes.medium()` instantiator:

```
>> med_1 = wc.obj.medium(epy = 2.25)
>> med_2 = wc.obj.medium()
>> med_2.epy(2.25)
>> med_1 == med_2
True
```

These methods have an advantage over part-wise replacement of the `epsilon` attribute, as they are able to reconcile type clashes:

```
>> med = wc.obj.medium()
>> med.epsilon[2,2] = 2.5+0.3j
TypeError: can't convert complex to float

>> med.epz(2.5+0.3j)
>> med.epsilon[2,2]
(2.5+0.3j)
```

However, their use is more limited than part-wise replacement, as they only act on the diagonal elements `epsilon`.

If given no argument, the methods simply return the value of their corresponding diagonal element, or a list of all three:

```
>> med = wc.obj.medium(epsilon='random')
>> med.epx(), med.epy(), med.epz(), med.ep_all()
(0.9756212421230921,0.28323204413139935,0.7000372176721711,
 [0.9756212421230921, 0.28323204413139935,0.7000372176721711])
```

The reader should keep in mind that the diagonal terms of the dielectric tensor are (in general) coordinate dependent, and therefore may not be illuminating.

5.1.4 fixmode

The fixmode method acts on WaveCalc wave objects. Its syntax is:

```
fixmode(self, ab=None, k0=None, conserve=None, verbose=None)
```

The motivation behind fixmode is that, as we have seen in Section 2.1, only certain plane wave modes are allowed in a medium. In general, if a wave object with well-defined kvec and efield attributes is placed into a new medium, the wave object will no longer represent a physically realizable wave (ignoring that plane waves are not technically realizable). To rectify this, fixmode searches for allowed wave vectors parallel to kvec, and modifies the attribute to give a new wave vector that is permitted by the physics:

```
>> wav = wc.obj.wave()
>> med = wc.obj.medium(ep_all = 2.25)
>> wav = wav + med
>> wav.kvec.T
array([[0., 0., 1.]])

>> wav.fixmode()
>> wav.kvec
array([[0. 0., 1.5]])
```

The optional argument ab allows the user to choose whether the a- or b- mode of the medium is selected as the new wave mode, by setting ab to 0 or 1 respectively. If unspecified or improperly specified, ab will default to 0. As in many other methods and functions, the k0 option allows for rescaling of the wave vector, which the user might find occasionally useful.

The fixmode method also changes the efield attribute of a wave to reconcile it with the new kvec. In the case where the Maxwell wave operator has degenerate eigenvalues, fixmode chooses the eigenvector which has the greatest absolute conjugate dot product with the original efield. Because the calculation is done in the frame where kvec is in the positive z-direction and kvec+efield lies in the x-z plane, polarization information will be intuitively retained when appropriate:

```
>> wav = wc.obj.wave()
>> wav.rotate(45, 'y')
>> med = wc.obj.medium(ep_all = 2.25)
>> wav = wav + med
>> wav.amp(5)
>> wav.k(), wav.pol().T
(1.0, array([[0.70710678, 0., -0.70710678]]))

>> wav.fixmode()
>> wav.k(), wav.pol().T
(1.5, array([[0.70710678, 0., -0.70710678]]))
```

Note that all amplitude information about the previously unrectified wave has been lost:

```
>> wav.amp()
1.0
```

Often it is not the case that the amplitude of the uncorrected wave should have any physically meaningful relationship to the fixed wave. However, in some cases, the user might find it useful to augment the amplitude in a way that conserves energy of the wave. This can be accomplished with the conserve option:

```
>> wav.amp(5)
>> wav.fixmode(conserve = True)
>> wav.amp()
5.0
```

5.1.5 index

The index method operates on WaveCalc wave objects. Its syntax is

```
index(self, k0=None)
```

When called, this method returns the real and complex indices of refraction which the wave object experiences:

```
>> wav = wc.obj.wave()
>> med = wc.obj.medium(2.25, 2.56, 2.89)
>> wav.rotate(25, 'y')
>> wav.rotate(54, 'x')
>> wav.rotate(72, 'z')
>> wav_1 = wav + med
>> wav_1.fixmode()
>> wav_1.index()
[1.592068847339228, 0.0]
```

The k0 option rescales the vacuum wave number, and allows for physical results to remain independent of unit choice:

```
>> wav_2 = wav + med
>> wav_2.fixmode(k0=3)
>> wav_2.index(k0=3)
[1.592068847339228, 0.0]
```

5.1.6 k

The k method operates on WaveCalc wave objects. Its syntax is:

```
k(self, k0=None)
```

This method is analogous to the amp method for wave vectors, in that it returns the amplitude of the wave vector (wave number), with the exception that it cannot be used to modify the wave number:

```
>> wav = wc.obj.wave()
>> med = wc.obj.medium(ep_all = 2.25)
>> wav = wav + med
>> wav.fixmode()
>> wav.k()
1.5
```

5.1.7 pol

The pol method operates on WaveCalc wave objects. Its syntax is:

```
pol(self, polar=None)
```

When called on a wave object with no other argument, it simply returns the unit electric field amplitude vector. If the polar argument is specified as an (3, 1) numpy.ndarray, the method returns nothing, but instead changes the polarization of the electric field while maintaining its amplitude:

```
>> wav = wc.obj.wave()
>> new_pol = numpy.array([[1, 1, 0]]).T
>> wav.pol(new_pol)
>> wav.efield.T
array([[0.70710678, 0.70710678, 0.]])
```

Note that the new amplitude might be subject to numerical error:

```
>> wav.amp()
0.9999999999999999
```

5.1.8 poynting

The poynting method acts on WaveCalc waves. Its syntax is:

```
poynting(self, scale=None, norm=None)
```

This method returns the time averaged Abraham form Poynting vector $\langle \mathbf{S} \rangle = \frac{1}{2}\mathbf{E} \times \mathbf{H}^*$ as a (3, 1) numpy.ndarray. The scale option is meant to set the quantity $\sqrt{\frac{\varepsilon_0}{\mu_0}}\frac{1}{k_0}$, and thus can be used to specify the unit system of the output. As an example, we can calculate the Poynting vector of a wave traveling in a biaxial medium:

```
>> wav = wc.obj.wave()
>> med = wc.obj.medium(2.25,2.56,2.89)
>> wav = wav + med
>> wav.fixmode()
>> wav.clean()
>> wav.poynting().T
array([[0., 0., 0.75]])
```

This method is useful to demonstrate the fact that waves in anisotropic media generally transport energy in a direction not quite parallel to their wave vector:

```
>> wav.rotate(25, 'y')
>> wav.rotate(54, 'x')
>> wav.rotate(72, 'z')
>> wav.fixmode()
>> wav.clean()
>> (wav.poynting()/wav.kvec).T
array([[0.48819196, 0.63277963, 0.50946073]])
```

5.1.9 rotate

The rotate method operates on all WaveCalc objects. Its syntax is:

```
rotate(self, ang, axis, medmove=None, fix=None, verbose=None)
```

When called, the rotate method will rotate a WaveCalc object by an angle ang in degrees about axis, specified as either 'x', 'y', or 'z', in accordance with the transformation properties of its attributes:

```
>> wav = wc.obj.wave()
>> wav.rotate(45, 'y')
>> wav.kvec.T, wav.efield.T
(array([[0.70710678, 0., 0.70710678]]), array([[0.70710678, 0., -0.70710678]]))
```

The medmove option allows the user to determine whether the dielectric tensors associated with the WaveCalc object are rotated: by default, they are only rotated for WaveCalc medium objects. Setting medmove to 'with' will rotate all dielectric tensors associated to the object:

```
>> surf = wc.obj.surface()
>> med = wc.obj.medium(2.25, 2.56, 2.89)
>> surf_1 = med + surf + med
>> surf_1.rotate(90, 'x', medmove = 'with')
>> surf_1.clean()
>> surf_1.out, surf_1.into
(array([[ 2.25, 0., 0.], [0., 2.89, -0.], [0., -0., 2.56]]),
 array([[2.25, 0., 0.], [0., 2.89, -0.], [0., -0., 2.56]]))
```

For surfaces, setting medmove to 'out' or 'into' allows the user to specify that only the out or into attributes should be rotated:

```
>> surf_2 = med + surf + med
>> surf_2.rotate(90, 'x', medmove = 'into')
>> surf_2.clean()
>> surf_2.out, surf_2.into
(array([[ 2.25, 0., 0.], [0., 2.56, 0.], [0., 0., 2.89]]),
 array([[2.25, 0., 0.], [0., 2.89, -0.], [0., -0., 2.56]]))
```

In the event that the user only wants to rotate attributes associated with physical media, medmove can be set to 'only'. For a wave object, this leaves the kvec and efield attributes untouched, but rotates the medium attribute:

```
>> wav = wc.obj.wave()
>> wav = wav + med
>> wav.rotate(90, 'x', medmove = 'only')
>> wav.kvec.T, wav.efield.T
(array([[0., 0., 1.]]), array([[1., 0., 0.]]))

>> wav.medium
array([[2.25, 0., 0.], [0., 2.89, -0.], [0., -0., 2.56]])
```

For surfaces, 'only' will rotate both the out and into attributes. To specify that only one should be rotated, medmove can be set to either 'onlyout' or 'onlyinto'. For medium objects, medmove has no meaning and is ignored:

```
>> med.rotate(90, 'x', medmove = 'with')
For wavecalc media, 'medmove' option has no meaning.
Variable 'medmove' will be set to None for following calculation.
```

When acting on wave objects, the user can make use of the `fix` option. Setting this option to `True` will automatically execute `fixmode` on the wave, with the `conserve` option set to `True`. This is especially useful in isotropic media, where the user might wish to rotate the wave without otherwise changing any of its physical properties:

```
>> wav = wc.obj.wave()
>> med = wc.obj.medium(ep_all = 2.25)
>> wav = wav + med
>> wav.amp(2)
>> wav.rotate(45, 'x', fix = True)
>> wav.amp(), wav.k()
(2.0, 1.5)
```

The user should beware: in many cases a calculation could involve many rotations of a single wave object. The `fixmode` method, and therefore the `rotate` method with a true `fix` option, attempts to maintain the polarization of the rotated wave in a meaningful way, however this can be subject to numerical errors which will compound with each successive rotation. Consider a two waves, one that is rotated once through 45 degrees, and one that is rotated 180 times through 0.25 degrees. In both cases, the waves are rotated through 45 degrees, however the `wavecalc` outputs are not identical:

```
>> wav_1 = wc.obj.wave()
>> wav_2 = wc.obj.wave()
>> wav_1.pol().T, wav_2.pol().T
(array([[1., 0., 0.]]), array([[1., 0., 0.]]))

>> wav_1.rotate(45, 'x', fix = True)
>> for _ in range(180):
>>     wav_2.rotate(.25, 'x', fix = True)
>> wav_1.pol().T, wav_2.pol().T
(array([[1., 0., 0.]]), array([[0.78114334, 0.44148334, 0.44148334]]))
```

The polarization of the wave which was rotated 180 times is significantly different than its initial polarization. Why errors tend to compound more quickly in the polarization of the wave than in the wave vector is still not entirely understood. It is possible that this could be ameliorated in some way, but in the meantime it is best to avoid writing code that results in many rotations of the same wave object if the polarization is critical to the calculation.

5.2 Overloaded Python Methods

As discussed in the introduction of this chapter, WaveCalc has adapted several of the built-in Python methods for use with WaveCalc objects. This practice is commonly referred to as overloading the methods. These methods have the advantage of being syntactically concise and familiar. For example, the `__add__` method can be called by placing the + symbol between the two arguments of the method.

Some of these methods are unary, and thus act on a single object instance from a single class, while others are binary, acting as both intra- and interclass operations. In contrast to the methods introduced in Section 5.1, none of the overloaded methods modify their arguments. Instead, they

return entirely new objects. The overloaded methods also can be used as a shorthand for certain WaveCalc functions, though that discussion will be delayed until Chapter 6.

As a word of caution: though the use of certain built-in methods may be intuitive, their algebraic properties are not necessarily so. For example, no kind of addition or subtraction over WaveCalc objects will constitute a group operator. In Chapter 8, we will formally describe the addition and subtraction operators on the spaces of waves, surfaces, and media. More immediately, however, we will introduce the methods through examples.

5.2.1 `__add__` and `__sub__`

WaveCalc waves, surfaces, and media can all interact within and between one another with the addition and subtraction operations. The full picture is quite complicated, and so in this section we will only address three categories of these operations: media on waves, media on media, and media on surfaces.

A WaveCalc medium object can be added to a wave object to return a new wave object:

```
>> wav = wc.obj.wave()
>> med = wc.obj.medium(2.25, 2.56, 2.89)
>> wav = wav + med
>> type(wav)
wavecalc.classes.wav
```

The new wave will be a clone of the previous wave, except with the `medium` attribute changed for the `epsilon` attribute of the medium object:

```
>> wav.medium
array([[2.25, 0., 0.], [0., 2.56, 0.], [0., 0., 2.89]])
```

This can be a convenient way for changing the `medium` attribute of a wave, as it is generally faster and easier to read than part-wise or whole replacement. When adding waves and media, the order does not matter:

```
>> wav = wc.obj.wave()
>> wav = med + wav
>> wav.medium
array([[2.25, 0., 0.], [0., 2.56, 0.], [0., 0., 2.89]])
```

In either case, the output is identical. Currently, there is no `__sub__` method that operates on waves and media together.

Two WaveCalc media objects can be added together to return a surface object:

```
>> med_1 = wc.obj.medium('random')
>> med_2 = wc.obj.medium('random')
>> add_surf = med_1 + med_2
>> type(add_surf)
wavecalc.classes.surface
```

This new surface has a normal in the z-direction:

```
>> add_surf.normal
array([0., 0., 1.])
```

and is coming out of the first medium and into the second:

```
>> add_surf.out is med_1.epsilon and add_surf.into is med_2.epsilon
True
```

Notice that unlike in many cases, this addition operator is not commutative: the ordering of the media matter. Typing instead `med_2 + med_1` will assign `med_2` as an `out` attribute and `med_1` as an `into`. Subtraction has been similarly defined:

```
>> sub_surf = med_1 - med_2
>> sub_surf.into is add_surf.out and sub_surf.out is add_surf.into
True
```

Under this convention, subtraction is equivalent to switching the order of addition, and vice versa.

We may also add and subtract surfaces and media to and from one another, the result being a new surface:

```
>> surf_1 = wc.obj.surface(into='random', out='random')
>> surf_2 = wc.obj.surface(into='random', out='random')
>> a = surf_1 + med_1
>> b = med_1 - surf_2
>> c = med_2 + surf_1
>> d = surf_2 - med_2
>> type(a)
wavecalc.classes.surface
```

Adding a surface on the left to a medium on the right is the same as subtracting a surface on the right from a medium on the left: in both cases, the output is a clone of the surface with the `into` attribute changed to the medium's `epsilon` attribute. Reversing the sign of the operator in either case produces the same clone except with the `out` attribute changed instead:

```
>> a.normal is surf_1.normal and a.out is surf_1.out
True

>> a.into is med_1.epsilon
True

>> a.into is b.into and c.out is d.out
True
```

5.2.2 __neg__

The __neg__ method operates on WaveCalc waves and surfaces. Its syntax is simple:

```
__neg__(self)
```

```
>> wav = wc.obj.wave()
>> wav = -wav
>> wav.efield.T, wav.kvec.T
(array([[-1., 0., 0.]]), array([[0., 0., 1.]]))
```

5.2.3 __pos__

The __pos__ method operates on all WaveCalc objects. Its syntax is:

```
__pos__(self)
```

And can be called by placing a + before the argument. Though one might intuitively expect __pos__ and __neg__ to be related in some way, please note that this is not the case. It is an unfortunate fact that Python does not allow for custom operator symbols. Instead, __pos__ can be seen as a shorthand for clean with all optional arguments set to default values:

```
>> wav = wc.obj.wave()
>> new_pol = numpy.array([[-1,1,0]]).T
>> wav.pol(new_pol)
>> wav.amp()
0.9999999999999999

>> wav_1 = +wav
>> wav_1.amp()
1.0
```

Note that __pos__ differs from clean in that it does not change any attributes of its argument:

```
>> wav.amp()
0.9999999999999999
```

When it takes a wave as its argument, __pos__ also acts as a shorthand for fixmode:

```
>> med = wc.obj.medium(2.25,2.56,2.89)
>> wav = wav + med
>> wav_2 = +wav
>> wav_2.kvec.T
array([[0., 0., 1.5]])
```

Again, it does not change the attributes of the original wave:

```
>> wav.kvec.T
array([[0., 0., 1.]])
```

5.2.4 __invert__

The __invert__ method acts on WaveCalc waves surfaces. Its syntax is:

```
__invert__(self)
```

and can be called by typing a ~ before the argument. When acting on a WaveCalc wave, __invert__ reverses the wave vector:

```
>> wav = wc.obj.wave()
>> wav_1 = ~wav
>> wav_1.kvec.T
array([[0., 0., -1.]])
```

When the argument is a surface, the action of __invert__ is to swap the out and into attributes:

```
>> surf = wc.obj.surface()
>> med = wc.obj.medium(2.25, 2.56, 2.89)
>> surf = surf + med
>> surf.out, surf.into
(array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]),
 array([[2.25, 0., 0.], [0., 2.56, 0.], [0., 0., 2.89]]))

>> surf =  ~surf
>> surf.out, surf.into
(array([[2.25, 0., 0.], [0., 2.56, 0.], [0., 0., 2.89]]),
 array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]))
```

This might be useful to a user who is looking to examine both input and output faces of an optical component.

5.2.5 __eq__

The __eq__ method acts on all WaveCalc objects. Its syntax is:

```
__eq__(self, other)
```

This method compares two objects by typing == between them, extending the notion of equivalence to WaveCalc objects. Two WaveCalc objects are equal if their attributes are all equal in the usual sense:

```
>> wav
```

Chapter 6

WaveCalc Functions

WaveCalc functions are defined in the `functions.py` module, however upon import of `wavecalc` the `__init__.py` module will alias this as `fun`. Though there are several background functions in this module that the user might find useful at times, we will focus on the four functions which are intended to be used in the foreground: `crash`, `reflect`, `transmit`, `rotate`, and `modes`. Documentation on other WaveCalc functions is included in the `functions.py` module itself.

6.1 `crash`

This function takes an input wave, given as a WaveCalc wave object, and a surface, given as a WaveCalc surface object, and outputs a list of (up to) four WaveCalc waves corresponding to the two reflected and two transmitted waves from the boundary interaction. It is called with the syntax:

```
crash(wave, surface, k0=None, coat=None, combine_same=None, verbose=None)
```

The `wave` argument is the WaveCalc wave which is to be crashed onto the `surface` argument, a WaveCalc surface. The `k0` optional argument allows the user to apply a custom scaling of the wave vectors by changing the value of k_0 from 1. The `coat` option allows the user to specify whether there is a special optical coating on the surface, either antireflective or highly reflective, by passing `'AR'` or `'HR'` as the argument; if left unspecified, it defaults to the surface coating attribute. In the event that a coating is specified, it is treated as ideal: either perfectly transmissive or perfectly reflective. When the `combine_same` option is set to `True`, output waves with the same `kvec` will be combined into one wave (by adding the electric fields together). The `verbose` option will print more information about the calculation if set to `True`.

In the following example, we create a wave propagating through vacuum in the x-z plane making a 45 degree angle with the z-axis, and crash it into a biaxial crystal that has been oriented somewhat arbitrarily with respect to the x-y surface plane:

```
>> wav = wc.obj.wave()
>> surf = wc.obj.surface()
>> med = wc.obj.medium(epx = 2.25, epy = 2.55, epz = 2.8)
>> wav.rotate(45, 'y')
>> med.rotate(-73, 'z')
>> med.rotate(68, 'x')
>> surf = surf + med
>> output = wc.fun.crash(wav, surf)
>> output
```

```
[<wavecalc.classes.wave>,
 <wavecalc.classes.wave>,
 <wavecalc.classes.wave>,
 <wavecalc.classes.wave>]
```

The output of our calculation is a list of four WaveCalc waves, the first two being the reflected waves:

```
>> output[0].clean()
>> output[1].clean()
>> output[0].kvec.T, output[1].kvec.T
(array([[0.70710678, 0., -0.70710677]]),
  array([[0.70710678+0.j, 0.+0.j, -0.7071068+0.j]]))
```

and the second two being the transmitted waves:

```
>> output[2].clean()
>> output[3].clean()
>> output[2].kvec.T, output[3].kvec.T
(array([[0.70710678, 0., 1.37016595]]),
  array([[0.70710678, 0., 1.50200523]]))
```

Both the reflected and transmitted waves are created with `medium` attributes corresponding to their appropriate media:

```
>> output[0].medium is surf.out and output[1].medium is surf.out
True

>> output[2].medium is surf.into and output[3].medium is surf.into
True

>> output[2].index(), output[3].index()
([1.5418672863970397, 0.0], [1.6601264136412388, 0.0])
```

The built-in Python matrix multiplication operator `@` has been overloaded to serve as a quick method for calling `crash`. With this method, there are only two arguments, the wave and the surface, though the order is not important:

```
>> out_1 = wav @ surf
>> out_2 = surf @ wav
>> out_1 == out_2
True
```

When the `@` operator is used, the `combine_same` option of `crash` is set to `True`

```
>> out_1
[<wavecalc.classes.wave>,
 <wavecalc.classes.wave>,
 <wavecalc.classes.wave>]
```

6.2 `reflect`

```
reflect(wav, surf, k0=None, coat=None, combine_same=None, verbose=None)
```

6.3 transmit

```
transmit(wav, surf, k0=None, coat=None, combine_same=None, verbose=None)
```

6.4 rotate

The rotation function rotates objects in three-dimensional Euclidean space. It is called with the syntax:

```
rotate(ob, ang, axis = None, medmove = None)
```

The `ob` argument is the object to be rotated, either a `(3, 1)` or `(3, 3)` `numpy.array`, or a WaveCalc object. The `ang` argument is the angle through which to rotate the object, given as an `int` or a `float`. The `axis` argument can be set to either `'x'`, `'y'`, or `'z'`, and specifies the axis of rotation. The `medmove` option is similarly defined as for the `rotate` method.

One must be mindful not to conflate the `rotate` method and the `rotate` function; while they both make use of the same basic code, their actions are distinct. As discussed previously in Section 5.1

```
>> wave_1 = wc.obj.wave()
>> wave_2 = wc.fun.rotate(wave_1, 45, 'y')
>> wave_1.kvec.T, wave_2.kvec.T
(array([[0, 0, 1]])), array([[0.70710678, 0., 0.70710678]]))

>> wave_1.rotate(45, 'y')
>> wave_1.kvec.T, wave_1.efield.T
(array([[0.70710678, 0., 0.70710678]]), array([[0.70710678, 0., -0.70710678]]))
```

6.5 modes

This function takes an input directional vector, given as a `(3, 1)` `numpy.ndarray` and a medium, given as either a WaveCalc medium object or as a `(3, 3)` `numpy.ndarray`. The output is a list of four WaveCalc waves corresponding to the allowed plane wave modes whose wave vector is parallel to the input vector. It is called with the syntax:

```
modes(vec, med, k0=None, verbose=None)
```

Chapter 7

WaveCalc Examples

In this chapter we will give examples of the utility of the `wavecalc` package. This is by no means intended to be a comprehensive survey of its ability, but instead should give the reader

To be able to follow all of these examples with a Python interpreter, it is necessary to make the following imports:

```
>> import wavecalc as wc
>> import numpy as np
>> import matplotlib.pyplot as plt
>> import copy
```

Notice that unlike in the previous chapters, we are now aliasing `numpy` as `np`. This is a commonly accepted alias in the Python community, and is done so to preserve the user's fingers. To the dismay of Python purists, we will also make use of semicolons throughout the examples in this chapter. The semicolon, `;`, allows us to separate multiple statements written on a single line, thereby reducing the code's FOOTPRINT on the printed page. As a general rule, we will only use semicolons to string together statements that are similar.

7.1 Brewster's Angle and Total Internal Reflection

Though it is certainly not necessary to use a specialized Python package to do so, `wavecalc` can be used to produce the familiar results for isotropic media. This also serves as a sanity check for the software.

In this section we consider two media: one is vacuum, and the other an isotropic material with index of refraction $n = 1.5$. As our first example computation, we would like to investigate the reflection coefficients for both s- and p-polarized light coming from the vacuum and incident on the isotropic material. To do so we can execute the following:

```
>> wav_s = wc.obj.wave(pol='x') ; wav_p = wc.obj.wave(pol='y')
>> surf = wc.obj.surface()
>> med = wc.obj.medium(ep_all=2.25)
>> surf = surf + med
>> thetas = np.linspace(0,89,90)
>> Refls_s = [] ; Refls_p = []
>> for t in thetas:
>>     wav_s.rotate(t,'x') ; wav_p.rotate(t,'x')
>>     out_s = (wav_s @ surf)[0] ; out_p = (wav_p @ surf)[0]
```

```
>>     refl_s = out_s.amp() ; refl_p = out_p.amp()
>>     Refls_s.append(refl_s) ; Refls_p.append(refl_p)
>>     wav_s.rotate(-t,'x') ; wav_p.rotate(-t,'x')
>> Refls_s = np.asarray(Refls_s) ; Refls_p = np.asarray(Refls_p)
```

This has created three arrays: the `thetas` array which is simply the set of integers from 0 to 89 inclusive, as well as `Refls_s` and `Refls_p`, which are comprised of the amplitudes of the s- and p-polarized reflections respectively. We can plot the two reflection curves with:

```
>> plt.plot(thetas,Refls_s,label='s-pol')
>> plt.plot(thetas,Refls_p,label='p-pol')
>> plt.title("n_in = 1, n_out = 1.5",fontsize=14)
>> plt.xlabel("Angle of Incidence (deg)",fontsize=14)
>> plt.ylabel("Reflection Coefficient",fontsize=14)
>> plt.legend(fontsize=12)
>> plt.show()
```

The result is shown as the left plot of Figure 7.1. As we expect, the s-polarization reflection coefficient rises monotonically, whereas the p-polarization has zero reflection at the special incidence known as Brewster's angle. In this case, Brewster's angle is $\tan^{-1}(1.5) \approx 56.3°$.

Now suppose we want to flip the situation: instead of waves coming from the vacuum incident on a material, consider waves propagating in a material and incident on a boundary with the vacuum. We can construct the analogous reflection curves by executing:

```
>> wav_s = wc.obj.wave(pol='x') ; wav_p = wc.obj.wave(pol='y')
>> surf = wc.obj.surface()
>> med = wc.obj.medium(ep_all=2.25)
>> surf = med + surf
>> wav_s = wav_s + med ; wav_p = wav_p + med
>> wav_s.fixmode() ; wav_p.fixmode()
>> thetas = np.linspace(0,89,90)
>> Refls_s = [] ; Refls_p = []
>> for t in thetas:
>>     wav_s.rotate(t,'x') ; wav_p.rotate(t,'x')
>>     out_s = (wav_s @ surf)[0] ; out_p = (wav_p @ surf)[0]
>>     refl_s = out_s.amp() ;refl_p = out_p.amp()
>>     Refls_s.append(refl_s) ; Refls_p.append(refl_p)
>>     wav_s.rotate(-t,'x') ; wav_p.rotate(-t,'x')
>> Refls_s = np.asarray(Refls_s) ; Refls_p = np.asarray(Refls_p)
```

Again, we have created three arrays with identical names and meanings as before, except now the media have been swapped. We can plot these reflection curves with:

```
>> plt.plot(thetas,Refls_s,label='s-pol')
>> plt.plot(thetas,Refls_p,label='p-pol')
>> plt.title("n_in = 1.5, n_out = 1",fontsize=14)
>> plt.xlabel("Angle of Incidence (deg)",fontsize=14)
>> plt.ylabel("Reflection Coefficient",fontsize=14)
>> plt.legend(fontsize=12)
>> plt.show()
```

The result is shown as the right plot of Figure 7.1. Again, we see a Brewster's angle dip for the p-polarization, this time at $\tan^{-1}(1/1.5) \approx 33.7°$. Additionally, we can the total internal reflection of both waves beyond a particular incidence angle; a phenomenon observed when the incident medium has a higher index than the transmission medium. For this case, the waves experience total internal reflection for incidence angles above $\sin^{-1}(1/1.5) \approx 41.8°$.

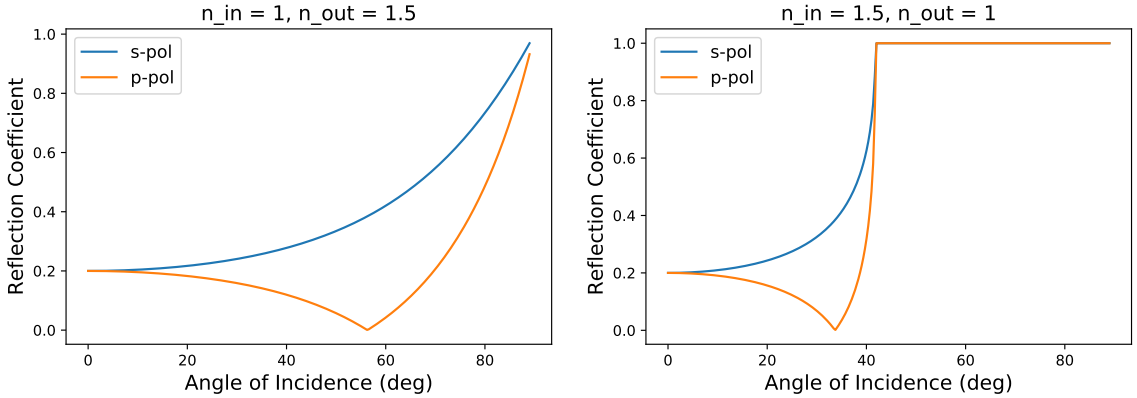

Figure 7.1: Left: The reflection amplitudes for s- and p-polarized waves from vacuum incident on an isotropic material with index $n = 1.5$. Right: The reflection amplitudes for s- and p- polarized waves from an isotropic material with index $n = 1.5$ incident on vacuum.

7.2 Reflection in Arbitrary Medium

```
>> wav = wc.obj.wave(efield=False)
>> surf = wc.obj.surface()
>> med = wc.obj.medium(epx=1.4**2,epy=1.9**2,epz=2.4**2)
>> med.rotate(45,'x') ; med.rotate(45,'y') ; med.rotate(45,'z')
>> surf = med + surf
>> wav = wav + med
>> thetas = np.linspace(0,60,61)
>> Ang_a = [] ; Ang_b = []
>> for t in thetas:
>>     wav.rotate(t,'x')
>>     wav.fixmode()
>>     out = wav@surf
>>     out_a = out[0] ; out_b = out[1]
>>     k_a = out_a.kvec ; k_b = out_b.kvec
>>     ang_a = (180/pi)*np.arctan(k_a[1,0]/k_a[2,0])
>>     ang_b = (180/pi)*np.arctan(k_b[1,0]/k_b[2,0])
>>     Ang_a.append(ang_a) ; Ang_b.append(ang_b)
>>     wav.rotate(-t,'x')
>> Ang_a = np.asarray(Ang_a) ; Ang_b = np.asarray(Ang_b)

>> plt.plot(thetas,Ang_a,label='alpha-wave')
>> plt.plot(thetas,Ang_b,label='beta-wave')
>> plt.plot(thetas,thetas,'k--')
```

```
>> plt.title('''Wave vector reflection, y-z incidence plane
>> nx=1.4, ny=1.9, nz=2.4, Rz(45)Ry(45)Rx(45)''',fontsize=14)
>> plt.xlabel("Angle of Incidence (deg)",fontsize=14)
>> plt.ylabel("Angle of Reflection (deg)",fontsize=14)
>> plt.legend(fontsize=12)
>> plt.show()
```

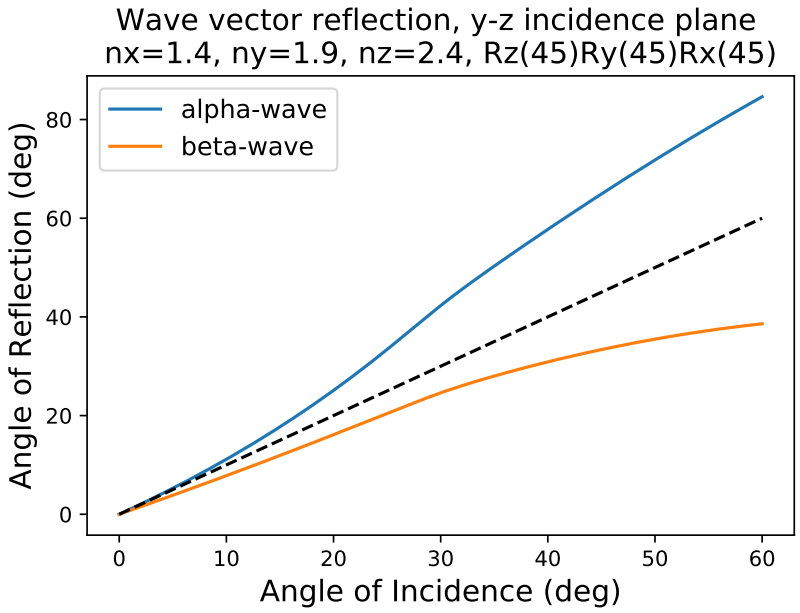

Figure 7.2:

7.3 Poynting Separation in Arbitrary Medium

```
>> wav = wc.obj.wave(efield=False)
>> surf = wc.obj.surface()
>> med = wc.obj.medium(epx=1.4**2,epy=1.9**2,epz=2.4**2)
>> med.rotate(45,'x') ; med.rotate(45,'y') ; med.rotate(45,'z')
>> surf = med + surf
>> wav = wav + med
>> wav_a = copy.deepcopy(wav) ; wav_b = copy.deepcopy(wav)
>> thetas = np.linspace(0,180,181)
>> S_k_ang_a = [] ; S_k_ang_b = []
>> for t in thetas:
>>     wav_a.rotate(t,'x') ; wav_b.rotate(t,'x')
>>     wav_a.fixmode(ab=0) ; wav_b.fixmode(ab=1)
>>     S_a = wav_a.poynting(norm=True) ; S_b = wav_b.poynting(norm=True)
>>     k_a = wav_a.kvec ; k_b = wav_b.kvec
>>     k_a = k_a/np.linalg.norm(k_a) ; k_b = k_b/np.linalg.norm(k_b)
>>     S_k_a = (S_a.T @ k_a)[0,0] ; S_k_b = (S_b.T @ k_b)[0,0]
>>     ang_a = (180/pi)*np.arccos(S_k_a).real
```

```
>>     ang_b = (180/pi)*np.arccos(S_k_b).real
>>     S_k_ang_a.append(ang_a) ; S_k_ang_b.append(ang_b)
>>     wav_a.rotate(-t,'x') ; wav_b.rotate(-t,'x')
>> S_k_ang_a = np.asarray(S_k_ang_a)
>> S_k_ang_b = np.asarray(S_k_ang_b)

>> plt.plot(thetas,S_k_ang_a,label='a-wave')
>> plt.plot(thetas,S_k_ang_b,label='b-wave')
>> plt.title('''Wave vector - Poynting Vector Separation
>> wave vector in y-z plane
>> nx=1.4, ny=1.9, nz=2.4, Rz(45)Ry(45)Rx(45)''',fontsize=14)
>> plt.xlabel("Azimuthal wave vector direction (deg)",fontsize=14)
>> plt.ylabel("Separation (deg)",fontsize=14)
>> plt.legend(fontsize=12)
>> plt.show()
```

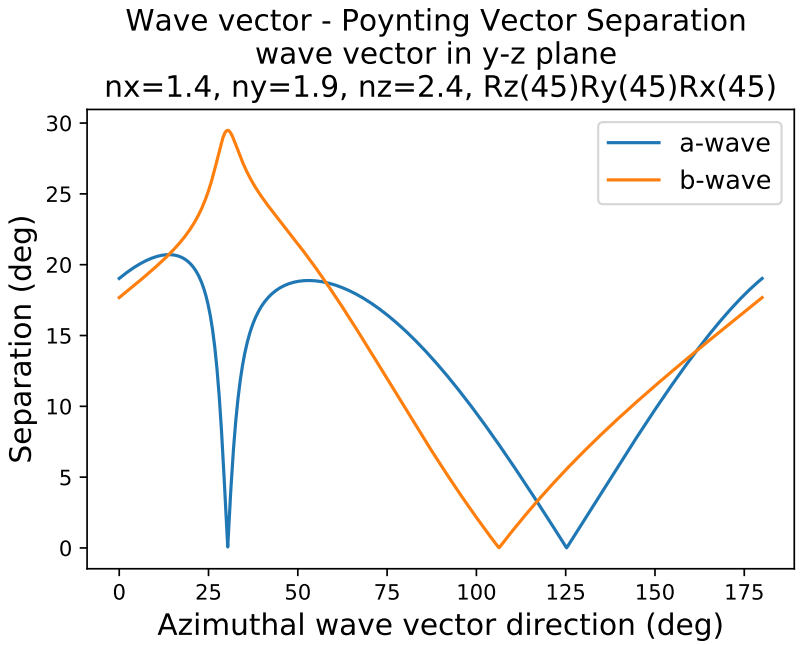

Figure 7.3:

7.4 The k-Surface of a Medium

Chapter 8

The Algebra of WaveCalc: UNDER CONSTRUCTION

If we denote the medium, surface, and wave spaces respectively as $\mathcal{M}$, $\mathcal{S}$, $\mathcal{W}$, then:
Space of 3-dimensional vectors as $\mathcal{V}$, and the space of 3×3 matrices as $\mathcal{A}$

$$\begin{array}{rcl} \mathcal{S} & = & \{\mathbf{s} = (s_1, s_2, s_3) : s_1, s_3 \in \mathcal{A}, \text{ and } s_2 \in \mathcal{V}\} \\ \mathcal{W} & = & \{\mathbf{w} = (w_1, w_2, w_3) : w_1, w_2 \in \mathcal{V}, \text{ and } w_3 \in \mathcal{A}\} \\ \mathcal{M} & = & \{\mathbf{m} = (m) : m \in \mathcal{A}\} \end{array} \tag{8.1}$$

For $\mathbf{s} = (s_1, s_2, s_3)$, s_1 is the `out` attribute, s_2 is the `normal` attribute, and s_3 is the `into` attribute. For $\mathbf{w} = (w_1, w_2, w_3)$, w_1 is the `kvec` attribute, w_2 is the `efield` attribute, and w_3 is the `medium` attribute. Finally, for $\mathbf{m} = (m)$, m is the `epsilon` attribute. In this notation, the default attribute assignment is denoted as a 1; for example, a surface with all default values would be represented as $\mathbf{s} = (1, 1, 1)$. All of the binary addition and subtractions are defined below:

$$\text{Yield a surface} \begin{cases} \mathbf{m}_a +^1 \mathbf{m}_b = (m_a, 1, m_b) \\ \mathbf{m}_a -^1 \mathbf{m}_b = (m_b, 1, m_a) \\ \mathbf{s} +^2 \mathbf{m} = (s_1, s_2, m) \\ \mathbf{m} +^3 \mathbf{s} = (m, s_2, s_3) \\ \mathbf{s} -^2 \mathbf{m} = (m, s_2, s_3) \\ \mathbf{m} -^3 \mathbf{s} = (s_1, s_2, m) \end{cases} \qquad \text{Yield a wave} \begin{cases} \mathbf{m} +^4 \mathbf{w} = (w_1, w_2, m) \\ \mathbf{w} +^5 \mathbf{m} = (w_1, w_2, m) \end{cases} \tag{8.2}$$

Where we have used superscripts to remind ourselves that each kind of addition and subtraction is a distinct mapping. From the definitions, many properties immediately follow; for example addition and subtraction over $\mathcal{M}$ are related by:

$$\mathbf{m}_a +^1 \mathbf{m}_b = \mathbf{m}_b -^1 \mathbf{m}_a \tag{8.3}$$

and similarly:

$$\mathbf{m} +^3 \mathbf{s} = \mathbf{s} -^2 \mathbf{m} \qquad \mathbf{m} -^3 \mathbf{s} = \mathbf{s} +^2 \mathbf{m} \tag{8.4}$$

These are the analogues of commutation relations in this strange new algebra. The operators over media and surfaces also have an analogue of associativity:

$$\mathbf{m}_a +^3 (\mathbf{m}_b +^1 \mathbf{m}_c) = (\mathbf{m}_a +^1 \mathbf{m}_b) +^2 \mathbf{m}_c \qquad \mathbf{m}_a -^3 (\mathbf{m}_b -^1 \mathbf{m}_c) = (\mathbf{m}_a -^1 \mathbf{m}_b) -^2 \mathbf{m}_c \tag{8.5}$$

however, one should be careful when attempting to generalize. It could be tempting to drop the superscripts and parentheses from (8.5), but take note that the order of additions matter. In particular, if we have a series of media:

$$\mathbf{m}_1 + \mathbf{m}_2 + \mathbf{m}_3 + \cdots + \mathbf{m}_{n-2} + \mathbf{m}_{n-1} + \mathbf{m}_n$$

the only meaningful ordering of operations is for the first evaluated sum to be $+^1$, and then every addition on the right side is $+^2$, and on the left side is $+^3$. A similar conclusion is true for subtraction, with $-^1$ playing the role of $+^1$, $-^2$ playing the role of $+^2$, and $-^3$ the role of $+^3$.

We will prove that this is true for the case of addition. Suppose we have a series of n media being added together, it immediately follows that there are $n-1$ addition operations in the expression of the series, which we number from left to right. Consider the first addition operation to be evaluated, we know two things about this addition: first, that it is an addition between two media, and so must be $+^1$, and second, that its order in the series, k, must satisfy $1 \leq k \leq n-1$. Suppose now, for the purpose of proof by contradiction, that at least one of three conditions hold true: 1) there is and addition operator $+^2$ with order $m < k$, 2) there is an addition operator $+^3$ with order $p > k$, or 3) there is and addition operator $+^1$ with order $q \neq k$. In the first case, we know that because $+^2$ adds surfaces on the left to media on the right, there must then be a surface object to the left of $\mathbf{m}_k$. In the second case, we know that because $+^3$ adds media on the left to surfaces on the right, there must be a surface to the right of $\mathbf{m}_{k+1}$. However, $\mathbf{m}_k +^1 \mathbf{m}_{k+1}$ is the first evaluated operation and results in a surface, and all subsequent additions with media will only result in new surfaces.

We will call these expressions well-ordered. With this knowledge, for any well ordered sums and differences of elements of $\mathcal{M}$, the following is true:

$$\mathbf{m}_1 \pm \mathbf{m}_2 \pm \mathbf{m}_3 \pm \cdots \pm \mathbf{m}_{n-2} \pm \mathbf{m}_{n-1} \pm \mathbf{m}_n = \mathbf{m}_1 \pm \mathbf{m}_n \tag{8.6}$$

$$\text{Unary operations} \begin{cases} -\mathbf{s} = (s_1, -s_2, s_3) \\ \sim \mathbf{s} = (s_3, s_2, s_1) \\ -\mathbf{w} = (-w_1, -w_2, w_3) \end{cases} \tag{8.7}$$

$$\mathbf{m} - (\sim \mathbf{s}) = \sim (\mathbf{m} + \mathbf{s}) \tag{8.8}$$

$$\mathbf{m} = (m) \in \mathcal{S} \tag{8.9}$$

$$\mathbf{w} = () \in \mathcal{W} \tag{8.10}$$

$$1 \tag{8.11}$$

$$
\begin{array}{rrcl}
 & \mathcal{M}\times\mathcal{M} & \to & \mathcal{S} \\
 & \mathcal{M}\times\mathcal{S} & \to & \mathcal{S} \\
 & \mathcal{S}\times\mathcal{M} & \to & \mathcal{S} \\
+: & \mathcal{M}\times\mathcal{W} & \to & \mathcal{W} \\
 & \mathcal{W}\times\mathcal{M} & \to & \mathcal{W} \\
 & \mathcal{S}\times\mathcal{W} & \to & \mathcal{W}\times\mathcal{W} \\
 & \mathcal{W}\times\mathcal{S} & \to & \mathcal{W}\times\mathcal{W}
\end{array}
\qquad
\begin{array}{rrcl}
 & \mathcal{M}\times\mathcal{M} & \to & \mathcal{S} \\
 & \mathcal{M}\times\mathcal{S} & \to & \mathcal{S} \\
 & \mathcal{S}\times\mathcal{M} & \to & \mathcal{S} \\
-: & \mathcal{S}\times\mathcal{W} & \to & \mathcal{W}\times\mathcal{W} \\
 & \mathcal{W}\times\mathcal{S} & \to & \mathcal{W}\times\mathcal{W} \\
 & \mathcal{S} & \to & \mathcal{S} \\
 & \mathcal{W} & \to & \mathcal{W}
\end{array}
\tag{8.12}
$$

$$\mathbf{m}_1 + \mathbf{m}_2 = (m_1, 1, m_2) \in \mathcal{S} \tag{8.13}$$

$$\mathbf{m}_1 - \mathbf{m}_2 = (m_2, 1, m_1) \in \mathcal{S} \tag{8.14}$$

$$\mathbf{m}_1 + \tag{8.15}$$

$$m_1 + m_2 + m_3 = m_1 + m_3 \tag{8.16}$$

$$m_1 - m_2 - m_3 = \tag{8.17}$$

$$o_1 + o_2 = o_2 - o_1 \tag{8.18}$$

We can best summarize the properties of addition and subtraction by considering three arbitrary WaveCalc objects, o_1, o_2, and o_3, of which at least two are media, none are waves, and no more than one is a surface. Then addition and subtraction are both associative:

$$
\begin{aligned}
o_1 + (o_2 + o_3) &= (o_1 + o_2) + o_3 \\
o_1 - (o_2 - o_3) &= (o_1 - o_2) - o_3
\end{aligned}
\tag{8.19}
$$

they are anti-commutative in the sense that:

$$o_1 + o_2 = o_2 - o_1 \tag{8.20}$$

and they are collapsible in the sense that:

$$
\begin{aligned}
o_1 + o_2 + o_3 &= o_1 + o_3 \\
o_1 - o_2 - o_3 &= o_1 - o_3
\end{aligned}
\tag{8.21}
$$

so long as o_2 is not a surface. Notice that while addition and subtraction are associative, they are

not mutually associative. To illustrate this we take the two quantities:

$$o_1 - (o_2 + o_3) \qquad\qquad (o_1 - o_2) + o_3$$

and write them in double addition form:

$$\begin{aligned} o_1 - (o_2 + o_3) &= o_2 + o_3 + o_1 \\ (o_1 - o_2) + o_3 &= o_2 + o_1 + o_3 \end{aligned} \tag{8.22}$$

As addition is not commutative, in general the objects $o_3 + o_1$ and $o_1 + o_3$ will be distinct, and so too will be the sums in (8.22).

Chapter 9

Notes on Future Development

Things that need further development in this document:

- optic axes section should be developed
- the quartic paradox needs to be explained and addressed
- how to treat AR or HR coatings theoretically should be addressed
- the overloaded unary operations need to be explained
- the algebra chapter needs to be completed
- the modes section of the implementation chapter needs to be written, even though it's boring

Things that need further development and implementation in `wavecalc`:

- handling of evanescent waves needs to be addressed
- `aux_rotate_copy` needs to only rotate attributes that are not Nonetype
- `clean` method must be revised
- `aux_remove_vacuum` needs to be developed and interfaced with `aux_waveinterf` to get rid of modes with no electric field amplitude
- chaining surfaces together and appropriately tracking waves

Things I might like to implement in the future:

- Include a unique identifier in the outputs of main functions that is only dependent on the other outputs. This would allow a quick way to verify that a change to the code has not changed the outputs of main functions. Making it actually unique would be difficult, but if the identifier is, for example, a very large number, it could be made effectively unique.
- A coating class for custom-property coatings. This would require an overhaul of the coating handling.
- Better handling of numerical errors
- Better handling of complex wave vectors
- Add a module for `wavecalc` specific exceptions

- Add some notion of physical extent and phase so that interference phenomenon can be directly investigated
- Include more documentation on how functions handle the calculations so that the user might better understand errors