Numerical Methods for Optical Interference Filters

Håkon Marthinsen

Master of Science in Physics and Mathematics
Submission date: June 2009
Supervisor: Brynjulf Owren, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

We study optical interference filters and methods for designing them. The starting point is a known model which describes the reflectance in terms of the refractive index and layer thickness of the materials used.

The work will include one or more elements of the following:
- Multiple layer model
- Model including a continuous spectrum of frequencies
- Comparison between different optimisation methods

Assignment given: 19. January 2009
Supervisor: Brynjulf Owren, MATH

Preface

This thesis is the culmination of the Master of Technology study programme at the Department of Mathematical Sciences, Norwegian University of Science and Technology (NTNU).

The idea for this thesis started out as a series of summer internships at the Norwegian Defence Research Establishment (FFI) where I worked with the theory behind optical interference filters in the department of electro-optics. I pursued this topic in the mandatory 9th semester report, where I described the physics behind filters, and formulated the problem of designing a one-layer anti-reflective filter as an optimisation problem in a Lie algebra. In this thesis, I present an approach for designing filters that avoids Lie groups and algebras. Several optimisation methods can be employed in the design process, so I also explore the performance of a selection of methods.

I would like to thank my advisor, Professor Brynjulf Owren for sharing his enthusiasm with me, and for being very patient with me during my periods of procrastination. You have given me inspiration to keep on writing. Also, a big thank you to Atle Rognmo at FFI for the best summer internships I have had. Last, but not least, to my friends and loved ones for both calming me and pushing me on when needed.

Only open-source tools were used in the work surrounding this thesis. Specifically, LaTeX was used for typesetting, Asymptote, Inkscape, Xfig and Matplotlib were used for figures, and Python with the SciPy library was used for the numerical work, everything under the Linux operating system.

Contents

Chapter 1

Introduction

In this thesis we will investigate how *optical interference filters* are designed. These filters have many real-world applications, such as anti-reflection coatings [7, Ch. 3] on camera lenses, high-reflectance mirrors [7, Ch. 5.2] used in laser cavities, and jewellery. In cameras, anti-reflection filters are employed to ensure that most of the incident visible light is transmitted through the lens, while most of the incident invisible light (ultraviolet and infrared) is reflected. This ensures that the detector chip does not erroneously register invisible light as visible. Laser mirrors need to be as reflective as possible at the wavelength of the laser, so these can be regarded as the opposite of anti-reflection filters. Sometimes, optical coatings are applied to jewellery because of their iridescent properties, which changes the colour of the jewellery depending on which angle it is viewed.

We will concentrate on design methods for *dielectric* anti-reflection filters. Dielectric filters contain only materials that are transparent, i.e. materials that do not absorb light. Most anti-reflection filters are dielectric, since we want as much light as possible to pass through in a chosen wavelength interval.

Chapter 2

Optical Interference Filters

This section follows closely the corresponding section of my previous report [8], but with a few modifications and corrections. We start by presenting the theory behind optical interference filters and will then look at the special case of a purely dielectric filter.

2.1 Physical Model

An optical interference filter is built up of thin discrete layers of materials with various optical properties. We make a few assumptions to simplify the problem:

- Each layer in the filter is completely characterised by the *layer thickness* $d > 0$, the *electric conductivity* $\sigma \geq 0$, the *electric permittivity* $\epsilon = \epsilon_r \epsilon_0 \in \mathbb{R}$ and the *magnetic permeability* $\mu = \mu_r \mu_0 \geq 0$. Note that ϵ_r can be negative. This is typical in metals.

- The incident light enters the filter as a monochromatic, sinusoidal plane-wave travelling along the x-axis, normal to the surface. This assumption implies that we can forget about the polarisation of the light (see [7, Ch. 2.2.3]).

- The electromagnetic parameters σ, ϵ and μ are constant within each layer. They may be frequency-dependant, but because of our assumption that the incoming light is monochromatic, this is of no consequence.

- The layers are deposited on a relatively thick, non-absorbing substrate and the medium that surrounds the filter is non-absorbing.

- The substrate is semi-infinite. This implies that after the wave has passed into the substrate, it will never be reflected back again. Although this is quite unrealistic, it will help to keep the model simple enough for our use. See [7, Ch. 2.14] for a description of how to incorporate the effects of a finite substrate.

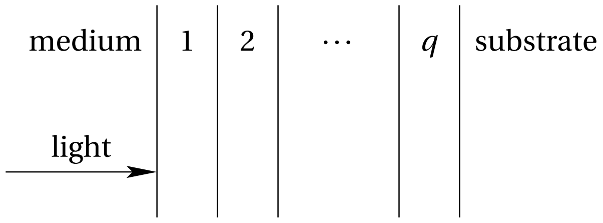


Figure 2.1: An optical interference filter.

See Fig. 2.1 for a schematic representation of a general filter with q layers.

We will follow the presentation in [7, Ch. 2.1–2.5] starting with Maxwell's equations[1]

$$\begin{aligned}\nabla\times\mathbf{H}&=\mathbf{J}+\frac{\partial\mathbf{D}}{\partial t},\\ \nabla\times\mathbf{E}&=-\frac{\partial\mathbf{B}}{\partial t},\\ \nabla\cdot\mathbf{D}&=\rho,\\ \nabla\cdot\mathbf{B}&=0,\end{aligned}$$

together with the relations

$$\begin{aligned}\mathbf{J}&=\sigma\mathbf{E},\\ \mathbf{D}&=\epsilon\mathbf{E},\\ \mathbf{B}&=\mu\mathbf{H}.\end{aligned}$$

First, we derive the electromagnetic wave equations within a single layer from the equations above. We eliminate **B** and **D**,

$$\begin{aligned}\nabla\times\mathbf{H}&=\sigma\mathbf{E}+\epsilon\frac{\partial\mathbf{E}}{\partial t},\\ \nabla\times\mathbf{E}&=-\mu\frac{\partial\mathbf{H}}{\partial t},\\ \nabla\cdot\mathbf{E}&=\frac{\rho}{\epsilon},\\ \nabla\cdot\mathbf{H}&=0,\end{aligned}\tag{2.1}$$

and obtain from these equations and by the assumption that the electromagnetic parameters are constant,

$$\begin{aligned}-\nabla\times(\nabla\times\mathbf{E})=\nabla^2\mathbf{E}&=\sigma\mu\frac{\partial\mathbf{E}}{\partial t}+\epsilon\mu\frac{\partial^2\mathbf{E}}{\partial t^2},\\ -\nabla\times(\nabla\times\mathbf{H})=\nabla^2\mathbf{H}&=\sigma\mu\frac{\partial\mathbf{H}}{\partial t}+\epsilon\mu\frac{\partial^2\mathbf{H}}{\partial t^2}.\end{aligned}\tag{2.2}$$

[1]See any book on electromagnetism for the derivation of these equations, e.g. [1, Ch. 7].

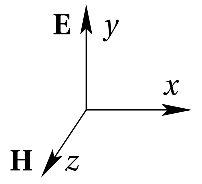


Figure 2.2: Orientation of the electric and magnetic fields.

We see that the electric and magnetic field intensities, **E** and **H** respectively, both satisfy the same differential equation.

We are only interested in solutions of Eq. (2.2) representing plane, sinusoidal waves moving in the x-direction of the form

$$\mathbf{E} = E\hat{\mathbf{y}} = \mathscr{E}\mathrm{e}^{\mathrm{i}\omega(t-x/v)}\hat{\mathbf{y}}, \qquad \mathbf{H} = H\hat{\mathbf{z}} = \mathscr{H}\mathrm{e}^{\mathrm{i}\omega(t-x/v)}\hat{\mathbf{z}} \tag{2.3}$$

where $\omega > 0$ is the angular frequency, v is the (complex) wave velocity, and $\mathscr{E}$ and $\mathscr{H}$ are (complex) constants. Note that this implies that **E** and **H** are complex vectors forming a right-handed set. Only the real parts of these have physical meaning, but the complex versions simplify the analysis. The real parts are also solutions since Eq. (2.2) is linear, allowing us to separate the real and imaginary parts. See Fig. 2.2 for an illustration of the directions of the electric and magnetic field intensities relative to the propagation direction. We insert Eq. (2.3) into Eq. (2.2) and get the *dispersion relation* (solution condition)

$$-\frac{\omega^2}{v^2} = \mathrm{i}\sigma\mu\omega - \epsilon\mu\omega^2.$$

Consider the solution in vacuum, where $\sigma = 0$, $v = c$, $\epsilon = \epsilon_0$ and $\mu = \mu_0$. This gives us an expression for the speed of light in vacuum

$$c^2 = \frac{1}{\epsilon_0\mu_0}.$$

Let us define the *complex refractive index* N by

$$N^2 \overset{\text{def}}{=} \frac{c^2}{v^2} = \epsilon_r\mu_r - \mathrm{i}\frac{\sigma\mu_r}{\epsilon_0\omega}. \tag{2.4}$$

Further, let us define

$$N = \frac{c}{v} \overset{\text{def}}{=} n - \mathrm{i}k,$$

where n is called the *refractive index* and k is called the *extinction coefficient.* Letting $\lambda \overset{\text{def}}{=} 2\pi c/\omega$ be the *wavelength* in vacuum, we substitute into Eq. (2.3) and get

$$\frac{E}{\mathscr{E}} = \frac{H}{\mathscr{H}} = \mathrm{e}^{\mathrm{i}(\omega t-(2\pi N/\lambda)x)} = \mathrm{e}^{-(2\pi k/\lambda)x}\mathrm{e}^{\mathrm{i}(\omega t-(2\pi n/\lambda)x)}. \tag{2.5}$$

From this, we readily see that the larger k is, the faster the wave will be absorbed by the material. This also rules out the possibility of k being negative, as this leads to the wave amplitude growing exponentially over time. It would only be physically possible to have $k < 0$ if we could somehow continuously supply energy to the wave, but this is outside the scope of this thesis.

Now, since

$$N^2 = n^2 - k^2 - 2\mathrm{i}nk,$$

we get from Eq. (2.4) that

$$\begin{aligned} n^2 - k^2 &= \epsilon_r \mu_r, \\ 2nk &= \frac{\sigma \mu_r}{\epsilon_0 \omega}. \end{aligned}$$

Since the right-hand side of the last equation and k are both non-negative, we must also have that n is non-negative.[2] Thus, N lies in the closed fourth quadrant of $\mathbb{C}$.

Our next step is to find the relationship between E and H. Let us insert the solution Eq. (2.3) into Eq. (2.1):

$$\begin{aligned} \nabla \times \mathbf{H} &= \sigma \mathbf{E} + \epsilon \frac{\partial \mathbf{E}}{\partial t} \\ -\frac{\partial H}{\partial x} \widehat{\mathbf{y}} &= (\sigma + \mathrm{i}\epsilon\omega) E \widehat{\mathbf{y}} \\ \mathrm{i}\frac{2\pi N}{\lambda_0} H &= \mathrm{i}\frac{\omega N^2}{c^2 \mu} E \\ \frac{\omega}{c} H &= \frac{\omega N}{c^2 \mu} E \\ H &= \frac{N}{c\mu} E. \end{aligned}$$

We define the *characteristic optical admittance*

$$y \overset{\text{def}}{=} \frac{N}{c\mu}, \tag{2.6}$$

so that $H = yE$. Let us denote y in vacuum as $y_0 = 1/c\mu_0$. Then $y = y_0 N/\mu_r$.

Now, let us discuss what happens when the wave crosses an interface between two layers. We assume that the two layers consist of materials with different optical properties, otherwise we could consider the two layers as one by simply adding their thicknesses. This will split the incident electromagnetic wave into two parts at the interface, with one reflected component and one transmitted component. Let us use

[2] If we had allowed $\mu_r < 0$, we could in fact have achieved a material with $n < 0$. This is the definition of a *metamaterial*, but that topic is outside the scope of this thesis.

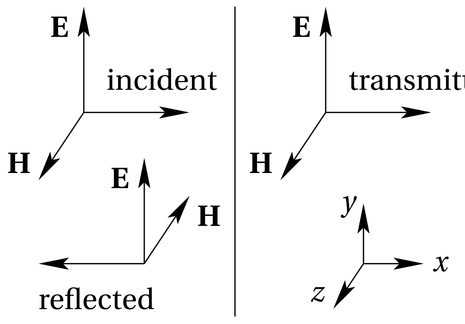


Figure 2.3: Convention defining positive directions for incident, reflected and transmitted waves.

the subscript i for the incident wave, r for the reflected wave and t for the transmitted wave. There will be no absorption at the interface since it has zero thickness and the electric and magnetic fields must vary continuously across it.

The electric and magnetic fields always form right-handed sets together with the direction of propagation. We define the positive direction of the electric field to always lie in the positive y-direction. This implies that the magnetic component of the reflected wave will point in the negative z-direction (see Fig. 2.3) and we get

$$E_t = E_i + E_r, \qquad H_t = H_i - H_r.$$

Consider now the situation where the filter consists of a single layer with complex refractive index N, characteristic optical admittance y and thickness d deposited on a substrate with characteristic admittance y_s. As mentioned earlier, we simplify the situation by assuming that there are no waves in the substrate travelling in the negative direction. We also define E_m and H_m as the electric and magnetic field intensities in the medium at the first interface, E_s and H_s as the intensities in the substrate at the second interface, E_m^+, H_m^+, E_m^-, H_m^- as the intensities of the forward and backward going waves respectively in the layer at the first interface, and E_s^+, H_s^+, E_s^-, H_s^- as the intensities of the forward and backward going waves respectively in the layer at the second interface. See Fig. 2.4 for a graphical summary.

Since the electric and magnetic field intensities are continuous across the interfaces and since there are no waves travelling in the negative direction inside the substrate, we have

$$E_s = E_s^+ + E_s^-, \qquad H_s = H_s^+ - H_s^- = y(E_s^+ - E_s^-).$$

We can transform these equations to

$$E_s^+ = \frac{1}{2}\left(E_s + \frac{H_s}{y}\right), \qquad E_s^- = \frac{1}{2}\left(E_s - \frac{H_s}{y}\right).$$

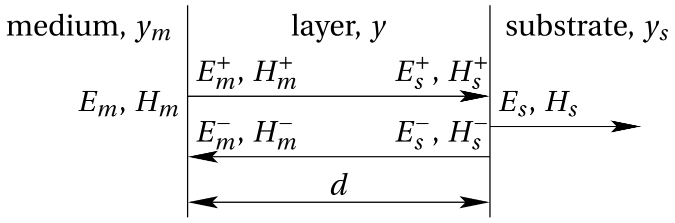


Figure 2.4: Electric and magnetic fields in a one-layer filter.

At the same moment in time, we can find the electric and magnetic field intensities E_m and H_m at the interface between the medium and the layer by considering Eq. (2.5). Define

$$\delta \overset{\text{def}}{=} \frac{2\pi N d}{\lambda}$$

so that we simply need to multiply by $e^{i\delta}$ for the positive-going wave and $e^{-i\delta}$ for the negative-going wave. We get

$$E_m^+ = E_s^+ e^{i\delta}, \qquad E_m^- = E_s^- e^{-i\delta}.$$

We are now ready to find E_m and H_m. The electric field intensity is

$$\begin{aligned} E_m &= E_m^+ + E_m^- \\ &= \frac{1}{2}\left(E_s + \frac{H_s}{y}\right)e^{i\delta} + \frac{1}{2}\left(E_s - \frac{H_s}{y}\right)e^{-i\delta} \\ &= \frac{e^{i\delta}+e^{-i\delta}}{2}E_s + \frac{e^{i\delta}-e^{-i\delta}}{2y}H_s \\ &= E_s\cos\delta + H_s\frac{i\sin\delta}{y}, \end{aligned} \tag{2.7}$$

and the magnetic field intensity is

$$\begin{aligned} H_m &= H_m^+ - H_m^- \\ &= y(E_m^+ - E_m^-) \\ &= \frac{y}{2}\left(E_s + \frac{H_s}{y}\right)e^{i\delta} - \frac{y}{2}\left(E_s - \frac{H_s}{y}\right)e^{-i\delta} \\ &= \frac{e^{i\delta}-e^{-i\delta}}{2}yE_s + \frac{e^{i\delta}+e^{-i\delta}}{2}H_s \\ &= E_s i y\sin\delta + H_s\cos\delta. \end{aligned} \tag{2.8}$$

We combine Eq. (2.7) and Eq. (2.8) to form the matrix formula

$$\begin{bmatrix} E_m \\ H_m \end{bmatrix} = \begin{bmatrix} \cos\delta & (i\sin\delta)/y \\ iy\sin\delta & \cos\delta \end{bmatrix} \begin{bmatrix} E_s \\ H_s \end{bmatrix}. \tag{2.9}$$

It is easy to generalise this to the case where we have q layers instead of just one (see [7, Ch. 2.4]). The result is

$$\begin{bmatrix} E_m \\ H_m \end{bmatrix} = \left(\prod_{r=1}^{q} \begin{bmatrix} \cos\delta_r & (\mathrm{i}\sin\delta_r)/y_r \\ \mathrm{i}y_r\sin\delta_r & \cos\delta_r \end{bmatrix} \right) \begin{bmatrix} E_s \\ H_s \end{bmatrix}, \tag{2.10}$$

where the ordering of the matrices in the product is such that the leftmost matrix corresponds to the layer next to the medium. Let us define a normalised version of the electromagnetic field

$$\begin{bmatrix} B \\ C \end{bmatrix} \overset{\text{def}}{=} \begin{bmatrix} E_m/E_s \\ H_m/E_s \end{bmatrix} = \left(\prod_{r=1}^{q} \begin{bmatrix} \cos\delta_r & (\mathrm{i}\sin\delta_r)/y_r \\ \mathrm{i}y_r\sin\delta_r & \cos\delta_r \end{bmatrix} \right) \begin{bmatrix} 1 \\ y_s \end{bmatrix}. \tag{2.11}$$

Analogously to Eq. (2.6), let us define the *input optical admittance*

$$Y \overset{\text{def}}{=} \frac{H_m}{E_m} = \frac{H_m/E_s}{E_m/E_s} = \frac{C}{B}.$$

This shows that the input optical admittance is independent of the electromagnetic fields and only depends on the optical properties of the layers and the substrate.

One of most important quantities in connection with optical filters is the *reflectance R* which is defined as the ratio of the reflected irradiance to the incident irradiance. Let us define the *amplitude reflection coefficient*

$$\rho \overset{\text{def}}{=} \frac{y_m - Y}{y_m + Y},$$

where y_m is the characteristic optical admittance of the medium. From [7, Ch. 2.2.1] we get that $R = |\rho|^2$. Alternatively, we can write

$$R = |\rho|^2 = \left| \frac{y_m B - C}{y_m B + C} \right|^2. \tag{2.12}$$

In Sec. 3.2 we prove that $0 \leq R < 1$ (at least for dielectric filters, which are defined in the next section).

An important fact that we will need later is that the matrices in Eq. (2.11) all have determinant equal to one. This means that they are members of the *special linear group* $\mathrm{SL}(2,\mathbb{C})$.

2.2 Dielectric Filters

An important special case, the case that we will focus our attention on from now on, is when all layers in the filter consist of dielectric materials, i.e. materials with $\sigma = 0$ and

$\epsilon_r > 0$. These conditions imply that $k = 0$. In other words, dielectric filters are filters that do not absorb light, only reflect and transmit it. This is important in practice, where large absorption can cause high temperatures inside the filter itself, potentially destroying it.

From Eq. (2.11), we see that the matrices only depend on δ and y. Since $N = n$ is real, by definition, so is δ and y. According to [7, Ch. 2.1], at optical wavelengths, we can assume that $\mu_r = 1$, so $y = ny_0$. Thus, we can completely characterise the filter by $\{(n_i, \delta_i)\}_{i=1}^{q} \in \mathbb{R}^{2q}$, in addition to the refractive index n_s of the substrate. Of course, we can only choose non-negative n_i and δ_i.

2.3 The Design Problem

We now turn our attention to the *design problem*. In the case of dielectric anti-reflection filters, we would like to design a dielectric filter that transmits as much light as possible in the wavelength interval that we are interested in. Outside that interval, it does not matter how the filter behaves. We wish to formulate this as an optimisation problem. Given the number of layers q of the filter and the refractive index n_s of the substrate, we must find a set of parameters $\alpha \overset{\text{def}}{=} \{\alpha_i\}_{i=1}^{q}$, where $\alpha_i \overset{\text{def}}{=} (n_i, \delta_i)$ so that the reflectance R is minimised in the wavelength interval. Let us discretise the interval by selecting m wavelengths $\{\lambda_j\}_{j=1}^{m}$ from it. From Eq. (2.12) we see that R is a function of the input optical admittance Y, the value of which is dependant only on α (and n_s, which is not altered in the optimisation), so we can write $Y = Y(\alpha)$. Next, we need an *objective function* f that reaches a minimum at the solution. Let us choose the sum of squared deviations,

$$f(\alpha) \overset{\text{def}}{=} \frac{1}{2} \sum_{j=1}^{m} R\big(Y(\alpha^j)\big)^2,$$

where $\alpha^j \overset{\text{def}}{=} \{\alpha_i^j\}_{i=1}^{q}$, $\alpha_i^j \overset{\text{def}}{=} (n_i, \delta_i/\lambda_j)$. Then we can write the optimal solution as

$$\alpha^* = \arg\min f(\alpha).$$

In Sec. 5.2 we use this sum of squares formulation to solve the design problem.

Now, let us consider the case where we are only interested in the behaviour at a single wavelength. We see that we can simply set $f = R$, i.e. we minimise the reflectance directly at the wanted wavelength. This design problem can be solved exactly, as we will see in Sec. 5.1.

Chapter 3

The Geometry of the Design Problem

We generalise the objective function f from Sec. 2.3 to all functions that are only dependent on the admittance Y (possibly at multiple wavelengths). Since $Y = C/B$, we are not interested in the value of B and C themselves, only the ratio between them. This means that we can multiply the vector $[1, y_s]^{\mathrm{T}}$ in Eq. (2.11) by any non-zero, complex number w without changing Y. Let us consider the normalised electromagnetic field vector at the surface of a one-layer filter:

$$\begin{bmatrix} B \\ C \end{bmatrix} w = \begin{bmatrix} \cos\delta & (\mathrm{i}\sin\delta)/y \\ \mathrm{i}y\sin\delta & \cos\delta \end{bmatrix} \begin{bmatrix} 1 \\ y_s \end{bmatrix} w = \begin{bmatrix} w\big(\cos\delta + (\mathrm{i}y_s\sin\delta)/y\big) \\ w(\mathrm{i}y\sin\delta + y_s\cos\delta) \end{bmatrix}.$$

We let

$$w = \frac{1}{\cos\delta + (\mathrm{i}y_s\sin\delta)/y},$$

so that

$$\begin{bmatrix} B \\ C \end{bmatrix} w = \begin{bmatrix} 1 \\ Y \end{bmatrix}.$$

This means that we can interpret each matrix in Eq. (2.11) (representing a layer in the filter) as a transformation acting on the admittance outputted from the previous filter layer. Thus, for a multi-layer filter, we arrive at the input optical admittance Y after a sequence of transformations is applied to y_s.

3.1 Linear Fractional Transformations

Linear fractional transformations (also called *Möbius transformations*) are maps on $\mathbb{C}$ of the form

$$w \mapsto \frac{aw+b}{cw+d},$$

where $a, b, c, d \in \mathbb{C}$ and $ad - bc \neq 0$. If we scale a, b, c and d by a common non-zero factor, we see that we actually get the same transformation by cancellation. By only considering transformations that have $ad - bc = 1$, we remove this ambiguity. We let the space of linear fractional transformations satisfying $ad - bc = 1$ be called M.

The transformation from y_s to Y is exactly of this form.

$$Y = \frac{\mathrm{i}y\sin\delta + y_s\cos\delta}{\cos\delta + (\mathrm{i}y_s\sin\delta)/y}.$$

A more natural form can be achieved if we define the *input impedance, medium impedance* and *substrate impedance* as

$$Z \overset{\text{def}}{=} \frac{1}{Y}, \qquad z_m \overset{\text{def}}{=} \frac{1}{y_m} \qquad \text{and} \qquad z_s \overset{\text{def}}{=} \frac{1}{y_s}, \tag{3.1}$$

respectively. Then

$$Z = \frac{(\cos\delta)z_s + (\mathrm{i}\sin\delta)/y}{(\mathrm{i}y\sin\delta)z_s + \cos\delta},$$

which is the linear fractional transformation in M with $a = d = \cos\delta$, $b = (\mathrm{i}\sin\delta)/y$ and $c = \mathrm{i}y\sin\delta$ that transforms z_s to Z.

This corresponds exactly to the elements of the original matrix and motivates the map

$$\pi\colon\ \mathrm{SL}(2,\mathbb{C}) \to M, \qquad \begin{bmatrix} a & b \\ c & d \end{bmatrix} \mapsto \frac{aw+b}{cw+d}.$$

It is easy to show that M is a group. Then, since $\mathrm{SL}(2,\mathbb{C})$ is a group, an immediate implication is that π is a *group homomorphism*, i.e.

$$\pi(XY) = \pi(X) \circ \pi(Y)$$

for all $X, Y \in \mathrm{SL}(2,\mathbb{C})$. The *kernel* of π, written $\ker\pi$, is the set of elements in $\mathrm{SL}(2,\mathbb{C})$ that get mapped to the identity transformation $e(w) = w$. We easily find that

$$\ker\pi = \{I, -I\}.$$

Let us define the *projective special linear group* as the quotient group where we collapse all scalar multiples of each element of the special linear group to a single element. In $\mathrm{SL}(2,\mathbb{C})$, this means that we identify every element X with $-X$:

$$\mathrm{PSL}(2,\mathbb{C}) \overset{\text{def}}{=} \mathrm{SL}(2,\mathbb{C})/\{I, -I\}.$$

Since the kernel is a normal subgroup, we have from the first isomorphism theorem (see [3, Thm. 34.2]) that M and $\mathrm{PSL}(2,\mathbb{C})$ are *isomorphic*. In other words, M and $\mathrm{PSL}(2,\mathbb{C})$ are structurally identical (in the sense of group theory).

In the case of a dielectric layer,

$$\phi(w) = \frac{w\cos\delta + \mathrm{i}(\sin\delta)/y}{\mathrm{i}wy\sin\delta + \cos\delta}, \tag{3.2}$$

where

$$\delta = \frac{2\pi n d}{\lambda} \qquad \text{and} \qquad y = n y_0,$$

as detailed in Sec. 2.2. We see that δ and y are both real and determine a unique element $\phi \in M$ if we choose δ so that $0 \le \delta < \pi$. Of course, we can only choose $n > 0$, which implies that $y > 0$. All the corresponding matrices in PSL(2,$\mathbb{C}$) are of the form

$$\begin{bmatrix} a & \mathrm{i}b \\ \mathrm{i}c & a \end{bmatrix},$$

where $a, b, c \in \mathbb{R}$. Let us calculate the product of two such matrices:

$$\begin{bmatrix} a_2 & \mathrm{i}b_2 \\ \mathrm{i}c_2 & a_2 \end{bmatrix} \begin{bmatrix} a_1 & \mathrm{i}b_1 \\ \mathrm{i}c_1 & a_1 \end{bmatrix} = \begin{bmatrix} a_2 a_1 - b_2 c_1 & \mathrm{i}(a_2 b_1 + b_2 a_1) \\ \mathrm{i}(c_2 a_1 + a_2 c_1) & -c_2 b_1 + a_2 a_1 \end{bmatrix}$$

Since in general $b_2 c_1 \neq c_2 b_1$, the product is not of the original form. However, if we generalise to matrices in PSL(2,$\mathbb{C}$) of the form

$$\begin{bmatrix} a & \mathrm{i}b \\ \mathrm{i}c & d \end{bmatrix},$$

with $a, b, c, d \in \mathbb{R}$, products *are* of this form. This means that these form a *subgroup* of PSL(2,$\mathbb{C}$), and by isomorphism the corresponding transformations form a subgroup of M. Let us call this subgroup M_d (d standing for dielectric).

3.2 The Geometry of the Reflectance

We are interested in how the reflectance R depends on the input impedance Z. From Eq. (2.12) and Eq. (3.1), we have

$$R = \left| \frac{y_m - Y}{y_m + Y} \right|^2 = \left| \frac{1/z_m - 1/Z}{1/z_m + 1/Z} \right|^2 = \left| \frac{Z - z_m}{Z + z_m} \right|^2.$$

Then the level set for $R = R'$ is $Z_{R'} = \{Z \in \mathbb{C} : R = R'\}$. This set consists of all Z such that $(Z - z_m)/(Z + z_m)$ lies on a circle with radius $\sqrt{R'}$ centred at the origin. Thus,

$$Z_{R'} = \left\{ Z \in \mathbb{C} : \frac{Z - z_m}{Z + z_m} = \sqrt{R'} \mathrm{e}^{\mathrm{i}\theta}, \text{for some } \theta \in \mathbb{R} \right\}.$$

After a little manipulation of the condition and letting $Z = x + \mathrm{i}y$, we can eliminate θ and get

$$x^2 + y^2 - 2 x z_m \frac{1 + R'}{1 - R'} = -z_m^2.$$

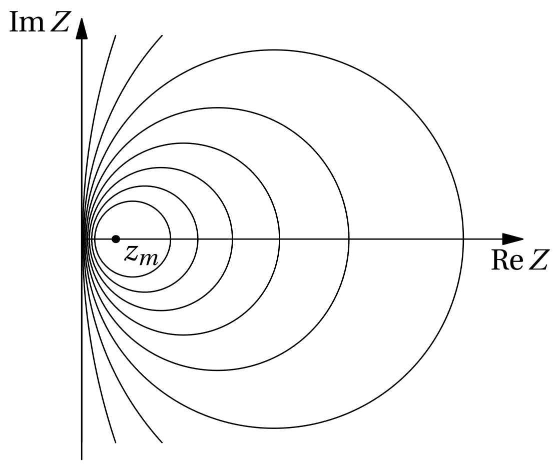


Figure 3.1: Level sets of $R = R'$, $0 \le R' < 1$.

This is the equation for a circle with centre

$$\left(z_m \frac{1+R'}{1-R'}, 0\right),$$

and radius

$$z_m \frac{2\sqrt{R'}}{1-R'}.$$

In Fig. 3.1, we have plotted some level sets for $0 \le R' < 1$. We see that the entire open right half-plane of $\mathbb{C}$ is covered by $0 \le R' < 1$. There are two special cases: $Z_0 = \{z_m\}$ (a circle with radius 0), and $Z_1 = \mathrm{i}\mathbb{R}$ (a 'circle' with radius ∞).

Let us split the transformation $\phi(z_s) \in M_d$ into its real and imaginary parts:

$$\phi(z_s) = \frac{az_s + \mathrm{i}b}{\mathrm{i}cz_s + d} = \frac{(az_s + \mathrm{i}b)(d - \mathrm{i}cz_s)}{c^2 z_s^2 + d^2} = \frac{z_s + \mathrm{i}(bd - acz_s^2)}{c^2 z_s^2 + d^2}$$

Since $z_s > 0$, we immediately see that $\phi(z_s)$ lies in the open right half-plane of $\mathbb{C}$, corresponding to the level sets for $0 \le R' < 1$. Thus, it is impossible to transform z_s into the closed left half-plane of $\mathbb{C}$. Consequently, R must be less than one.[1] Physically, this means that it is *impossible* to construct a perfect mirror with only dielectric materials.

In a real-world anti-reflection filter, we would like $R = 0$ at some wavelength, so we must find $\phi \in M_d$ that transforms z_s to z_m. Then we must have $\phi(z_s) = z_m$, so

$$z_s = z_m(c^2 z_s^2 + d^2) \qquad \text{and} \qquad bd = acz_s^2,$$

together with the usual $ad + bc = 1$. Let us parametrise the possible transformations by setting

$$d = \sqrt{\frac{z_s}{z_m}}\, t.$$

[1] It is possible to get arbitrarily close to $R = 1$, though.

Then

$$c = \pm\frac{1}{z_s}\sqrt{\frac{z_s}{z_m} - d^2} = \pm\sqrt{\frac{1-t^2}{z_m z_s}},$$

which implies that we must choose t so that $t^2 \le 1$. We can combine the three equations to get

$$b = z_m z_s c = \pm\sqrt{z_m z_s(1-t^2)}.$$

Another combination gives us

$$a = \frac{z_m}{z_s} d = \sqrt{\frac{z_m}{z_s}}\, t.$$

This parametrisation actually covers the same elements twice in M_d. Elements that are parametrised by positive t are identified with the elements mapped by negative t if we switch the sign of b and c. One way to remedy this is to only allow positive square roots in the expressions for b and c, while letting $-1 < t \le 1$. If we interpret d as time, we see that we trace out half an ellipse in abc-space as t goes from -1 to 1.

We can write this simpler by letting $t = \cos\theta$, where $0 \le \theta < \pi$. Then

$$a = \sqrt{\frac{z_m}{z_s}}\cos\theta, \quad b = \sqrt{z_m z_s}\sin\theta, \quad c = \frac{\sin\theta}{\sqrt{z_m z_s}} \quad \text{and} \quad d = \sqrt{\frac{z_s}{z_m}}\cos\theta.$$

3.3 Physically Realisable Transformations

As we saw in Eq. (3.2), an element of M_d corresponding to a physical layer with impedance $z = 1/y$ is of the special form

$$\phi(w) = \frac{w\cos\delta + \mathrm{i}(\sin\delta)/y}{\mathrm{i}wy\sin\delta + \cos\delta} = \frac{w\cos\delta + \mathrm{i}z\sin\delta}{\mathrm{i}w(\sin\delta)/z + \cos\delta}. \tag{3.3}$$

We will now explore how ϕ transforms points w in the complex plane. Let $w = u + \mathrm{i}v$, $\phi(w) = x + \mathrm{i}y$, $\alpha + \mathrm{i}\beta = \mathrm{e}^{\mathrm{i}\delta}$, multiply Eq. (3.3) by the denominator, and separate the real and imaginary parts. Then we get

$$\begin{aligned} x\left(\alpha - \beta\frac{v}{z}\right) - y\beta\frac{u}{z} &= \alpha u, \\ x\beta\frac{u}{z} + y\left(\alpha - \beta\frac{v}{z}\right) &= \alpha v + \beta z. \end{aligned}$$

By eliminating α and β, we can transform this to

$$x^2 + y^2 - 2x\frac{|w|^2 + z^2}{2u} = -z^2.$$

This is the equation for a circle with centre

$$(x_0, y_0) = \left(\frac{|w|^2 + z^2}{2u}, 0 \right) \tag{3.4}$$

and radius

$$r = \sqrt{x_0^2 - z^2}.$$

If we set $\delta = 0$ in Eq. (3.3), we get $\phi(w) = w$ the identity transformation, as expected. If we set $\delta = \pi/2$, $\phi(w) = z^2/w$. So all possible values that $\phi(w)$ can take lie on the unique circle that goes through w and z^2/w with centre on the real line. Note that if $z = w$, the circle collapses to a single point.

Now we show that the right half-plane is transformed into the right half-plane. By checking the sign of $x_0 - r$ (the leftmost point of the circle), we will see if it is possible to end up in the left half-plane. Let $u > 0$. From Eq. (3.4) we then have that $x_0 > 0$. So

$$x_0 - r = x_0 - \sqrt{x_0^2 - z^2}$$

is positive as long as $x_0^2 - z^2 \geq 0$, (and of course, $z \neq 0$). We show that this is indeed the case:

$$\begin{aligned} x_0^2 - z^2 &= \left(\frac{u^2 + v^2 + z^2}{2u} \right)^2 - z^2 \\ &= \frac{(u^2 + v^2 + z^2)^2 - 4u^2 z^2}{4u^2} \\ &= \frac{(u^2 - z^2)^2 + v^2(v^2 + 2u^2 + 2z^2)}{4u^2} \\ &\geq 0. \end{aligned}$$

Thus, we have proved the important fact that any sequence of physically realisable transformations of the impedance z_s will give a result that lies in the open right half-plane of $\mathbb{C}$. The same result holds if we choose to use admittances instead of impedances.

Chapter 4

Optimisation Methods

To solve our design problem, we need numerical optimisation methods. In this chapter, we present the methods that we will use. The presentation here is not rigorous or complete in any manner. See [9, Ch. 3, 5 and 6] for a more comprehensive survey.

The main goal in optimisation is to find the minimum of some objective function $f\colon \mathbb{R}^n \to \mathbb{R}$. We can formulate this more mathematically: Find an $x^* \in \mathbb{R}^n$ so that

$$x^* = \operatorname*{arg\,min}_{x \in \mathbb{R}^n} f(x).$$

Line search methods and *trust-region* methods are two of the most used strategies in numerical optimisation. We will focus on the former of these and present two important classes of line search methods that we will apply to the design problem. These classes are *quasi-Newton* methods (see Sec. 4.2) and *nonlinear conjugate gradient* methods (see Sec. 4.3). Much of the material in this chapter is taken from [9, Ch. 3, 5 and 6].

4.1 Line Search Methods

Line search methods are iterative methods that work as follows. We start at some given point $x_0 \in \mathbb{R}^n$, and for each iteration, say iteration k, find a *search direction* $p_k \in \mathbb{R}^n$ that is a *descent direction*, i.e. a direction satisfying $p_k^{\mathrm{T}} \nabla f(x_k) < 0$. Next, we find a step length α_k satisfying the *strong Wolfe conditions*

$$\begin{aligned} f(x_k + \alpha_k p_k) &\le f(x_k) + c_1 \alpha_k \nabla f(x_k)^{\mathrm{T}} p_k, \\ |\nabla f(x_k + \alpha_k p_k)^{\mathrm{T}} p_k| &\le c_2 |\nabla f(x_k)^{\mathrm{T}} p_k|, \end{aligned}$$

with $0 < c_1 < c_2 < 1$.[1] This ensures that f decreases sufficiently for the iteration to converge to a minimiser. According to [9, Lem. 3.1], as long as f is continuously differentiable and bounded below, we can always find a step length that satisfies the strong Wolfe conditions. The next iterate is then

$$x_{k+1} = x_k + \alpha_k p_k.$$

The only thing that separate the different line search methods from each other is the algorithm we use to find p_k.

4.2 Quasi-Newton Methods

Quasi-Newton methods are methods that are related to the *Newton* method. In each iteration of both methods, we first determine a model function

$$m_k(p) = f(x_k) + \nabla f(x_k)^{\mathrm{T}} p + \frac{1}{2} p^{\mathrm{T}} B_k p, \tag{4.1}$$

where B_k is a symmetric and positive definite matrix. The model function quadratically approximates f around x_k. We easily find the minimum of m_k by differentiating Eq. (4.1). The minimiser is

$$p_k = -B_k^{-1} \nabla f(x_k). \tag{4.2}$$

As the notation suggests, we use this minimiser as the search direction p_k.

In the Newton method, $B_k = \nabla^2 f(x_k)$, the Hessian of f. The Hessian may be expensive to calculate (this is the case for us), so instead we make do with an approximation. The approximation is refined at each step so that, gradually we approach the true Hessian. This gives us the quasi-Newton methods.

4.2.1 The BFGS Method

The BFGS method[2] is one of the most important quasi-Newton methods. It is based on the idea that ∇m_{k+1} should match ∇f at both x_k and x_{k+1}. This condition implies that B_{k+1} must satisfy the *secant equation*

$$B_{k+1} s_k = y_k,$$

where

$$s_k \overset{\text{def}}{=} x_{k+1} - x_k, \quad \text{and} \quad y_k \overset{\text{def}}{=} \nabla f(x_{k+1}) - \nabla f(x_k).$$

[1]Nocedal and Wright [9] suggest using $c_1 = 10^{-4}$ and $c_2 = 0.9$ or $c_2 = 0.1$ for quasi-Newton and nonlinear conjugate gradient methods respectively.

[2]Named after Broyden, Fletcher, Goldfarb, and Shanno.

The strong Wolfe conditions guarantee that the secant equation has a solution, but it may not be unique. To get uniqueness, we select the symmetric positive definite B_{k+1} that lies closest to B_k in the sense of norms:

$$B_{k+1} = \underset{B}{\arg\min} \|B - B_k\|_W,$$

where we use the weighted Frobenius norm

$$\|A\|_W \overset{\text{def}}{=} \|W^{1/2} A W^{1/2}\|_F.$$

The weighting matrix W is defined as $W = \overline{G}_k^{-1}$, where

$$\overline{G}_k \overset{\text{def}}{=} \int_0^1 \nabla^2 f(x_k + \tau \alpha_k p_k)\, \mathrm{d}\tau$$

is the *average Hessian.* The unique solution is then

$$B_{k+1} = (I - \rho_k y_k s_k^{\mathrm{T}}) B_k (I - \rho_k s_k y_k^{\mathrm{T}}) + \rho_k y_k y_k^{\mathrm{T}}, \tag{4.3}$$

where

$$\rho_k \overset{\text{def}}{=} \frac{1}{y_k^{\mathrm{T}} s_k}.$$

Eq. (4.3) is called the *DFP updating formula.*[3]

From Eq. (4.2) we see that we do not need B_k directly, but rather its inverse. Let us denote

$$H_k \overset{\text{def}}{=} B_k^{-1}$$

We want to find an updating formula for H_{k+1}. By imposing the conditions that we used to find B_{k+1} on H_{k+1} instead, we get

$$H_{k+1} = (I - \rho_k s_k y_k^{\mathrm{T}}) H_k (I - \rho_k y_k s_k^{\mathrm{T}}) + \rho_k s_k s_k^{\mathrm{T}}.$$

This is the *BFGS updating formula* and is the one that is used in practice.

4.3 Nonlinear Conjugate Gradient Methods

Another important class of line search methods is the class of *nonlinear conjugate gradient* methods. They are based on the linear conjugate gradient method (simply called the CG method), which is a method for solving a linear system of equations,

[3]Named after Davidon, Fletcher and Powell.

$Ax = b$, where A is a symmetric positive definite matrix. We can restate this as an optimisation problem. Find

$$\min \phi(x), \quad \text{where} \quad \phi(x) \overset{\text{def}}{=} \frac{1}{2} x^{\mathrm{T}} A x - b^{\mathrm{T}} x.$$

The gradient of ϕ is

$$\nabla \phi(x) = Ax - b \overset{\text{def}}{=} r(x), \tag{4.4}$$

where r is called the *residual* of the linear system. In each iteration of the CG method, we perform the following calculations (see [9, Alg. 5.2]):

$$\begin{aligned} \alpha_k &= \frac{\|r_k\|}{p_k^{\mathrm{T}} A p_k}, \\ x_{k+1} &= x_k + \alpha_k p_k, \\ r_{k+1} &= r_k + \alpha_k A p_k, \\ \beta_{k+1} &= \frac{\|r_{k+1}\|}{\|r_k\|}, \\ p_{k+1} &= -r_{k+1} + \beta_{k+1} p_k. \end{aligned}$$

In nonlinear conjugate gradient methods, we find α_k with a line search along p_k, calculate β_{k+1} in one of the ways discussed in the following subsections, and set $p_{k+1} = -\nabla f(x_{k+1}) + \beta_{k+1} p_k$ (see [9, Alg. 5.4]). By comparison with Eq. (4.4), we see that substitution of the residuals with the gradient of the objective function f is quite natural.

In [9, Sec. 5.2] several possibilities for the calculation of β_{k+1} are suggested. We present them here and later test their performance on the design problem. Nocedal barely touches the final two (the DY and HZ methods) because they are relatively new, so it will be extra interesting to see how they perform.

4.3.1 Fletcher–Reeves (FR)

This is the simplest nonlinear conjugate gradient method, where we simply substitute the residuals with the corresponding gradients.

$$\beta_{k+1}^{\mathrm{FR}} = \frac{\nabla f(x_{k+1})^{\mathrm{T}} \nabla f(x_{k+1})}{\nabla f(x_k)^{\mathrm{T}} \nabla f(x_k)} = \frac{\|\nabla f(x_{k+1})\|}{\|\nabla f(x_k)\|}$$

4.3.2 Polak–Ribière (PR)

Experience has shown that this variant is more efficient and robust than the FR method.

$$\beta_{k+1}^{\mathrm{PR}} = \frac{\nabla f(x_{k+1})^{\mathrm{T}} y_k}{\|\nabla f(x_k)\|}$$

4.3.3 Modified Polak–Ribière (PR+)

For the PR method, the strong Wolfe conditions are not enough to guarantee that p_k is a descent direction, but by setting

$$\beta_{k+1}^{\mathrm{PR+}} = \max\{\beta_{k+1}^{\mathrm{PR}}, 0\},$$

giving us the PR+ method, we restore this property.

4.3.4 Fletcher–Reeves–Polak–Ribière (FR–PR)

For all $k \geq 2$,

$$\beta_k^{\mathrm{FR\text{–}PR}} = \begin{cases} -\beta_k^{\mathrm{FR}} & \text{if} \quad \beta_k^{\mathrm{PR}} < -\beta_k^{\mathrm{FR}} \\ \beta_k^{\mathrm{PR}} & \text{if} \quad |\beta_k^{\mathrm{PR}}| \leq \beta_k^{\mathrm{FR}} \\ \beta_k^{\mathrm{FR}} & \text{if} \quad \beta_k^{\mathrm{PR}} > \beta_k^{\mathrm{FR}}. \end{cases}$$

This method is a modification of the PR method, based on the fact that it is possible to get global convergence for any β_k satisfying $|\beta_k| \leq \beta_k^{\mathrm{FR}}$.

4.3.5 Hestenes–Stiefel (HS)

This variant is quite similar to the PR method in theoretical convergence properties.

$$\beta_{k+1}^{\mathrm{HS}} = \frac{\nabla f(x_{k+1})^{\mathrm{T}} y_k}{y_k^{\mathrm{T}} p_k}$$

4.3.6 Dai–Yuan (DY)

From [2], we have

$$\beta_{k+1}^{\mathrm{DY}} = \frac{\|\nabla f(x_{k+1})\|}{y_k^{\mathrm{T}} p_k}.$$

4.3.7 Hager–Zhang (HZ)

From [4], we have

$$\beta_{k+1}^{\mathrm{HZ}} = \left(y_k - 2 p_k \frac{\|y_k\|}{y_k^{\mathrm{T}} p_k} \right)^{\mathrm{T}} \frac{\nabla f(x_{k+1})}{y_k^{\mathrm{T}} p_k}.$$

4.4 The Levenberg–Marquardt Method

OpenFilters uses a nonlinear least-squares method in its optimisation routines. Specifically, the Levenberg–Marquardt method (see [6]). The method is described in [9, Sec. 10.3]. We have not implemented this method in this thesis, but include it here since we want to compare our results with the results obtained with OpenFilters.

Chapter 5

Solving the Design Problem

In this chapter, we will first solve the design problem at a single wavelength, then we will look at the multi-wavelength problem.

5.1 Single-Wavelength Solution

From the previous chapters, we now have all the tools necessary to solve the design problem at a single wavelength. As we will see, this problem can be solved exactly, without resorting to numerical methods.

5.1.1 One-Layer Dielectric Anti-Reflective Filters

If all we need is $R = 0$ at a single wavelength, we can solve the design problem in a single layer. We need to construct a one-layer dielectric filter that transforms z_s to z_m. Applying what we learnt in the previous section, we must find a circle that goes through both z_s and z_m and has its centre on the real line. This is the circle with centre $(z_m + z_s)/2$ and radius $(z_m - z_s)/2$ (see Fig. 5.1).

Then, from Eq. (3.4) with $w = u = z_s$, we have that

$$x_0 = \frac{z_s + z_m}{2} = \frac{z_s^2 + z^2}{2z_s}.$$

We solve for z and get

$$z = \sqrt{z_m z_s}.$$

The only thing left to do is to find the layer thickness that gives us the point z_m on the circle. In other words, we must find $0 \le \delta < \pi$ so that $\phi(z_s) = z_m$. From Eq. (3.3), we get

$$z_m = \frac{z_s \cos\delta + \mathrm{i}\sqrt{z_m z_s}\sin\delta}{\mathrm{i}z_s(\sin\delta)/\sqrt{z_m z_s} + \cos\delta} = \frac{z_s\sqrt{z_m z_s}\cos\delta + \mathrm{i}z_m z_s \sin\delta}{\mathrm{i}z_s \sin\delta + \sqrt{z_m z_s}\cos\delta}.$$

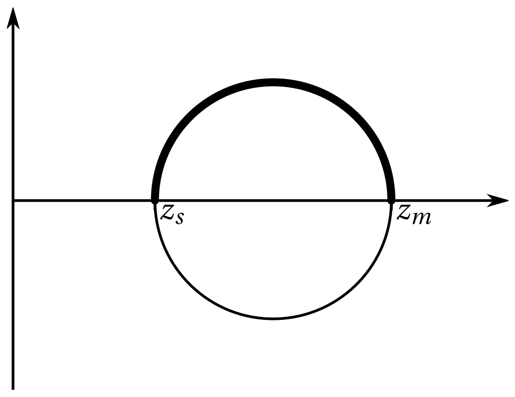
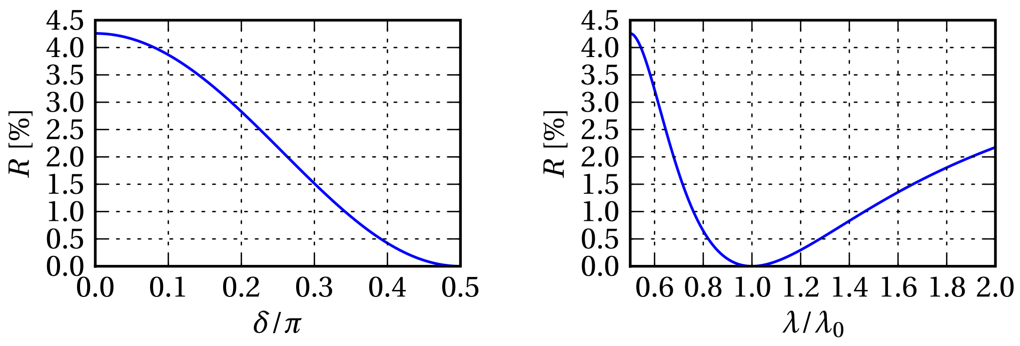


Figure 5.1: One-layer solution that gives $R = 0$ at a chosen wavelength.



(a) Varying δ, with $\lambda = \lambda_0$. (b) Varying λ, with $\delta = \pi/2$.

Figure 5.2: Optimal one-layer anti-reflective filter with $n_s = 1.52$.

Since $z_m \neq z_s$ in general, we must have $\cos\delta = 0$. The solution is simply $\delta = \pi/2$. In physical quantities, the solution is $n = \sqrt{n_m n_s}$, $d = \lambda/4n$.[1] In Fig. 5.1, the input optical impedance changes continuously from z_s to z_m along the thick path as we increase the thickness of the layer from zero to $\lambda/4n$. In Fig. 5.2(a) we see how R drops to zero at $\delta = \pi/2$, as expected.

5.1.2 Two-Layer Dielectric Anti-Reflective Filters

The major drawback of the one-layer filter from the previous section is that we are forced to use a material with impedance as close as possible to $\sqrt{z_m z_s}$. This may not be physically feasible, but as we shall see in this section, we can get around this

[1]This actually solves the problem in my report [8] that I used numerical methods to solve, once and for all.

by introducing another layer to the filter. Each layer corresponds to a circle centred on the real line in the impedance-plane, one that goes through z_s and one that goes through z_m. The circle that goes through z_s corresponds to the first layer and can lie on either the left or the right side of z_s. The circle that goes through z_m corresponds to the second layer and must lie on the left side of z_m. This is because a circle that lies on the right side of z_m corresponds to a material with impedance larger than z_m, or equivalently, refractive index smaller than one, which is very rare at optical wavelengths. All we need to do is to make sure that the circles cross at some point. This crossing point is the impedance after going through the first layer, which has to be transformed to z_m to get $R = 0$.

We can divide all possible solutions into three different classes by considering the characteristic impedance z_1 of the first layer. The three classes are summarised in Fig. 5.3.

(a) $z_1 < z_s$: The first circle lies to the left of z_s. We see this from Eq. (3.4) with $w = u = z_s$ and $z = z_1 < z_s$, which gives us that the centre of the circle lies to the left of z_s.

(b) $z_s < z_1 < \sqrt{z_m z_s}$: The first circle lies to the right of z_s, but does not encompass z_m. Complementary to (a), the condition $z_1 > z_s$ gives us that the centre of the circle lies to the right of z_s. The condition $z_1 < \sqrt{z_m z_s}$ ensures that the rightmost point of the circle is less than z_m.

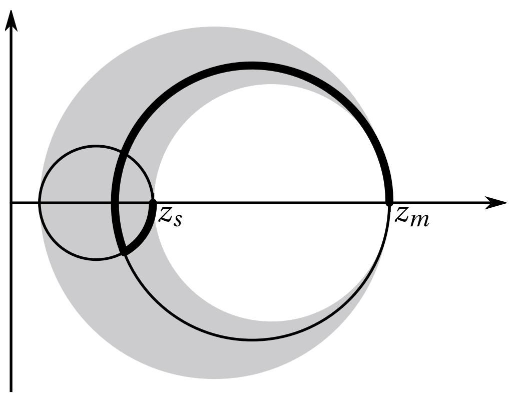
(c) $\sqrt{z_m z_s} < z_1 < z_m$: The first circle lies to the right of z_s, and encompasses z_m. As in (b), $z_1 > z_s$, so the centre of the circle lies to the right of z_s, but $z_1 > \sqrt{z_m z_s}$ ensures that the rightmost point of the circle is to the right of z_m.

The characteristic impedance z_2 of the second layer must be chosen so that the second circle intersects the first circle. The possible choices of z_2 are such that the second circle must lie in the shaded areas of Fig. 5.3(a)–(c). As in Fig. 5.1, the thick paths show how the input impedance changes as we gradually build up the filter, layer by layer. We could also consider the cases $z_1 = z_s$ and $z_1 = \sqrt{z_m z_s}$, however, they are trivial as they correspond to one of the circles having radius zero, and are thus equivalent to one-layer designs. In Fig. 5.3(d), the choices of z_1 and z_2 that give intersecting circles lie in the shaded areas.[2] These areas correspond to the three classes, as marked in the figure.

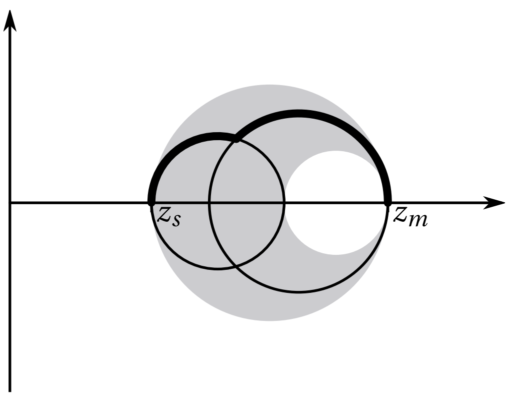
Note that the circles corresponding to the interior of the shaded areas in Fig. 5.3(d) intersect in two points, so there are actually two possible choices of layer thicknesses for these solutions, but for simplicity, we choose to explicitly display only one of them as thick paths. See Fig. 5.4 for an example of how the reflectance as a function of wavelength differs in the two cases of a class (a) solution.[3] We see that when $\delta_1 < \pi/2$,

[2]This diagram is equivalent to a *Schuster diagram* (see [7, Fig. 3.8]).

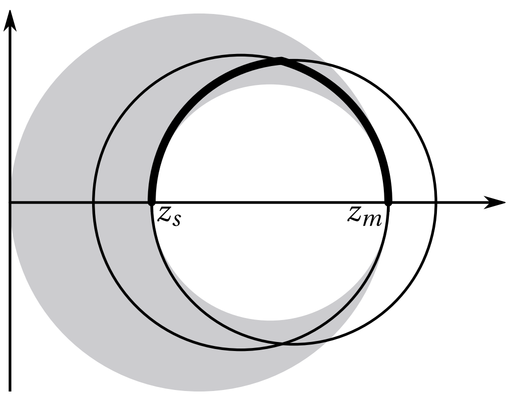
[3]The refractive indices of this example are taken from the example in [7, Sec. 3.2.2].



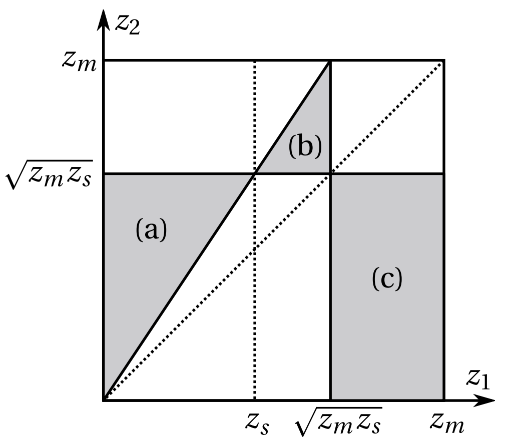
(a) $z_1 < z_s$.



(b) $z_s < z_1 < \sqrt{z_m z_s}$.



(c) $\sqrt{z_m z_s} < z_1 < z_m$.



(d) Solution regions.

Figure 5.3: Optimal two-layer anti-reflective filters.

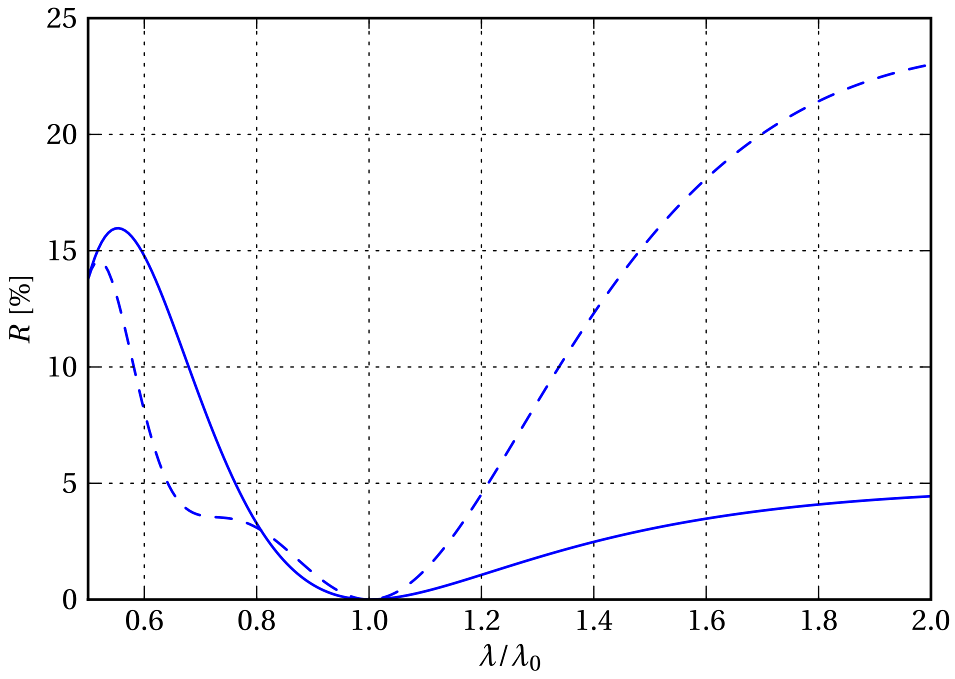


Figure 5.4: Reflectance of the two possible optimal two-layer filters with $n_1 = 2.20$, $n_2 = 1.38$ and $n_s = 1.52$. The solid curve corresponds to the solution where $\delta_1 < \pi/2$, while the dashed curve corresponds to $\pi/2 < \delta_1 < \pi$.

the reflectance curve is flatter around the minimum. Which one of the two cases we prefer depends on what we want the filter to do.

5.2 Multi-Wavelength Solution

Now we shift our attention to the design of filters that must satisfy some condition on a whole interval in the spectrum, not just a single wavelength as in the previous sections. Again, we focus on anti-reflective filters.

As an example, let us design a q-layer anti-reflective filter for the visible spectrum. According to [11], the visible spectrum spans the wavelength interval from 380 nm to 750 nm. We construct the least squares objective function f as shown in Sec. 2.3 by discretising the wavelength continuum into $m = 50$ equidistant wavelengths. We choose $\lambda_0 = 550$ nm so that the the part of the x-axis in the figures that correspond to visible light is approximately the interval $0.69 \leq \lambda/\lambda_0 \leq 1.37$.

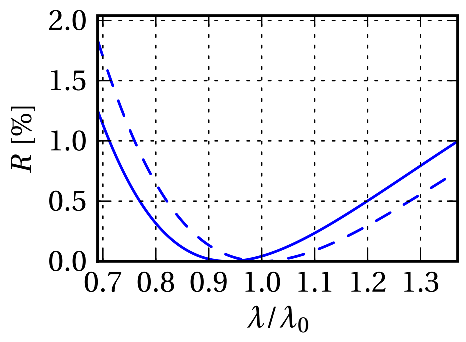


Figure 5.5: Optimal one-layer anti-reflective filter in the visible region. The dashed curve is the original filter, while the solid curve is the optimised filter.

5.2.1 One-Layer Dielectric Anti-Reflective Filters

In Fig. 5.2(b), we see how the reflectance of a one-layer filter that has $R = 0$ at wavelength λ_0 varies with wavelength. We need to find n and δ so that the objective function f is minimised. By using one of the optimisation methods in Sec. 4, we easily solve this problem. In Fig. 5.5 we have used the BFGS method of Sec. 4.2.1 to find a minimum of f. As a starting point, we have used the optimal single-wavelength one-layer design at $\lambda_0 = 550$ nm. The solution converged to $n = \sqrt{n_m n_s}$ and $\delta = 1.47265264 \approx 0.4688\pi$ after six iterations. The value of f decreased from $9.459 \cdot 10^{-4}$ to $7.241 \cdot 10^{-4}$.

5.2.2 Two-Layer Dielectric Anti-Reflective Filters

In Fig. 5.6 we have used the BFGS method to optimise a two-layer filter. We see that the reflectance is very low in the desired interval. As a starting point, we used $n_1 = 2.20$, $n_2 = 1.38$ and $\delta_1 = \delta_2 = \pi/2$. After 51 iterations, we get $n_1 = 1.3525586$, $n_2 = 1.12379604$ and $\delta_1 = \delta_2 = 1.44751467 \approx 0.4601\pi$. This reduces f from 0.243185 to $5.0425 \cdot 10^{-6}$.

5.2.3 Multi-Layer Dielectric Anti-Reflective Filters

The general case of q layers is very similar to the two-layer case, except that even better anti-reflective filters are possible. However, the optimisation methods used on them are the same, so they do not bring anything new to the discussion. We will not discuss the general case any further in this thesis.

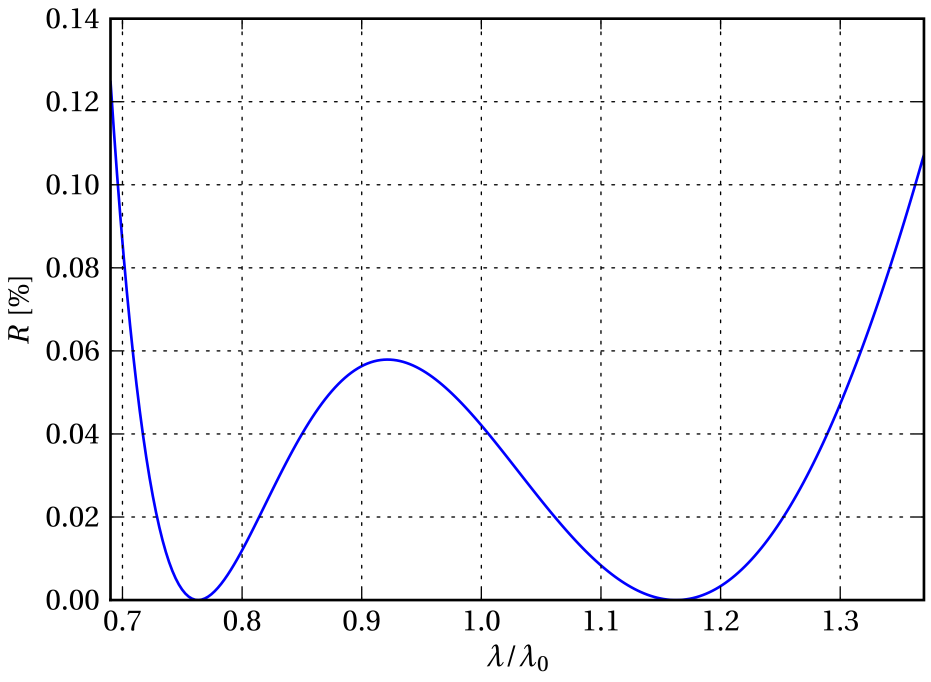


Figure 5.6: Optimal two-layer anti-reflective filter in the visible region.

Chapter 6

Software Design

We performed all the numerical calculations in Python (see [10]) using the SciPy numerics library (see [5]). We created the following classes:

- OpticalFilter: Consists of a list of Layer objects, an Optimiser object, and a CostFunction object.
- Layer: Stores the n and δ parameters that define the layer.
- CostFunction: Wrapper for the objective function and its gradient.
- Optimiser: This is the object that modifies the parameters in the Layer objects so that the function in the CostFunction object is minimised. If the Optimiser class is instantiated, it is set up to perform the BFGS method that is built into SciPy. It can also be subclassed so that other optimisation methods can be used:
  - NonlinCG: Performs our custom made nonlinear conjugate gradient methods. The variant is selected on instantiation.
  - BFGS: Performs our custom made BFGS method.

The BFGS method and nonlinear conjugate gradient method that came with SciPy were too complicated for our use, so we stripped them down to only contain the basic algorithms, and then extended them to perform the wanted optimisation methods. Most of the error checking and advanced options were not necessary for our case, and would only slow down the execution speed.

See App. A for the source code.

Chapter 7

Numerical Results

We test the optimisation methods on the two-layer anti-reflective filter from Sec. 5.2.2. We start at $n_1 = 2.20, n_2 = 1.38$ and $\delta_1 = \delta_2 = \pi/2$, and iterate until $\|\nabla f(x_k)\|_\infty < 10^{-10}$. The number of iterations, function and gradient evaluations are recorded in Tab. 7.1.

All the methods converge to the solution shown in Fig. 5.6, but as we see from the table, the work required to get there varies a lot. The BFGS method is clearly the best method. Of the nonlinear conjugate gradient methods, the Hager–Zhang method gives the best results.

In Fig. 7.1, we see that the solution found by OpenFilters gives a much higher reflectance in the visible wavelength interval than our methods (note that in this figure the y-axis is *not* in %). The reason for this is that OpenFilters constrains the allowed refractive indices, so that the optimisation never reaches low enough indices to get the result that we have obtained with our methods. The reflectance curve in OpenFilters before optimisation was identical to the curve calculated by our methods, so we are pretty confident that our methods are correct.

Method	Iterations	Function evaluations	Gradient evaluations
BFGS	51	52	52
FR	92	384	384
PR	100	379	379
PR+	120	460	460
FR–PR	108	425	425
HS	312	1403	1375
DY	92	389	389
HZ	72	275	275

Table 7.1: Optimisation of two-layer filter.

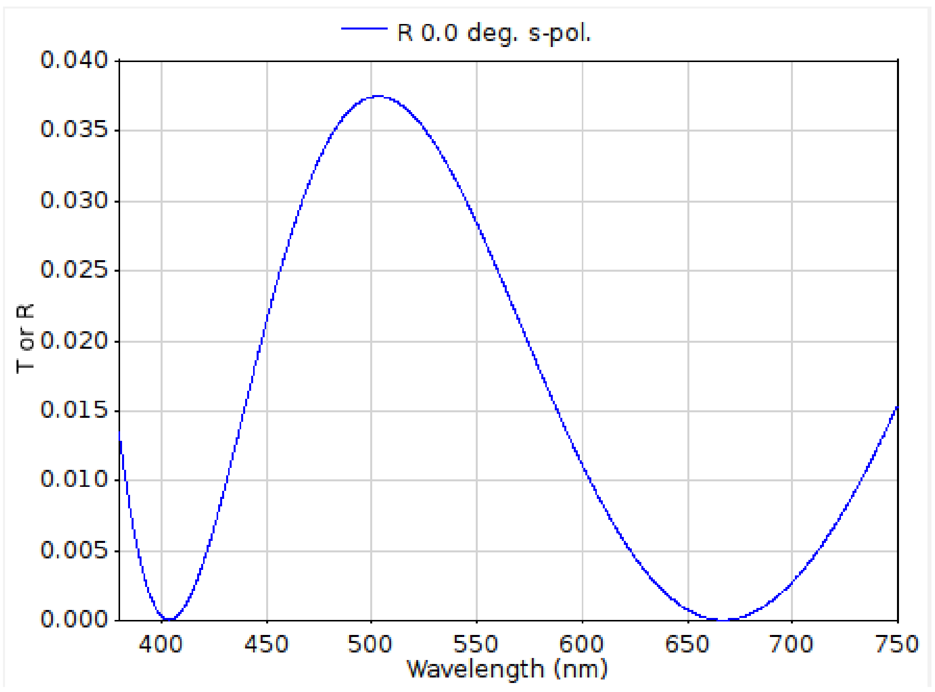


Figure 7.1: Screenshot of optimal two-layer filter found in OpenFilters.

Chapter 8

Conclusion and Future Work

In this thesis we have presented the physics behind optical interference filters and formulated the problem of designing them as an optimisation problem. Specifically, we have focused our attention on the design of dielectric anti-reflective filters, as these are very important in practice. We looked at the case where we want to minimise the reflectance at one specific wavelength, and at the case where we minimise the reflectance over a range of wavelengths.

By examining the geometry of the design problem, we found that we can solve the single-wavelength problem exactly without resorting to numerical methods. The multi-wavelength problem, on the other hand, we solved numerically by constructing a least squares objective function.

We presented two classes of line-search optimisation methods (quasi-Newton methods and nonlinear conjugate gradient methods), and applied variants of these to the design problem.

We found that the BFGS method is superior to all the nonlinear conjugate gradient methods tested on our two-layer test design problem. Of the nonlinear conjugate methods, we found that the variant by Hager and Zhang was the one that performed best. Finally, we compared our results with the results calculated by OpenFilters. OpenFilters is based on the Levenberg-Marquardt algorithm, which is a constrained optimisation algorithm. This makes comparison difficult, since our methods are based on unconstrained optimisation. Even though our methods found a better solution than the one OpenFilters found, the refractive indices found by our methods may be too low to be physically attainable.

In the future, it would be interesting to see how constrained optimisation methods, such as quadratic programming, and trust-region methods would work on the design problem. However, because of time constraints, we had to restrict our attention to the unconstrained methods presented in this thesis.

We have reached the goal of gaining geometric insight into the interesting problem of designing filters.

Bibliography

[1] David K. Cheng: *Field and Wave Electromagnetics.* Addison-Wesley, 1992.

[2] Y. H. Dai and Y. Yuan: *A Nonlinear Conjugate Gradient Method with a Strong Global Convergence Property.* SIAM Journal on Optimization, 10(1):177–182, 2000.

[3] John B. Fraleigh: *A First Course in Abstract Algebra.* Addison-Wesley, 2003.

[4] William W. Hager and Hongchao Zhang: *A New Conjugate Gradient Method with Guaranteed Descent and an Efficient Line Search.* SIAM Journal on Optimization, 16(1):170–192, 2005.

[5] Eric Jones, Travis Oliphant, Pearu Peterson, *et al.*: *SciPy: Open source scientific tools for Python*, 2001–. `http://www.scipy.org/`.

[6] Stéphane Larouche and Ludvik Martinu: *OpenFilters: open-source software for the design, optimization, and synthesis of optical filters.* Applied Optics, 47(13):219–230, 2008.

[7] H. Angus Macleod: *Thin-Film Optical Filters.* Institute of Physics Publishing, 2001.

[8] Håkon Marthinsen: *Optimisation on Lie Groups Applied to Optical Interference Filters.* Mandatory report written at NTNU, 2009.

[9] Jorge Nocedal and Stephen J. Wright: *Numerical Optimization.* Springer, 2006.

[10] Guido van Rossum and Fred L. Drake, Jr.: *Python Language Reference Manual.* Network Theory, 2006.

[11] Wikipedia: *Visible spectrum — Wikipedia, The Free Encyclopedia*, 2009. `http://en.wikipedia.org/w/index.php?title=Visible_spectrum&oldid=298452246`, [Online; accessed 24-June-2009].

Appendix A

Python Source Code

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""Optical interference filter design optimisation software."""

import numpy
from numpy import pi, eye, array, cos, sin, dot, Inf, isinf, outer, finfo, \
  arange, squeeze, zeros
from scipy.optimize import fmin_bfgs, fmin_l_bfgs_b, fmin_cg, line_search
from numpy.lib.scimath import sqrt
import scipy.optimize.linesearch
from matplotlib import rc
from matplotlib.pyplot import figure, axes, plot, xlabel, ylabel, grid, show, \
  xlim, ylim

# Constants
c0 = 299792458 # Speed of light in free space
mu0 = pi * 4e-7 # Magnetic permeability of free space
z_0 = c0 * mu0 # Impedance of free space
ym = 1 / (c0 * mu0) # Admittance of free space
ns = 1.52 # Refracive index of crown glass
ys = ns * ym # Admittance of crown glass
z_s = 1 / ys # Impedance of crown glass
I = eye(2) # Identity matrix

# Basis of parameter space
e1 = array([1.0, 0.0])
e2 = array([0.0, 1.0])

# Various needed vectors
vs = array([[1.0], [ys]])
vmm = array([ym, -1.0])
vmp = array([ym, 1.0])
```

```
# Machine epsilon and tolerance
eps = float(finfo(float).eps)
sqrteps = sqrt(eps)
tol = 100.0 * eps

def wrap_function(function, args):
    """Copied from SciPy."""
    ncalls = [0]
    def function_wrapper(var):
        """Copied from SciPy."""
        ncalls[0] += 1
        return function(var, *args)
    return ncalls, function_wrapper

def vecnorm(var, order = 2):
    """Copied from SciPy. order must not be equal to zero."""
    if order == Inf:
        return numpy.amax(abs(var))
    elif order == -Inf:
        return numpy.amin(abs(var))
    else:
        return numpy.sum(abs(var)**order, axis = 0)**(1.0 / order)

def tau(alpha):
    """tau map from the parameter space P to PSL(2)."""
    c_d = cos(alpha[1])
    is_d = 1J * sin(alpha[1])
    z = z_0 / alpha[0]
    return array([[c_d, z * is_d], [is_d / z, c_d]])

def dn_tau(alpha):
    """Calculate the derivative of the tau map wrt. n"""
    is_d = 1J * sin(alpha[1])
    return array([[0, -z_0 * is_d / (alpha[0]**2)], [is_d / z_0, 0]])

def ddelta_tau(alpha):
    """Calculate the derivative of the tau map wrt. delta"""
    ic_d = 1J * cos(alpha[1])
    s_d = sin(alpha[1])
    z = z_0 / alpha[0]
    return array([[-s_d, z * ic_d], [ic_d / z, -s_d]])

def nu(beta):
    """Multi-layer version of tau."""
    x = []
    for i in xrange(len(beta) / 2):
        alpha = beta[2 * i : 2 * (i + 1)]
        x.append(tau(alpha))
    return x
```

```
def grad_nu(beta):
    x = []
    for i in xrange(len(beta) / 2):
        alpha = beta[2 * i : 2 * (i + 1)]
        x.append(dn_tau(alpha))
        x.append(ddelta_tau(alpha))
    return x

def R(X):
    """Calculate the reflectance R."""
    tmp = dot(X, vs)
    rho = dot(vmm, tmp) / dot(vmp, tmp)
    return (rho * rho.conjugate()).real[0]

def psi(x):
    """Calculate product of matrices."""
    Y = I
    for X in x:
        Y = dot(X, Y)
    return Y

def grad_psi(beta):
    """Calculate the gradient of psi."""
    matrices = nu(beta)
    d_matrices = grad_nu(beta)
    num_layers = len(beta) / 2

    product_start = [I, I]
    product_end = [I, I]
    for i in xrange(num_layers - 1):
        tmp = dot(matrices[i], product_end[-1])
        product_end.append(tmp)
        product_end.append(tmp)
        tmp = dot(product_start[-1], matrices[num_layers - 1 - i])
        product_start.append(tmp)
        product_start.append(tmp)

    product_start.reverse()
    x = []
    for i in xrange(num_layers * 2):
        x.append(dot(dot(product_start[i], d_matrices[i]), product_end[i]))

    return array(x)

def S(beta, mu):
    """Sum of squares. mu is an array of scaling constants."""
    beta_mod = beta.copy()
    s = 0.0
    for mu_i in mu:
        # Scale deltas by mu_i and calculate reflectance
        beta_mod[1 : : 2] = beta[1 : : 2] * mu_i
        R_i = phi(beta_mod)
        s = s + R_i**2

    return 0.5 * s

def grad_S(beta, mu):
    """Gradient of S."""
    beta_mod = beta.copy()
    grad_s = zeros(len(beta))

    for mu_i in mu:
        # Scale deltas by mu_i and calculate reflectance
        beta_mod[1 : : 2] = beta[1 : : 2] * mu_i
        phi_i = phi(beta_mod)
        grad_phi_i = grad_phi(beta_mod)
        grad_phi_i[1 : : 2] = grad_phi_i[1 : : 2] * mu_i
        grad_s = grad_s + phi_i * grad_phi_i

    return grad_s

def phi(beta):
    """Objective function phi: P -> R."""
    return R(psi(nu(beta)))

def grad_phi(beta):
    """Gradient of phi in beta."""
    tmp = dot(psi(nu(beta)), vs)
    num = dot(vmm, tmp)
    den = dot(vmp, tmp)
    rho = num / den
    grad_rho = squeeze(dot((vmm - rho * vmp), dot(grad_psi(beta), vs))) / den
    return 2 * (grad_rho * rho.conjugate()).real

def cb(beta):
    """Callback for the optimisers. Is called after every iteration."""
    for i in xrange(len(beta)):
        # Ensure that we don't get negative values.
        beta[i] = max(beta[i], 0.0)

class Layer:
    """Class representing one layer in the filter."""
    def __init__(self, n = 1.4, delta = pi / 2):
        self.alpha = array([n, delta])

class Optimiser:
    """Minimising function wrapper."""
    def run(self, optFilt):
        """Default minimiser (SciPy BFGS)."""
        print "Selected SciPy-BFGS method."
        beta = fmin_bfgs(optFilt.costF.f, optFilt.parameters(), fprime = \
          optFilt.costF.grad_f, gtol = 1.0e-10, callback = cb)
        for i in xrange(len(optFilt.layers)):
            optFilt.layers[i].alpha = beta[2 * i : 2 * (i + 1)]

    def linesearch(self, f, fprime, xk, pk, gfk, old_fval, old_old_fval, \
      warnflag, c2 = 0.9):
        # These values are modified by the line search, even if it fails
        old_fval_backup = old_fval
        old_old_fval_backup = old_old_fval
        print "f:", f(xk)

        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
            scipy.optimize.linesearch.line_search(f, fprime, xk, pk, gfk, \
            old_fval, old_old_fval, c2)
        if alpha_k is None:  # line search failed try different one.
            alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                line_search(f, fprime, xk, pk, gfk, old_fval_backup, \
                old_old_fval_backup, c2)
            if alpha_k is None or alpha_k == 0:
                # This line search also failed to find a better solution.
                warnflag = 2
        return alpha_k, fc, gc, old_fval, old_old_fval, gfkp1, warnflag

class BFGS(Optimiser):
    """BFGS method."""
    def run(self, optFilt):
        """Modified version of the BFGS method provided by SciPy."""
        print "Selected BFGS method."
        x0 = optFilt.parameters()
        gtol = 1.0e-10
        norm = Inf
        disp = 1

        maxiter = len(x0) * 200

        func_calls, f = wrap_function(optFilt.costF.f, ())
        grad_calls, fprime = wrap_function(optFilt.costF.grad_f, ())

        gfk = fprime(x0)
        k = 0
        In = eye(len(x0), dtype = int)
        Hk = In
        old_fval = f(x0)
        old_old_fval = old_fval + 5000
```

```
xk = x0
warnflag = 0
gnorm = vecnorm(gfk, order = norm)

while (gnorm > gtol) and (k < maxiter):
    pk = -dot(Hk, gfk)

    # These values are modified by the line search, even if it fails
    old_fval_backup = old_fval
    old_old_fval_backup = old_old_fval

    alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
      scipy.optimize.linesearch.line_search(f, fprime, xk, pk, gfk, \
      old_fval, old_old_fval, c2 = 0.9)
    if alpha_k is None:  # line search failed try different one.
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
          line_search(f, fprime, xk, pk, gfk, old_fval_backup, \
          old_old_fval_backup, c2 = 0.9)
        if alpha_k is None or alpha_k == 0:
            # This line search also failed to find a better solution.
            warnflag = 2
            break

    if warnflag == 2:
        break
    sk = alpha_k * pk
    xk += sk
    if gfkp1 is None:
        gfkp1 = fprime(xk)

    yk = gfkp1 - gfk
    gfk = gfkp1
    cb(xk)
    k += 1
    gnorm = vecnorm(gfk, order = norm)
    if (gnorm <= gtol):
        break

    try: # this was handled in numeric, let it remaines for more safety
        rhok = 1.0 / dot(yk, sk)
    except ZeroDivisionError:
        rhok = 1000.0
        print "Divide-by-zero encountered: rhok assumed large"
    if isinf(rhok): # this is patch for numpy
        rhok = 1000.0
        print "Divide-by-zero encountered: rhok assumed large"

    if k == 1: # Modify initial Hessian approximation
        Hk = In * dot(yk, sk) / dot(yk, yk)
```

```
                tmp = rhok * outer(sk, yk)
                Hk = dot(dot(In - tmp, Hk), In - tmp.transpose()) + rhok * \
                  outer(sk, sk)

            if disp:
                fval = old_fval
            if warnflag == 2:
                if disp:
                    print "Warning: Desired error not necessarily achieved due to \
                      precision loss"
                    print "         Current function value: %f" % fval
                    print "         Iterations: %d" % k
                    print "         Function evaluations: %d" % func_calls[0]
                    print "         Gradient evaluations: %d" % grad_calls[0]

            elif k >= maxiter:
                warnflag = 1
                if disp:
                    print "Warning: Maximum number of iterations has been exceeded"
                    print "         Current function value: %f" % fval
                    print "         Iterations: %d" % k
                    print "         Function evaluations: %d" % func_calls[0]
                    print "         Gradient evaluations: %d" % grad_calls[0]
            else:
                if disp:
                    print "Optimization terminated successfully."
                    print "         Current function value: %f" % fval
                    print "         Iterations: %d" % k
                    print "         Function evaluations: %d" % func_calls[0]
                    print "         Gradient evaluations: %d" % grad_calls[0]

            for i in xrange(len(optFilt.layers)):
                optFilt.layers[i].alpha = xk[2 * i : 2 * (i + 1)]

class NonlinCG(Optimiser):
    """Nonlinear Conjugate Gradient methods."""
    def __init__(self, method = "PR+"):
        if method == "PR+":
            # Polak-Ribiere +
            print "Selected PR+ method."
            self.calc_beta = self.PRP
        elif method == "PR":
            # Polak-Ribiere
            print "Selected PR method."
            self.calc_beta = self.PR
        elif method == "HS":
            # Hestenes-Stiefel
            print "Selected HS method."
            self.calc_beta = self.HS
        elif method == "FR-PR":
            # Fletcher-Reeves-Polak-Ribiere
            print "Selected FR-PR method."
            self.calc_beta = self.FRPR
        elif method == "DY":
            # Dai-Yuan
            print "Selected DY method."
            self.calc_beta = self.DY
        elif method == "HZ":
            # Hager-Zhang
            print "Selected HZ method."
            self.calc_beta = self.HZ
        elif method == "FR":
            # Fletcher-Reeves
            print "Selected FR method."
            self.calc_beta = self.FR
        else:
            print "Invalid method: " + method + ". Defaulting to PR+."
            self.calc_beta = self.PRP

    def PRP(self, gfk, gfkp1, pk, k):
        return pymax(0, self.PR(gfk, gfkp1, pk, k))

    def PR(self, gfk, gfkp1, pk, k):
        return dot(gfkp1 - gfk, gfkp1) / dot(gfk, gfk)

    def HS(self, gfk, gfkp1, pk, k):
        yk = gfkp1 - gfk
        return dot(yk, gfkp1) / dot(pk, yk)

    def FRPR(self, gfk, gfkp1, pk, k):
        betaPR = self.PR(gfk, gfkp1, pk, k)
        if k >= 2:
            betaFR = self.FR(gfk, gfkp1, pk, k)
            if betaPR < -betaFR:
                return -betaFR
            elif betaPR > betaFR:
                return betaFR
            else:
                return betaPR
        else:
            return betaPR

    def DY(self, gfk, gfkp1, pk, k):
        return dot(gfkp1, gfkp1) / dot(pk, gfkp1 - gfk)

    def HZ(self, gfk, gfkp1, pk, k):
        yk = gfkp1 - gfk
        zk = 1.0 / dot(pk, yk)
        return dot(yk - 2.0 * pk * dot(yk, yk) * zk, gfkp1 * zk)

    def FR(self, gfk, gfkp1, pk, k):
        return dot(gfkp1, gfkp1) / dot(gfk, gfk)

    def run(self, optFilt):
        """Modified version of the nonlinear CG-method provided by SciPy."""
        x0 = optFilt.parameters()
        gtol = 1.0e-10
        norm = Inf
        maxiter = len(x0) * 200
        disp = 1

        func_calls, f = wrap_function(optFilt.costF.f, ())
        grad_calls, fprime = wrap_function(optFilt.costF.grad_f, ())

        gfk = fprime(x0)
        k = 0
        xk = x0
        old_fval = f(xk)
        old_old_fval = old_fval + 5000

        warnflag = 0
        pk = -gfk
        gnorm = vecnorm(gfk, order = norm)
        while (gnorm > gtol) and (k < maxiter):

            # These values are modified by the line search, even if it fails
            old_fval_backup = old_fval
            old_old_fval_backup = old_old_fval

            alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
              scipy.optimize.linesearch.line_search(f, fprime, xk, pk, gfk, \
              old_fval, old_old_fval, c2 = 0.1)
            if alpha_k is None:  # line search failed try different one.
                alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                  line_search(f, fprime, xk, pk, gfk, old_fval_backup, \
                  old_old_fval_backup, c2 = 0.1)
                if alpha_k is None or alpha_k == 0:
                    # This line search also failed to find a better solution.
                    warnflag = 2
                    break

            xk += alpha_k * pk
            if gfkp1 is None:
                gfkp1 = fprime(xk)
            beta_k = self.calc_beta(gfk, gfkp1, pk, k)
            pk = -gfkp1 + beta_k * pk
            gfk = gfkp1
            gnorm = vecnorm(gfk, order = norm)
            cb(xk)
            k += 1

        if disp:
            fval = old_fval
        if warnflag == 2:
            if disp:
                print "Warning: Desired error not necessarily achieved due to \
                  precision loss"
                print "         Current function value: %f" % fval
                print "         Iterations: %d" % k
                print "         Function evaluations: %d" % func_calls[0]
                print "         Gradient evaluations: %d" % grad_calls[0]

        elif k >= maxiter:
            warnflag = 1
            if disp:
                print "Warning: Maximum number of iterations has been exceeded"
                print "         Current function value: %f" % fval
                print "         Iterations: %d" % k
                print "         Function evaluations: %d" % func_calls[0]
                print "         Gradient evaluations: %d" % grad_calls[0]
        else:
            if disp:
                print "Optimization terminated successfully."
                print "         Current function value: %f" % fval
                print "         Iterations: %d" % k
                print "         Function evaluations: %d" % func_calls[0]
                print "         Gradient evaluations: %d" % grad_calls[0]

        for i in xrange(len(optFilt.layers)):
            optFilt.layers[i].alpha = xk[2 * i : 2 * (i + 1)]

class CostFunction:
    """The cost function and its gradient wrapper."""
    def __init__(self, f = phi, grad_f = grad_phi):
        self.f = f
        self.grad_f = grad_f

class OpticalFilter:
    """The filter, consisting of one or more layers."""
    def __init__(self, layers = [Layer()], optimiser = Optimiser(), costF = \
      CostFunction()):
        """Initialise the filter with layers, one standard layer by default."""
        self.layers = layers
        self.optimiser = optimiser
        self.costF = costF
    def addLayer(self, layer = Layer()):
        """Add a layer to the existing ones."""
        self.layers.append(layer)

    def optimise(self):
        """Minimise the cost function."""
        self.optimiser.run(self)

    def parameters(self):
        """Return the parameters of the filter."""
        tmp = []
        for layer in self.layers:
            tmp.extend(layer.alpha.tolist())
        return array(tmp)

wl_low = 0.69
wl_high = 1.37
wl_number = 50
wl_div = (wl_high - wl_low) / (wl_number - 1)
wl_factors = arange(wl_low, wl_high + eps, wl_div)
mu = 1.0 / wl_factors

#optFilt = OpticalFilter(layers = [Layer(2.2, pi * 0.5)], costF = \
#  CostFunction(lambda x: S(x, mu), lambda x: grad_S(x, mu)), optimiser = \
#  NonlinCG("HZ"))
optFilt = OpticalFilter(layers = [Layer(2.2, pi * 0.5)], costF = \
  CostFunction(lambda x: S(x, mu), lambda x: grad_S(x, mu)), optimiser = \
  BFGS())
optFilt.addLayer(Layer(1.38, pi * 0.5))
optFilt.addLayer(Layer(1.5, pi * 0.5))

optFilt.optimise()
```