NAZARBAYEV UNIVERSITY

School of Engineering and Digital Sciences

Bachelor of Engineering in
Mechanical and Aerospace Engineering

Electromagnetic Wave Interactions with Multiple Arbitrarily Shaped Nanotubes

(Final Capstone Project Report)

by

Abzal Aznabayev, Aidana Faizulla, and Ravil Ashirmametov

Lead Supervisor: Prof. Konstantinos Kostas

Co-Supervisor: Prof. Constantinos Valagiannopoulos

April, 2024

Declaration

We, Abzal Aznabayev, Aidana Faizulla, and Ravil Ashirmametov, hereby declare that this report, entitled "Electromagnetic Wave Interactions with Multiple Arbitrarily Shaped Nanotubes" is the result of our own project work except for quotations and citations which have been duly acknowledged. We also declare that it has not been previously or concurrently submitted for any other degree at Nazarbayev University or elsewhere.

Signature:

Name: Ravil Ashirmametov
Date: April 28, 2024

Signature:

Name: Abzal Aznabayev
Date: April 28, 2024

Signature:

Name: Aidana Faizulla
Date: April 28, 2024

Abstract

This work studies the interaction of electromagnetic waves with arbitrarily shaped nanotubes. The research starts with the analysis of cylindrical-shaped nanotubes, due to their simplicity, further replacing them with arbitrary-shaped nanotubes for the same conditions. In this work, the objective is to increase the concentration of the electric field inside pairs of nanotubes by optimizing their shape appropriately. The Isogeometric-Analysis-based Boundary Element Method (IGABEM) approach was used to achieve the objective in pair with optimization approaches involving gradient-based and guided random search methods. This combination of mathematical programming with an appropriate parametric model, for the accurate modeling of free-form boundaries, and the IGABEM method for the shapes' performance evaluation was successful in identifying the optimal shape of nanotubes. So far obtained results show a substantial increase in the concentration of electric field inside the shape-optimized nanotubes. A field concentration increase of more than 20 times compared to circular nanotubes was achieved and minimization showed 57 times decrease. Furthermore, the optimized shapes showed a better field concentration over a wide range of EM wave orientations and distances between nanotubes. This result demonstrates the robustness of the obtained results and their potential applicability in a wide range of applications.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background Information

Nanotubes refer to tube structures with diameters measured in the nano-scale. They can be visualized as thin 2D sheets rolled up forming a tube. The interest in nano-structures has been developing since 1991 when Iijima introduced a manufacturing method for Carbon Nanotubes (CNTs) [2]. Carbon nanotubes are the first generation of nanotubes whose properties sparked the interest in nanotubes exploration. In the case of CNTs, the nanotubes possess amazing mechanical strength, thermal conductivity, and other physical properties that attract scientists' and engineers' attention [3]. Due to their high tensile strength and Young's modulus, they are commonly used in composite materials. Moreover, they found application in coatings and films, improving the mechanical properties [4]. Another important property of carbon nanotubes is their electrical conductivity, compared to other materials. Based on this property nanotubes find a wide range of applications in electronics and other spheres, such as electrochemical devices, field emission devices, and nano-scale electric devices [5]. Carbon nanotubes are not the only nanotubes being studied globally. Boron nitride, gallium nitride, silicon, and carbon-titanium dioxide nanotubes are among the other materials considered in nanotubes research [6, 7].

One of the intriguing aspects of nanotubes is that their properties are shape-dependent, which gives additional degrees of freedom in research on nanotubes. Depending on its shape, a nanotube might exhibit metallic or semiconducting behavior[8]. Based on their shape and surface conductivity, significantly varying behaviors can be exhibited by such nanotubes. For most applications, only circular shapes are considered as their analysis and fabrication are significantly simpler than arbitrarily shaped ones. However, not all nanotubes are perfectly circular, for example, carbon nanotubes can be single-walled and multi-walled with defects and discontinuities present [9]. Thus, geometry considerations offer a new avenue for research in nanotube-properties modifications. Specifically, in our case, optimization of the nanotube shape with the aim of maximizing or minimizing certain behaviors. This perspective is researched in this work. The main objective of shape optimization in this project is to maximize the concentration of the electromagnetic (EM) field in the domain region enclosed by two nanotubes. The minimization possibility of the EM wave concentration inside nanotubes is also considered in this project. The two nanotube wave interaction and wave interference problem is the spotlight of this project work. The EM wave interaction with two nanotubes is an interesting topic, which

can be used to form the identified optimal shapes, by extending this study to a larger array of nanotubes, i.e. the shapes reported in this work could be also considered with meta-material boundaries. For example, Kaili Jiang et al. report different applications and modified mechanical properties for carbon nanotube arrays [10]. EM wave interaction of arbitrarily shaped nanotubes is not a well-researched area, although works have studied conventional cases in pertinent literature, which motivated us to study the case examined in this project work. In this work, the study is constituted by a pair of arbitrarily shaped nanotubes subjected to an EM wave field with transverse magnetic polarization. This means that the electric field is parallel to the z-axis (in and out of the plane), while the magnetic field is propagated through the xy plane (across the cross-section of nanotubes). The wave hits the nanotubes at variable angles and the distance between nanotubes is also changed. The distance and cross-sectional area of the nanotubes are normalized with respect to the wavelength. As an initial part of the project, Maxwell's equations were solved and the interference interaction of nanotubes was observed.

The IsoGeometric-Analysis-based Boundary Element Method (IGABEM) is the main tool used to compute the field along the nanotube boundaries as well as the field within the region bounded by them. Computer simulations using a number of parameters are performed in order to determine the electric field concentration inside the nanotubes. The IGABEM approach uses the same basis functions for representations of the geometry and solution field, which results in several advantages in engineering analysis. In brief, it maintains the accuracy of the geometrical representation, eliminates the need for an approximate mesh, and significantly reduces the computational effort needed to achieve a certain level of accuracy as high-degree polynomials are employed in the approximation of the field quantity [11]. Hughes et al. introduced the IsoGeometric-Analysis for the Finite Element Method in engineering analysis[12]. The IGA method was later coupled with the Boundary Element Method [13]. In this project work, this approach will be employed to all relevant quantities for the EM wave interaction with nanotubes in a 2D case.

1.2 Applications

The maximization of the electric field inside the nanotube can be proven beneficial for a wide range of applications in communications and data transfer. Shiva Piltan and Dan Sievenpiper proposed the replacement of semiconductors with vacuum/gas and use photons to transfer information[14]. This method provides faster information transfer with less power. The meta-material of the nanotubes with optimized shape could be also used to enhance this transfer. This can be explained due to the fact that in certain conditions nanotubes can exhibit semiconducting properties.

Furthermore, concentrating the electric field has an application in improving wireless signal transmission. For example, the optimized shape could be used for antennas in a

router and other signal transmission devices. In addition to this application, the meta-material could be used to secure the signal transmission. Lu Lv et al. considered using interference in signal transmission to enhance communication and secure the channel from eavesdroppers. Hence, the meta-material could be used in this application too [15].

Another possible application is gas sensors that utilize carbon nanotubes. Yang in 2021 researched the possibility of using fractal geometry in the gas nanotube sensors to enhance electric field. In this way, a combination of fractal geometry and shape optimization can result in better sensitivity of the sensors to gases[16].

S. Zhang, Y. Wang, and others in 2018 investigated the usage of carbon nanotubes as a coating for electrodes used for capacitive deionization [17]. Results showed that using carbon nanotubes improves the electrochemical and absorptive properties of electrodes. Shape-optimized nanotubes can perform better in terms of accumulation and storage of electric field and current. These electrodes show promising applications in water treatment and desalination purposes. Bocharov and Eletskii in 2013 proposed an array of carbon nanotubes on a cathode used in field emission devices to increase electric concentration near the tips of the carbon nanotubes [18].

In 2019 S. Zhang, N. Nguyen, and others assessed the feasibility of creating macro-scale conductors made of carbon nanotubes. They highlighted the excellent properties of CNTs and compared them with copper [19]. Improved electrical properties and the ability of CNT to concentrate higher currents can yield to a mass production of CNT-based materials that can find an application in transistors, sensors, and electrical wiring.

In addition to that, metamaterial made of shape-optimized nanotubes can be used to store and accumulate excessive electromagnetic energy. Placing a sheet made of metamaterial in front of a transverse magnetic wave can both shield from radiation and store electric charge inside.

Chapter 2

Methodology

2.1 Problem Definition

There are two arbitrarily shaped nanotubes placed at a certain distance d from each other. Both nanotubes have identical constant complex surface conductivity σ and identical area. The Transverse Magnetic (TM) wave is incident on the nanotubes at an angle α; see Fig. 2.1.

The problem is being defined using the TM polarization wave with a wavelength λ. TM, as mentioned before, corresponds to a transverse magnetic wave mode, called E waves, where the magnetic field is perpendicular to the direction of wave propagation (in xy plane). In the case of TE wave (transverse electric), the magnetic field is parallel to the direction of propagation. The electric field in the TM wave is parallel to the z-axis. The main attention is being paid to the xy cross-section as the length of the nanotube compared to its cross-sectional dimensions is much higher. This capstone project started by studying the problem for two interacting circular nanotubes to identify the regions of interest, i.e. wave angle and distance, between the tube centroids, are changed to find layouts, at which concentration values can be potentially minimized or maximized. This initial study is performed for two nanotube areas and two surface conductivity values, resulting in four colormaps; see Fig 4.2 and 4.3 in the Results chapter, section 4.2. An extended study using a significantly increased range of areas, distance, as well as surface conductivity values is performed at a later stage to fully cover the space of possible designs; see also section 4.2.1.

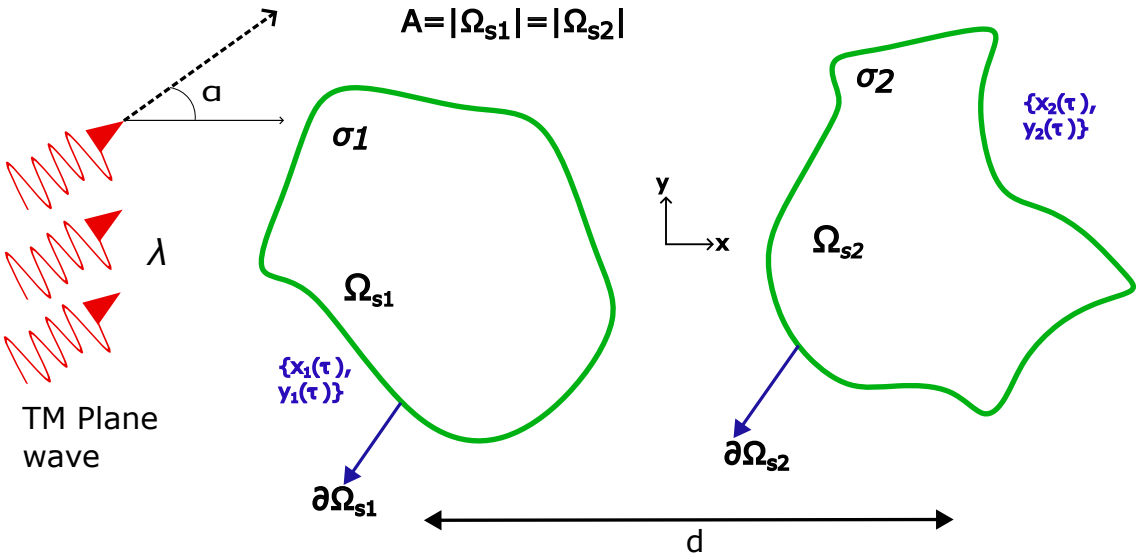


Figure 2.1: The Problem Definition

In the initial analysis, the relevant parameters are normalized with the wavelength:

- Wavelength $\lambda = 1$ unit of length
- Normalized area $\frac{A_1}{\lambda^2} = 1.5$ and $\frac{A_2}{\lambda^2} = 5$
- Radius is $r = \sqrt{\frac{A\lambda}{\pi}}$
- Complex surface conductivity $\sigma_1 = 0.01 + 0.01i$ and $\sigma_2 = 0.005 - 0.005i$

Next, the nanotubes interaction and electric field concentration are formulated and analyzed.

The electric field outside the nanotubes, $E_{out}(x, y)$, comprises a background term $E_{back}(x, y)$ and a scattering term $E_{scat}(x, y)$. Electric fields inside the nanotubes are $E_{1,in}(x, y)$ and $E_{2,in}(x, y)$, respectively.

The scattering field is given by the following equation:

$$E_{z,scat}(x,y) = -ik_0\eta_0 \oint_{(\partial\omega_{s1})} G(x_1, y_1, X_1(l), Y_1(l))K_1(l)dl - ik_0\eta_0 \oint_{(\partial\omega_{s2})} G(x_1, y_1, X_2(l), Y_2(l))K_2(l)dl \tag{2.1}$$

where:

$\eta_0 = 120\pi$ Ohm - wave impedance into vacuum

$K_1(l)$ and $K_2(l)$ are axial currents:

$$\mathbf{K_1} = K_1\hat{\mathbf{z}} = -\sigma_1\hat{\mathbf{n}} \times [\hat{\mathbf{n}} \times \hat{\mathbf{E}}_{out}] = -\sigma_1\hat{\mathbf{n}} \times [\hat{\mathbf{n}} \times \hat{\mathbf{E}}_{1,in}] \tag{2.2}$$

$$\mathbf{K_2} = K_2\hat{\mathbf{z}} = -\sigma_2\hat{\mathbf{n}} \times [\hat{\mathbf{n}} \times \hat{\mathbf{E}}_{out}] = -\sigma_2\hat{\mathbf{n}} \times [\hat{\mathbf{n}} \times \hat{\mathbf{E}}_{2,in}] \tag{2.3}$$

$G(x, y, X_1(l), Y_1(l))$ and $G(x, y, X_2(l), Y_2(l))$ are Green's function for first and second nanotube. Each equal to:

$$G(x, y, X, Y) = -\frac{i}{4}H_0(k_0\sqrt{(x - X)^2 + (y - Y)^2} \tag{2.4}$$

where:

H_0 is the Hankel function of zero order and second kind.

$k_0 = 2\pi/\lambda$ - is the wavenumber into free space.

In Figure 2.1, the cross-section of the arbitrary curves is Ω_{s1} and Ω_{s2} with $\partial\Omega_{s1}$ and $\partial\Omega_{s2}$ being the boundary. The boundary is parametrically represented as $\partial\Omega_s = \{x(\tau), y(\tau)\}$ for $\tau \in I \subset \mathbb{R}$. Hence, the parametric description of the nanotubes' boundary is assumed to be: $\{x_1(\tau), y_1(\tau)\}$ and $\{x_2(\tau), y_2(\tau)\}$ for $0 < \tau < 1$.

Boundary Integral Equation for the nanotubes:

$$\begin{aligned} &E_{1,in}(x_1(\tau), y_1(\tau)) + \\ &ik_0\sigma_1\eta_0 \int_0^1 G(x_1(\tau), y_1(\tau), x_1(t), y_1(t)) E_{1,in}(x_1(\tau), y_1(\tau)) \sqrt{[\dot{x}_1(t)]^2 + [\dot{y}_1(t)]^2} dt + \\ &ik_0\sigma_2\eta_0 \int_0^1 G(x_1(\tau), y_1(\tau), x_2(t), y_2(t)) E_{2,in}(x_1(\tau), y_1(\tau)) \sqrt{[\dot{x}_2(t)]^2 + [\dot{y}_2(t)]^2} dt = \\ &E_{back}(x_1(\tau), y_1(\tau)) \end{aligned} \tag{2.5}$$

$$\begin{aligned} &E_{2,in}(x_2(\tau), y_2(\tau)) + \\ &ik_0\sigma_1\eta_0 \int_0^1 G(x_2(\tau), y_2(\tau), x_1(t), y_1(t)) E_{1,in}(x_2(\tau), y_2(\tau)) \sqrt{[\dot{x}_1(t)]^2 + [\dot{y}_1(t)]^2} dt + \\ &ik_0\sigma_2\eta_0 \int_0^1 G(x_2(\tau), y_2(\tau), x_2(t), y_2(t)) E_{2,in}(x_2(\tau), y_2(\tau)) \sqrt{[\dot{x}_2(t)]^2 + [\dot{y}_2(t)]^2} dt = \\ &E_{back}(x_2(\tau), y_2(\tau)) \end{aligned} \tag{2.6}$$

From Equations 2.5 and 2.6, the electric field on the two boundaries $E_{1,in}(x_1(\tau), y_1(\tau))$ and $E_{2,in}(x_2(\tau), y_2(\tau))$, can be determined. Using the determined quantity, the axial electric field $E(x(\tau), y(\tau))$ inside and outside the nanotube can be evaluated. These values are achieved by evaluating Equation 2.1. Specifically, by using the calculated values of the electric field on nanotubes boundary in $E(x(\tau), y(\tau)) = E_{back}(x(\tau), y(\tau)) + E_{scat}(x(\tau), y(\tau))$.

The background field of the TM with unitary amplitude forming an angle α with x axis:

$$E_{z,back}(x, y) = \exp(-ik_0(xcos(\alpha), ysin(\alpha))) \tag{2.7}$$

The objective of this work is to maximize or minimize the electric field concentrations inside a single nanotube in pair with a circular one and inside a pair of arbitrarily shaped nanotubes. Electric field concentration inside each nanotube is equal to:

$$Q = \frac{1}{A} \iint_{(A)} |E_{z,in}(x, y)|^2 dxdy \tag{2.8}$$

where:

A - The cross-sectional area of the nanotube

The normalized electric field concentration can be evaluated by:

$$\frac{Q_1 + Q_2}{2 \cdot Q_c} \tag{2.9}$$

where:

Q_1 and Q_2 - the electric field concentrations of the two nanotubes having a distance d to each other.

Q_c - circular nanotube electric field concentration of the same cross-sectional area at an infinite distance.

This normalization is performed to determine how optimized shapes can outperform circular ones in either maximization or minimization of the electric field.

2.2 Parametric model

Non-Uniform Rational B-Splines (NURBS) consitute a mathematical representation which uses rational B-splines to represent geometries. NURBS can potentially represent any type of geometry, 1D, 2D, 3D ... ND (functions, curves, surfaces, volumes, etc.). The type of geometry they represent depends on the type of control values and the parametric domain. For example, the $E(t)$ is a 1D NURBS representation of the electrical field. NURBS is a flexible and accurate method of representing geometry. Hence, NURBS are widely used in many areas starting from engineering design analysis and ending with game development [20]. A NURBS curve requires control points and a knot vector to be determined. In Figure 2.2, an example of a NURBS curve representing a free-form closed curve with 10 control points is given. NURBS are used for representing the geometry of the nanotubes as well as the solution in the IGABEM formulation. For the generation of the corresponding spline space, the so-called knot vector should be created as an ordered set of parameters $I = \{\tau_0, \tau_1, ..., \tau_{p+m+1}\}$. Where $m+1$ is the number of the basis functions and $p+1$ is their order, with p being the polynomial degree, and τ_i is the ith knot. Equation 2.10 defines the zeroth order B-spline.

$$N_{i,0}(\tau) = \begin{cases} 1, & \text{if } \tau_i \leq \tau \leq \tau_{i+1} \\ 0, & \text{Otherwise} \end{cases} \tag{2.10}$$

Then, the higher-order B-splines can be constructed using a recursive relation as shown in Equation 2.11.

$$N_{i,p}(\tau) = \frac{\tau - \tau_i}{\tau_{i+p} - \tau_i} N_{i,p-1}(\tau) + \frac{\tau_{i+p+1} - \tau}{\tau_{i+p+1} - \tau_{i+1}} N_{i+1,p-1}(\tau) \tag{2.11}$$

Finally, Equation 2.13 describes the NURBS curve.

$$r(\tau) \coloneqq \sum_{i=0}^{n} b_i N_{i,p}(\tau), \tau \in [t_p, t_m + 1], \tag{2.12}$$

where $N_{i,p}(\tau)$ are the B-spline basis functions defined over I and b_i are the control points. With the NURBS weights applied, the expression of NURBS basis functions turns to:

$$R_{i,p}(\tau) \coloneqq \sum_{i=0}^{n} b_i \frac{w_i N_{i,p}(\tau)}{\sum_{j=0}^{n} w_j N_{j,p}(\tau)} N_{i,p}(\tau), \tag{2.13}$$

where w_i is the ith weight value. If all weights are equal to 1, the NURBS become a B-spline. $R_{i,p}$ denotes the ith rational basis function of degree p.

The optimization of nanotube shapes requires addressing several challenges. One of them is the potentially high number of control points for complicated shapes which raises the problem of optimization in a high-dimensional space when control point coordinates are employed as design parameters. Furthermore, the resulting shape should not possess unwanted properties such as self-intersections. These issues can be adressed with a set of complicated constraints which however further hinder shape optimization. The NURBS-based parametric models developed in this project work provide a solution to both of these issues. Specifically, they employ a relatively small set of high-level parameters that address the issue of high-dimensional space. At the same time, the employed parametric models divide the space into sectors in which the control points are located and hence self-intersections are avoided.

The parametric model accepts a $2 \times N$ matrix of parameters, renging from 0 to 1, which determine the polar coordinates of the employed N control points in the representation of the closed curve correspomding to the nanotube's boundary. At the same time a maximum allowable polar radius is also provided. These N points are bound to lie within an appropriate sector of the circle. Both parameters have a constraint between 0 and 1, such that the maximum value for the radius is the maximum radius given, and the maximum angle

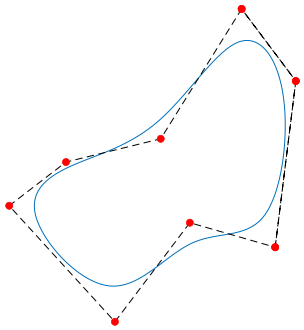


Figure 2.2: NURBS example

is 360 degrees divided by the number of parameters. The control points, $\{r_i, \phi_o\}_{i=0}^{m}$, are derived from the parameters as follows, the radius is given by:

$$r_i = r_{max}(0.99p_1 + 0.01), \tag{2.14}$$

and the angle is given by:

$$\phi_i = (p_2 + (p_i - 1))/\frac{2\pi}{n}. \tag{2.15}$$

Finally Eq. 2.16 is used for writing the ith control point (b_i) in homogeneous Cartesian coordinates.

$$b_i = \begin{bmatrix} r_i w \cos(\phi_i + \frac{2\pi i}{n}) \\ r_i w \sin(\phi_i + \frac{2\pi i}{n}) \\ w \end{bmatrix} \tag{2.16}$$

We only use a 2D representation as the length of the nanotube compared to its cross-sectional area is assumed to be infinitely long. Thus analysis performed in XY plane (2D) is suitable for any region of nanotubes' length.

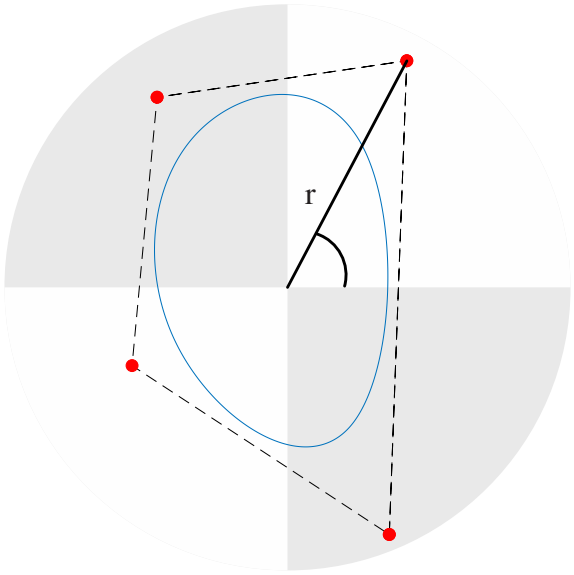


Figure 2.3: Parametric Model Constructing a Closed Curve

2.2.1 Parametric Model Version 2

The second version of the parametric model improves the previous version parametric model by addressing two issues that limit the performance of optimizations: non-linear constraints (related to nanotube's area)

and the number of optimizers that can incorporate non-linear constraints. Previously, during optimization, an area constraint was used to satisfy the requirement of given constant nanotube areas for the considered cases. This non-linear constraint makes the optimization problem harder and additionally limits the number of optimization algorithms that can be used. Hence, a number of global optimization algorithms could not be used and the optimization time was negatively affected by the existence of the non-linear constraint. The area constraint is explained in Section 3.4. The second version of the parametric model solves these problems by integrating the area constraint in the parametric model. It includes the area constraint by scaling the curve using the maximum radius described in Equation 2.14. The generated curve's area is calculated and then the difference between the calculated area and the desired area is found. This difference is used in the Newton-Raphson method to find the radius that would give the required area. Hence, the shape is not changed but scaled to give the right area.

2.2.2 Parametric Model Version 3

After development and successive simulations performed using the first and second parametric models, the team and supervisor decided to develop an additional parametric model to permit the generation of more complicated shapes with less parameters and simplify the procedure of comparing the resulting shapes with reference values coming from cylindrical tubes. Compared to the first and second parametric models the biggest difference of this model is that it is symmetric around the x-axis. This increases computational efficiency in terms of used parameters as their number is halved for the same shape complexity. Previously, to represent a 10 DOFs figure, 20 parameters were required (10 for radius + 10 for angle), now in turn, a 10 DOFs figure requires 10 parameters (5 for radius + 5 for angle) and it is mirrored appropriately around the x-axis.

Another huge advantage of this model is that it allows precise control of the distance between the curves. In previous versions when the shape of the nanotube was adjusted its centroid slowly traveled up, down, left, or right. Due to this issue, a separate procedure was required to acquire the corresponding reference value. In this symmetric parametric model, we can just calculate the x-coordinate of the curve's centroid using the new function CurveCentroid. Obviously, the y-coordinate of the centroid is not needed since by construction the curve is symmetric around the x-axis and therefore the y-coordinate of the centroid is always 0. With these tools, functions, and models we can precisely control the placement of the nanotubes, such that it will match the distance used in the calculation of precomputed reference values.

Also, it is important to mention here that the new parametric model adopted the sector division model from the previous models, as well as the incorporation of the non-linear area constraint from the second model. Overall, this model is faster and allows us to generate more complicated shapes with half of the parameters compared to previous models.

2.3 IsoGeometric-Analysis-based Boundary Element Method

The nature of nanotubes makes the IsoGeometric-Analysis-based Boundary Element Method (IGABEM) the best method for estimation of the solution of the problem presented in section 2.1. The Boundary Element Method (BEM) is suitable for our problem since the problem involves an unbounded domain. Using the Finite Element Method (FEM) would require discretizing the whole planar domain. Furthermore, employing an IGA approach brings the benefits explained in Section 1. The IGABEM employs the same spline basis to represent the geometry as well as the solution field. This means that the same basis functions are used to describe the geometry and solution of the problem. Hence, there is no need to approximate the geometry of

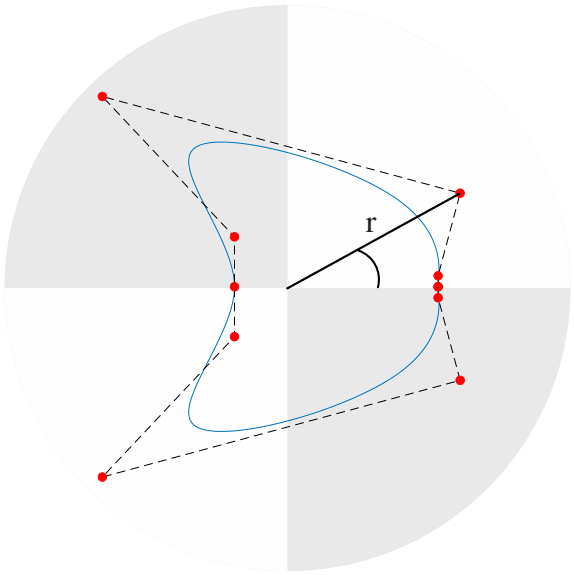


Figure 2.4: Parametric Model Version 3

the nanotube by creating a mesh to estimate the solution. This approach greatly reduces the computational time and gives better accuracy at lower degrees of freedom employed [21].

The combined power of IGABEM and NURBS is utilized in this project to estimate the solution to the described problem and then optimize the shape of the nanotubes to achieve a better concentration of the electric field. Similar to the expression of geometry using NURBS basis functions in Equation 2.12 and 2.13, the solution field can be expressed the same way. The unknown quantity in the problem is the electric field $E(x, y)$ and its approximation $\tilde{E}(x, y)$ in the corresponding spline space can be written as:

$$E(x(\tau), y(\tau)) = \tilde{E}(x(\tau), y(\tau)) \coloneqq \sum_{i=0}^{n} e_i R_{i,p}(\tau) \tag{2.17}$$

When refinements of the approximation are needed, a finite number of nested splines spaces can be considered, i.e., $S_k(I^{(0)} \subset ... \subset S_k(I^{(m-1)}) \subset S_k(I^{(m)})$, where m denotes the number of knots inserted in I during the refinement process [11] (not to be confused with $m + 1$ used to denote the number of basis functions used in section 2.2). The projection of $E(x, y)$ on these spline spaces is shown in Equation 2.18.

$$\tilde{E}(x(\tau), y(\tau)) \coloneqq \sum_{i=0}^{n+m} e_i R_{i,p}^{(m)}(\tau) \tag{2.18}$$

Then, the new basis function can be used in Equation 2.5 and Equation 2.6 resulting in the following equations:

$$\begin{aligned} &\sum_{i=0}^{n_1+m} e_{1,i} R_{i,p}(\tau) + \\ ik_0\sigma_1\eta_0 &\sum_{i=0}^{n1+m} e_{1,i} \int_0^1 G(x_1(\tau), y_1(\tau), x_1(t), y_1(t)) R_{i,p}(t) \sqrt{[\dot{x}_1(t)]^2 + [\dot{y}_1(t)]^2} dt + \\ ik_0\sigma_2\eta_0 &\sum_{i=0}^{n2+m} e_{2,i} \int_0^1 G(x_1(\tau), y_1(\tau), x_2(t), y_2(t)) R_{i,p}(t) \sqrt{[\dot{x}_2(t)]^2 + [\dot{y}_2(t)]^2} dt = \\ &E_{back}(x_1(\tau), y_1(\tau)) \end{aligned} \tag{2.19}$$

$$\begin{aligned} &\sum_{i=0}^{n_2+m} e_{2,i} R_{i,p}(\tau) + \\ ik_0\sigma_1\eta_0 &\sum_{i=0}^{n2+m} e_{2,i} \int_0^1 G(x_2(\tau), y_2(\tau), x_1(t), y_1(t)) R_{i,p}(t) \sqrt{[x_1'(t)]^2 + [y_1'(t)]^2} dt + \\ ik_0\sigma_2\eta_0 &\sum_{i=0}^{n1+m} e_{1,i} \int_0^1 G(x_2(\tau), y_2(\tau), x_2(t), y_2(t)) R_{i,p}(t) \sqrt{[\dot{x}_2(t)]^2 + [\dot{y}_2(t)]^2} dt = \\ &E_{back}(x_1(\tau), y_1(\tau)) \end{aligned} \tag{2.20}$$

Solving the system of Equations 2.19 and 2.20 is not possible as it is, since it needs to be satisfied for all τ values which are obviously infinite. Several methods, such as Galerkin and collocation methods, can be employed to determine the projection onto the finite-dimensional space $S_k(I^{(m)})$. The latter method is used in this work. This method involves the use of collocation points which would need to be equal or larger than n+m+1. If they are equal, the system is converted into a linear system which is solved directly (interpolation) whereas when more than n+m+1 collocation points are used a least-square approach can be employed. Hence, the collocation points are defined as $P(\hat{\tau_j})$, $j = 0, 1, ..., (n + m + 1)$, and in our case are chosen to be the Greville points linked to the Greville abscissae [22]. The Greville abscissae are knot averages evualated by: $\hat{\tau}_j = \frac{1}{p}(\tau_{i+1} + \tau_{i+2} + \tau_{i+1} + ... + \tau_{i+k-1})$. These knots mostly lie close to the parameter value which corresponds to maximum $R_{i,p}$ and therefore result in a linear system with a relatively low condition number. Inserting the collocation points into Equations 2.19 and 2.20 will result in the following linear system:

$$\begin{aligned} &\sum_{i=0}^{n_1+m} e_{1,i} R_{i,p}(\hat{\tau}_{j_1}) + \\ ik_0\sigma_1\eta_0 &\sum_{i=0}^{n_1+m_2+1} e_{1,i} \int_0^1 G(x_1(\hat{\tau}_{j_1}), y_1(\hat{\tau}_{j_1}), x_1(t), y_1(t)) R_{i,p}(t) \sqrt{[\dot{x}_1(t)]^2 + [\dot{y}_1(t)]^2} dt + \\ ik_0\sigma_2\eta_0 &\sum_{i=0}^{n_2+m_2+2} e_{2,i} \int_0^1 G(x_1(\hat{\tau}_{j_2}), y_1(\hat{\tau}_{j_2}), x_2(t), y_2(t)) R_{i,p}(t) \sqrt{[\dot{x}_2(t)]^2 + [\dot{y}_2(t)]^2} dt = \\ &E_{back}(x_1(\hat{\tau}_{j_1}), y_1(\hat{\tau}_{j_1})) \end{aligned} \tag{2.21}$$

$$\begin{aligned} &\sum_{i=0}^{n_2+m} e_{2,i} R_{i,p}(\hat{\tau}_{j_2}) + \\ ik_0\sigma_1\eta_0 &\sum_{i=0}^{n_2+m_1+1} e_{2,i} \int_0^1 G(x_2(\hat{\tau}_{j_1}), y_2(\hat{\tau}_{j_1}), x_1(t), y_1(t)) R_{i,p}(t) \sqrt{[x_1'(t)]^2 + [y_1'(t)]^2} dt + \\ ik_0\sigma_2\eta_0 &\sum_{i=0}^{n_1+m_1+2} e_{1,i} \int_0^1 G(x_2(\hat{\tau}_{j_2}), y_2(\hat{\tau}_{j_2}), x_2(t), y_2(t)) R_{i,p}(t) \sqrt{[\dot{x}_2(t)]^2 + [\dot{y}_2(t)]^2} dt = \\ &E_{back}(x_2(\hat{\tau}_{j_2}), y_2(\hat{\tau}_{j_2})) \end{aligned} \tag{2.22}$$

Solving the linear system $(n_1 + m_1 + n_2 + m_2 + 2) \times (n_1 + m_1 + n_2 + m_2 + 2)$ from Equations 2.21 and 2.22 for the $\{e_{1,i}\}_{i=0}^{n_1+m_1}$ and $\{e_{2,i}\}_{i=0}^{n_2+m_2}$ results in the approximate electric field $\tilde{E}(t)$ on the boundaries of respective nanotubes using the definition in the Equation 2.18. Afterward, the electric field everywhere inside and outside the boundaries can be obtained using Equation 2.1.

2.4 Parametric Study

The parametric study was conducted to determine the effect of the distance between the two circular nanotubes and the angle of incidence of the electromagnetic plane wave on the cumulative concentration of the electric field inside them. Two identical circles created using NURBS curves were considered. The study was conducted for different sizes and surface conductivities of nanotubes. The initial study was conducted four times for different combinations of nanotube's size and electrical conductivity: $\frac{A_1}{\lambda^2}$ & σ_1, $\frac{A_1}{\lambda^2}$ & σ_2, $\frac{A_2}{\lambda^2}$ & σ_1, $\frac{A_2}{\lambda^2}$ & σ_2. The variable parameters are the normalized distance, $\frac{d}{\lambda}$, between the centers of the nanotubes and the incidence angle of the plane wave. The study was conducted for the range of distance: $\frac{d}{\lambda} = [\frac{2r}{\lambda} : \frac{0.125r}{\lambda} : \frac{15r}{\lambda}]$ so that the shapes do not intersect and are in the range of interaction. The range for the incidence angle was taken as $\alpha = [0, 90]$ in degrees since the pair of circles is symmetric with respect to both x and y-axes, and therefore only one of the quadrants is sufficient for the analysis.

The results of the study can be visualized by showing color maps of the field concentration at different parameters. There are several plots of results that can be obtained: actual cumulative field concentration values $Q_1 + Q_2$, normalized cumulative field concentration $(Q_1 + Q_2)/2Q_c$, and normalized field concentration for each nanotube Q_1/Q_c, Q_2/Q_c. These plots can reveal which regions can be used for maximization and minimization of the field for each nanotube or for the whole system.

2.4.1 Extensive Parametric Study

The extensive parametric study had the same purpose as the previous one: to analyze the effect of the distance between nanotubes and the angle of the incident. However, in this study, the range of nanotube normalized areas and conductivities were significantly increased. In the previous study, the result was the four colormaps. In this extensive study, a three-dimensional matrix is produced containing every colormap for every normalized area and conductivity in the following ranges. Also, viewing the actual values of the concentration as well as normalized concentration values is possible.

- $\frac{A}{\lambda^2} \in [0.5, 10]$ with a step 0.5
- $\sigma_{real} \in [-0.014, 0.014]$ with a step equal to 0.002 for both components of σ
- $\sigma_{imaginary} \in [-0.014, 0.014]$

2.5 Shape Optimization of One Nanotube for both Maximization and Minimization

After the parametric study, the areas with the largest normalized electric field concentration (equation 2.9) were picked for shape optimization. For the identified points the parameters are: area $\frac{A}{\lambda^2} = 1.5$, complex surface conductivity $\sigma_1 = \sigma_2 = 0.005 - 0.005i$, distance between center of the circular nanotubes $\frac{d}{\lambda^2} = 5 \cdot r$, angle of incidence of TM wave $45°$. Using the parametric model, and the developed IGABEM solver, shape optimization using fmincon and patternsearch was performed. The initial optimization was done using 5 control values, gradually increasing them to 14. The number of actual design parameters will be double the size of control values, as was described in 2.2. For each number of parameters, 10 runs were done with random starting points. This was done in order to approach global maxima, as the abovementioned optimization methods may stuck in local maxima.

Additionally, a genetic algorithm was used for a similar single nanotube optimization. Results for all three cases are presented in section 4.3

Regarding minimization, the procedure stayed the same with only the objective function being modified to minimize the electric field inside. Results of this optimization can be found in section 4.5 for results performed in MATLAB and in section 4.8 for the improved C++ code.

2.6 Shape Optimization of Two Nanotubes for Concentration Maximization and Minimization

To optimize both nanotubes the same code and optimizers have been used. Instead of placing a circular tube and optimizing the 2nd shape, the number of parameters has been doubled such that both nanotubes are optimized. Results of this optimization can be found in section 4.4. Optimization was performed in MATLAB and C++ programming environments. For MATLAB mainly patternsearch and genetic algorithms were employed, while for C++ we have been using several optimizers from pagmo family [23], namely improved harmony search, compass search, and cobyla [1, 24]. All three parametric models results are shown in chapter 4. Same procedure was performed for minimization of the electric field.

Chapter 3

Implementation

3.1 Programming Environment

The main programming environments for this project are Matlab and C++. In the beginning, the project was implemented only in Matlab due to its simplicity in the implementation of the solvers. However, the computational time when using MATLAB was high, taking approximately a week to conduct one global optimization. Implementing the project in C++ considerably reduced the corresponding time for computation to 3-6 hours.

There are several steps in the implementation of the problem in Matlab:

- Implementing parametric models for generating the nanotube boundaries
- Implementing the IGABEM solver for estimating the electric field values on the boundary and subsequently the field concentration in the interior; see 2.9
- Implementing the parametric study using the solver
- Implementing optimizer

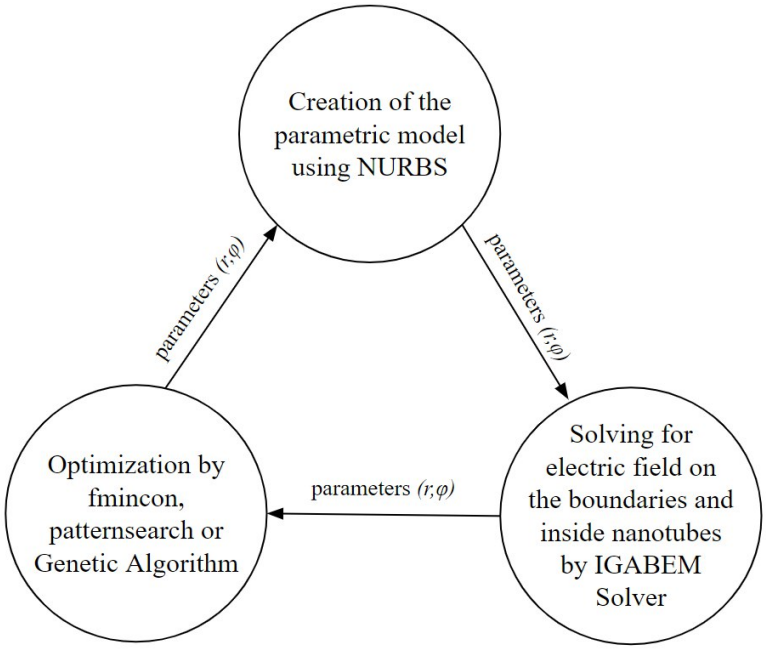


Figure 3.1: Implementation procedure in Matlab

Next, the program was completely rewritten in the C++ programming environment with additional functionalities which extended the capabilities of the program. The resulting program proved to be much more time-efficient compared to MATLAB's implementation. Similar optimizations showed a 10-20 times increase in computational efficiency, yielding better results from both qualitative (higher improvements)

and quantitative perspectives (time-efficient). As discussed above there were several issues associated with MATLAB and the transition to a different programming environment was crucial. Reasons to move to C++ include the following:

- Reducing computational time
- Using the extensive family of PAGMO's optimizers, which are more flexible compared to the three ones that were used in the MATLAB environment
- Extend the capabilities of the program by adding functions that are difficult to implement in MATLAB
- Scaling up and parallelization: Proper and fast division of the tasks between CPU cores and threads using OpenMPI and Intel's oneAPI libraries.

Implementation of the problem in Matlab required 644 lines of code and the transition to C++ required 2682 lines of code and 58 functions. Here are some of the main functions used in C++ to solve the optimization problem:

```
//! generate a closed curve of arbitrary shape using the 1st parametric model
↪  at position curve_index:
void CurveFromParameters_pm1(vector<double> parameters, double r_max,
↪  vector<double> v = { 0,0 }, unsigned int curve_index = 0);

//! generate a closed curve of arbitrary shape using the 2nd parametric model
↪  (with given enclosed area) at position curve_index and return a value that
↪  will stop the optimization if the area tolerance can not be achieved:
double CurveFromParameters_pm2(vector<double> parameters, double r_max, double
↪  area0, vector<double> v = { 0,0 }, unsigned int curve_index = 0);

//! generate an x-symmetric closed curve of arbitrary shape using the 3rd
↪  parametric model (with given enclosed area) at position curve_index and
↪  return the x-coordinate of the generated curve's centroid:
double CurveFromParameters_pm3(vector<double> parameters, double r_max, double
↪  area0, vector<double> v = { 0,0 },unsigned int curve_index = 0);

//! refine the analysis model placing knots_per_span knots uniformly
↪  distributed knots for each knot span for the curve at curve_index:
void RefineCurve(unsigned int knots_per_span, unsigned int curve_index = 0);

//! compute the electric field on two curves from a given background field
↪  using threads and Simpson's integration:
void ComputeFieldOnTwoCurves_TBB(double wavelength, double angle,
↪  vector<complex<double> >sigma, vector<unsigned int> curve_index, double
↪  step = 1e-4);

//! calculate  integral of E inside a single tube when two tubes are present
↪  (using GSL adaptive intergration and TBB):
double CalculateQuantityOnMeshFromTwoTubes_GSL(double wavelength, double
↪  angle, vector<complex<double> > sigma, vector<unsigned int> curve_index,
↪  double tolerance = 1e-3, string filename = "");
```

3.2 Matlab Optimizers

This study requires the implementation of optimizers. The main objective of this project is to maximize the concentration of the electromagnetic field inside the curve. The optimizer will use the parameters that are involved in the creation of the curves in the parametric model. Three main optimizers were used in this capstone project:

- Fmincon
- Patternsearch
- Genetic Algorithm

All of the optimizing methods are available in Matlab's relevant Optimization Toolboxes. Hence, their usage is rather straightforward.

3.2.1 Fmincon Optimizer

Fmincon is a gradient-based nonlinear optimizer that supports both linear and nonlinear constraints. It was the starting point of the optimization of the curve shapes in this work. One of the advantages of using this optimizer is that it has the fastest running time compared to others as it uses a gradient-based technique that converges rapidly to local minima/maxima.

However, it has several disadvantages. One of them is that it is prone to finding the local maxima of the solution. This means that the result of the optimizer is not necessarily the best shape which will exhibit the highest field concentration. The optimizer may get stuck in local maxima, while the best parameters may never be reached. Another disadvantage is that it requires an initial point, i.e., a starting shape to begin its gradient descent approach. Since fmincon is a deterministic optimizer, providing the same initial point will yield the same result, but different starting points may produce wildly varying results, especially when the objective functions have many local maxima. Due to these characteristics, when looking for a global optimum one needs to run the optimization several times with different starting points. This way, we increase our chances of finding the global optimum although this is obviously not guaranteed.

3.2.2 Patternsearch Optimizer

Patternsearch is a family of numerical optimization methods that do not require a gradient. It evaluates the objective function of neighboring points and chooses the best relative position. As it is not a derivative-based method, it is not as prone to stuck in the local maxima area as fmincon. However, it still does not necessarily return the best possible optimized parameters, and multiple runs with varying starting points are required to

increase the chances of finding the global optimum. The efficiency of the patternsearch approach with respect to computational time and results quality is between fmincon and the next optimizer, Genetic Algorithm.

3.2.3 Genetic Algorithm Optimizer

The Genetic Algorithm is a method of solving constrained and unconstrained optimization problems. It employs biological evolution as it simulates natural selection[25]. It creates a population of different random solutions to the problem. Then, it takes these solutions as parents and creates a child that is a mix of both solutions based on their fitness. This process is implemented via three major mechanisms: a) selection (of parents based on fitness), b) crossover (mixing of parents' genes), and c) mutation (randomly changing some genes with low probability). In the case of the project, it would mix the encoding of the parameter values of two solutions. Afterward, the best parameters are taken as the next parents for the generation. These successive generations converge to the optimal parameters. The genetic algorithm guarantees the calculation of a design that is sufficiently close to the actual optimum. However, it is considerably more time-consuming compared to previous optimizers.

3.3 PAGMO Optimizers

Transition to a C++ programming environment required finding optimization algorithms in C++ code [23]. The external library Pagmo is used for optimization. Pagmo is a scientific library written in C++. It contains 25 constrained and unconstrained optimization algorithms that can be easily employed. This optimization library is used by the European Space Agency. The second and third versions of the parametric model allow us to use all available algorithms. It includes bio-inspired, evolutionary-based algorithms. In this project, we used the following optimizers:

- Cobyla
- Compass Search
- Imroved Harmony Search

3.3.1 Cobyla Optimizer

Cobyla - is a derivative-free local optimizer supporting linear and nonlinear inequality and equality constraints. The Cobyla stands for Constrained Optimization by Linear Approximation. It uses a linear approximation of the function to analyze the neighboring points and determine the next point to evaluate. It is a quick algorithm and is used in

local optimization. In this project, it is used mainly for testing the objective function and acquiring quick results.

3.3.2 Compass Search

Compass Search is a derivative-free optimizer in the Pagmo library. It evaluates the parameters around the current design by adjusting the current set of parameters and navigates through the domain of solutions. During the optimization, the step size of adjustments becomes smaller to converge into the optimal solution as shown in Figure 3.2 [1]. It is an analog to pattern search in the Matlab optimization library.

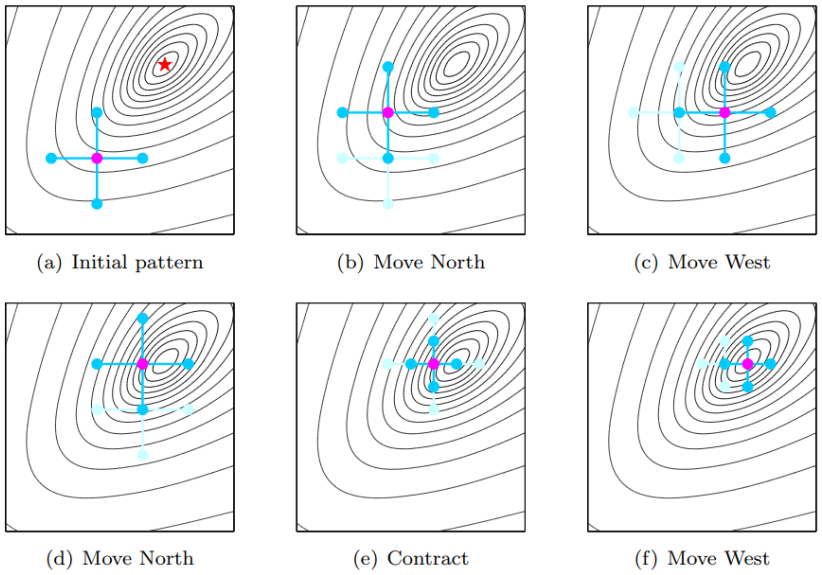


Figure 3.2: Compass Search Optimization Process [1]

3.3.3 Improved Harmony Search Optimizer

Improved Harmony Search (IHS) - is the global optimizer that imitates the musician's improvisation. It is a derivative-free optimizer and requires no initial values. During the play, the musicians improvise and change their pitch to achieve harmony. IHS does the same thing by adjusting the parameters to achieve higher objective function value. IHS starts with randomly generated parameters that compose Harmony Memory (HM). After that, it starts to tune the parameters. It has a probability of choosing the parameters from the HM or newly randomly generated parameters. If it chooses the parameters from the HM, it has another probability of altering the parameter. If the parameters are chosen to be altered, then it is randomly adjusted within boundaries. Then, the new parameters are compared to the parameters with the worst objective function value and replaced if they are better. This process is described in Figure 3.3. Then the stop criteria are checked and optimization is stopped when it is satisfied [24].

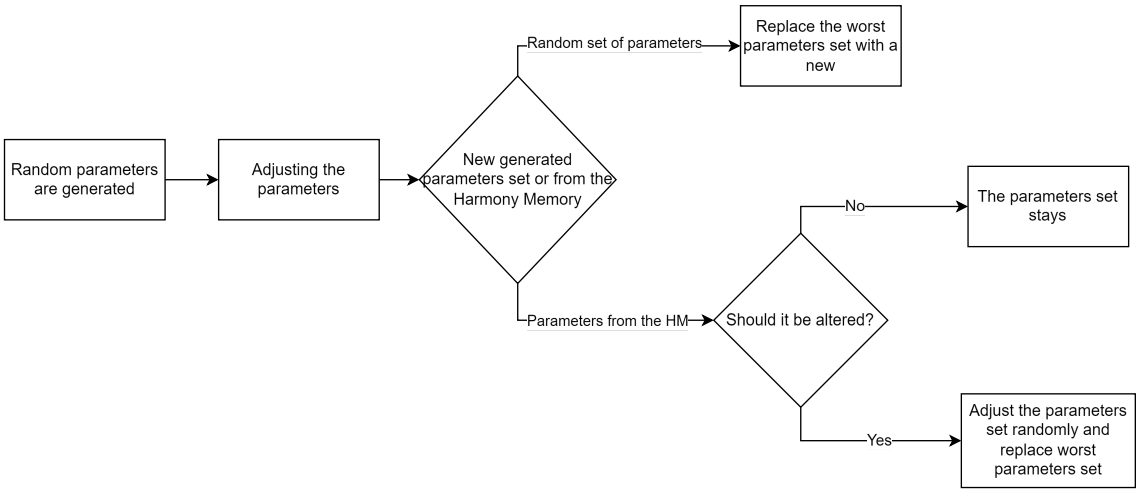


Figure 3.3: Parameters Adjustment Process in IHS

3.4 Optimization Constraints

All of the mentioned optimizers may be used with or without constraints depending on how we formulate the optimization problem. For the problem formulated with the 1st parametric model, there is a linear side constraint, i.e., all parameter values need to lie between 0 and 1, and one nonlinear constraint which is the Area constraint. For accurate results, all tested shapes should have the same area. For example, in this project, we have initially tested two areas of interest: $A/\lambda^2 = 1.5$ and $A/\lambda^2 = 5$. Hence, the optimizer will seek only for the parameters that satisfy this condition. A small error margin equal to 0.01 is used when checking the area constraint so that the curve area can be bigger or smaller than the specified value by this amount. This is done to not limit the optimizer to an exact area, and to prevent ignoring the curves that could give higher results with small deviation from the specified area. The area difference of 0.01 is small, and stricter constraint enforcement may have hindered the optimization methods for reaching the optimal points.

$$|\frac{A_{optimized}}{\lambda^2} - \frac{A_{constrain}}{\lambda^2}| \leq 0.01$$

Optimizers also require specifying lower and upper boundaries for the parameters optimized. Due to the nature of the parametric model the lower and upper boundaries of parameters are simply zeros and ones.

$$0 \leq x \leq 1$$

The area constraint calls for the an effective calculation of the area of arbitrary shapes. Hence, Green's Theorem is employed for the calculation of the area. The line integral described in Equation 3.1 is the equation used in the code to compute the area of the

nanotube for optimization constrain.

$$A = \frac{1}{2} \oint_C xdy - ydx \tag{3.1}$$

This area constraint is only used with the first version of the parametric model. With parametric model versions 2 and 3 the area constraint is not required and the abovementioned method is not applied.

3.4.1 Libraries used in C++

In the following table, libraries used in C++ are shown. Overall 6 major software libraries are used: GoTools, GSL, PAGMO, TBB, GMSH, MPI. Their description and usage are briefly described in the table below.

Library name	Description	Used for
GoTools [26]	Libraries by SINTEF	Represent geometries and electrical fields with NURBS in parametric models and the IGABEM solver
GSL[27]	GNU Scientific Library for numerical jobs	Used to work with linear algebra, obtain solution for the linear systems and perform numerical quadrature
PAGMO [23]	Scientific library for parallel optimization developed by Max Planck Institute for Astronomy and European Space Agency	Used to find an optimum shapes using different optimizers
TBB [28]	Threading Building Blocks library to introduce parallel computation	Used to distribute calculation on CPUs with processors having large number of cores
GMSH [29]	3D FEM mesh generator	Used to discretize the domain and visualize the geometry
MPI [30]	Message Passing Interface for parallel programming	Used to compute in parallel on an arbitrary number of cooperating computers

Table 3.1: Software Libraries Breakdown

Chapter 4

Results and Discussion

4.1 Analytical Solution

Before starting any calculations and optimizations, it was crucial to verify that solver implementation using NURBS and IGABEM is accurate. A single nanotube subjected to a point source of TM wave was put at a certain distance normalized to the wavelength. The numerical results compared to the analytical solution were identical, which proves that the implemented solver is accurate and can represent the solution inside the nanotube. The convergence of analytical and numerical solutions implemented can be seen in Figure 4.1.

On the graph, both real and imaginary values are plotted, as well as their absolute value. Circular dots represent the analytical solution, whereas the continuous lines represent the IGABEM solver. Close overlap between dots and continuous lines indicates that real, imaginary, and absolute values agree very well.

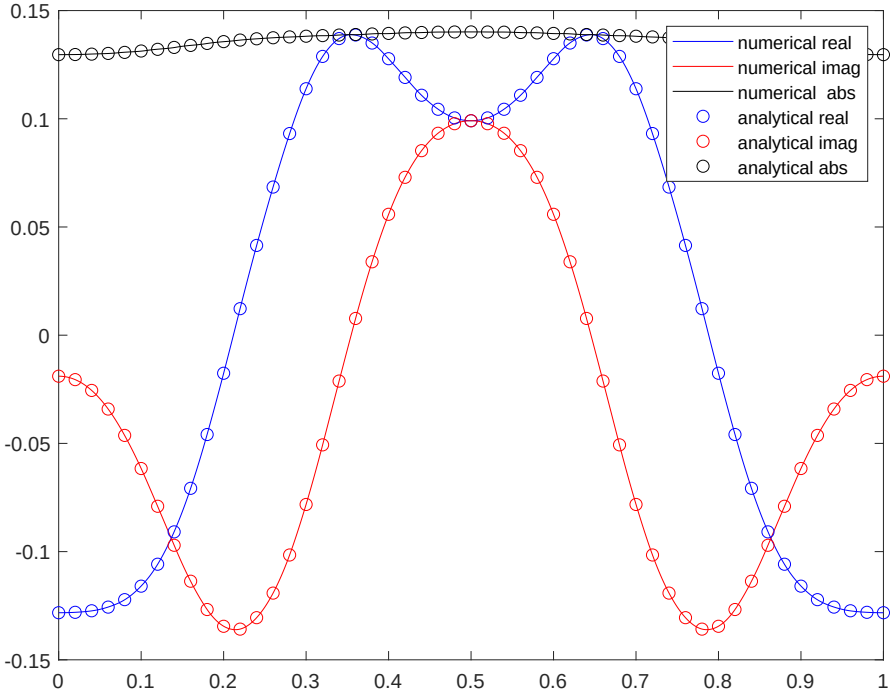


Figure 4.1: Electric Field vs Location on the Boundary

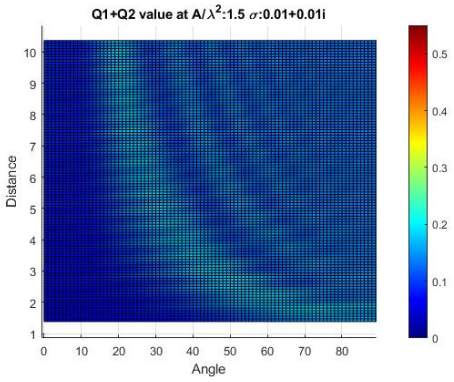
4.2 Parametric Study

The initial parametric study was conducted for two nanotubes of circular shape as was described in section 2.4. The results of this parametric study are shown in Figures 4.2 and 4.3. There are four color maps plotted for each combination of area and surface conductivity. The results are given as follows: $\frac{Q_1+Q_2}{2Q_c}$ - the concentration of electric field inside of two identical cylindrical nanotubes divided by double the concentration inside a single cylindrical nanotube of the same size and conductivity; $Q_1 + Q_2$ - the magnitude of the electric field concentration of two nanotubes combined; $\frac{Q_1}{Q_c}$ and $\frac{Q_2}{Q_c}$ - the concentration of electric field inside of the nanotube located on the left and on the right, respectively, compared to a single cylindrical nanotube of the same size and conductivity.

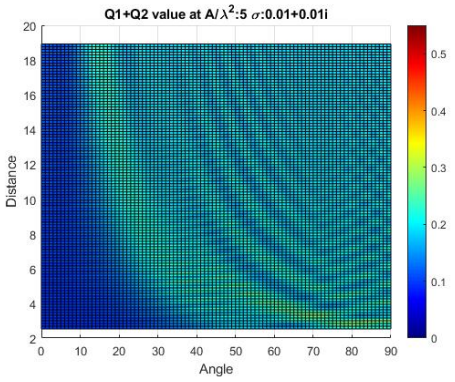
Comparing the color-maps in Figures 4.2 and 4.3 (a) and (b), it can be observed that the magnitude of electric field concentration is higher for a larger area and for smaller surface conductivity, however, the normalized concentration results in Figures 4.2 and 4.3 (c) and (d) show higher values for a smaller area and larger conductivity. This implies that nanotubes of smaller area and larger electrical surface conductivity concentrate a larger amount of electrical field when in pairs rather than alone.

It can also be observed that there is a hyperbolic pattern in regions where field concentration is relatively high. The pattern repeats when shifting to the up-right corner of the colormap and the field concentration also reduces in those regions.

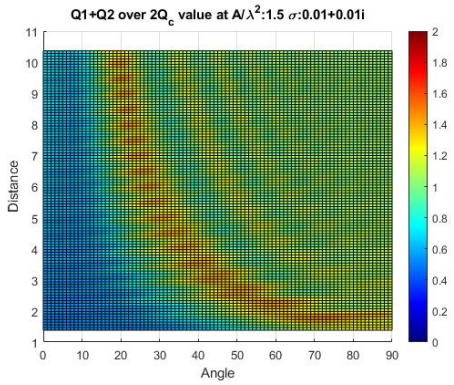
The electromagnetic field interaction between two nanotubes can be viewed in Figures 4.2 (e),(f),(g),(h) and 4.3 (e),(f),(g),(h). Subfigures (e) and (f) show a larger area on the colormap with relatively high electric field concentration, while subfigures (g) and (h) show less electric field concentration, especially in regions where the angle of incidence of EM wave is small and the distance between the nanotubes is large. This means that the nanotube located on the left blocks the EM wave directed to the right. The red regions in the bottom right corners of colormaps in subfigures (g) and (h) show that, when EM wave hits at an angle of 90 degrees or from below and the distance between nanotubes is small, the field concentration is high for both nanotubes located on the left and right, which implies maximum interaction between nanotubes. It is important to note that on the right side of colormaps in subfigures (e) and (g), (f) and (h) the field concentration is similar, which means that at an angle of 90 degrees, the nanotubes interact with the EM wave symmetrically.



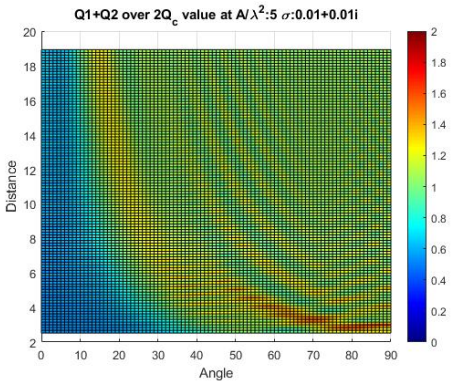
(a) $Q_1 + Q_2$ for $A = 1.5\lambda^2$



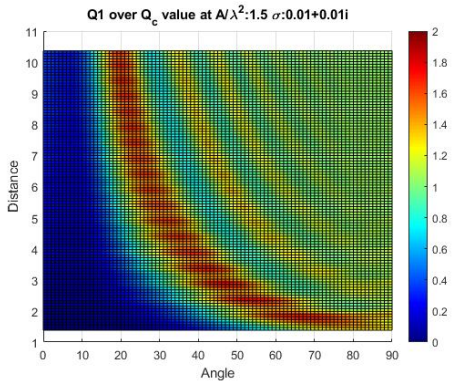
(b) $Q_1 + Q_2$ for $A = 5\lambda^2$



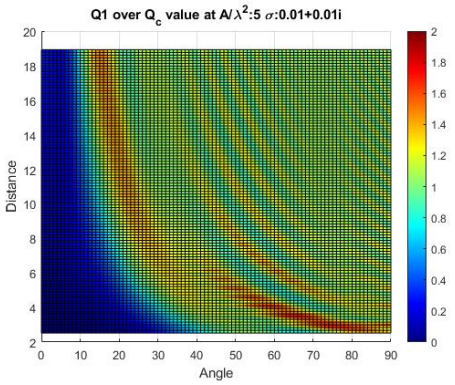
(c) $\frac{Q_1+Q_2}{2Q_c}$ for $A = 1.5\lambda^2$



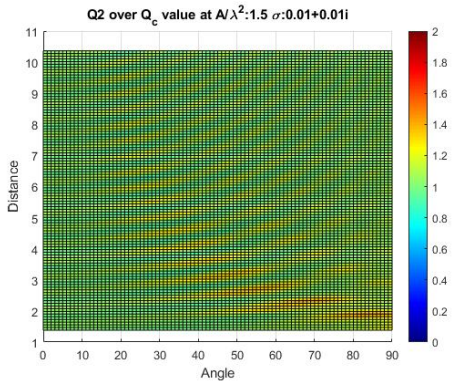
(d) $\frac{Q_1+Q_2}{2Q_c}$ for $A = 5.0\lambda^2$



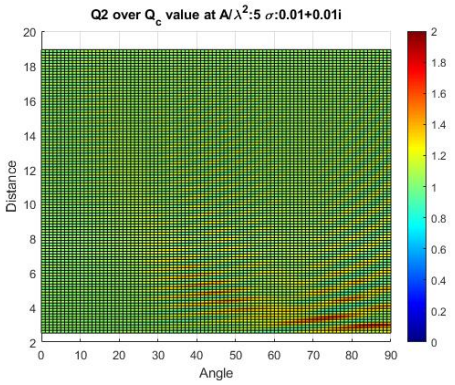
(e) $\frac{Q_1}{Q_c}$ for $A = 1.5\lambda^2$



(f) $\frac{Q_1}{Q_c}$ for $A = 5.0\lambda^2$

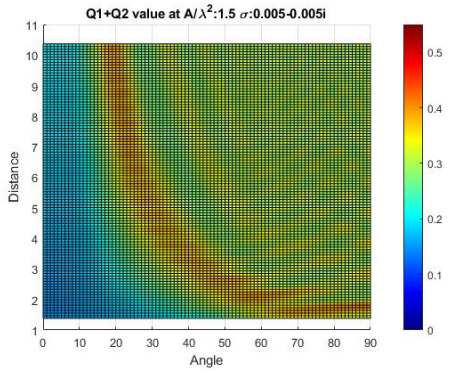


(g) $\frac{Q_2}{Q_c}$ for $A = 1.5\lambda^2$

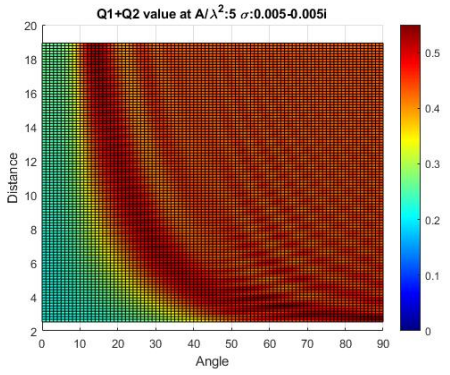


(h) $\frac{Q_2}{Q_c}$ for $A = 5.0\lambda^2$

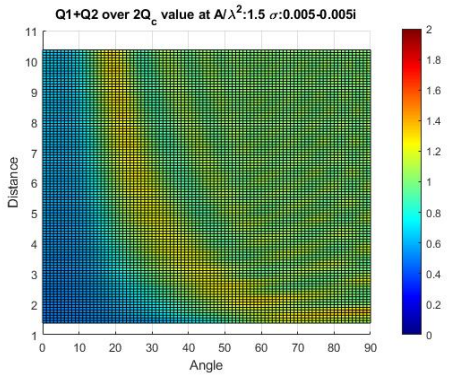
Figure 4.2: Parametric study results of electromagnetic field concentration for two cylindrical nanotubes with surface conductivity $\sigma = 0.01 + 0.01i$.



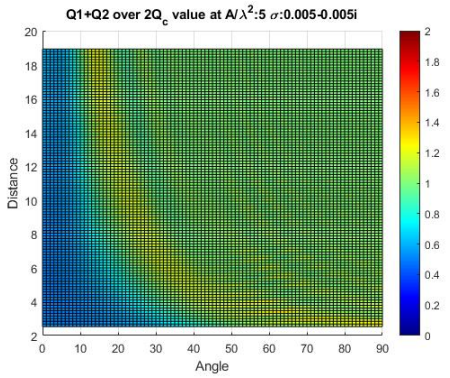
(a) $Q_1 + Q_2$ for $A = 1.5\lambda^2$



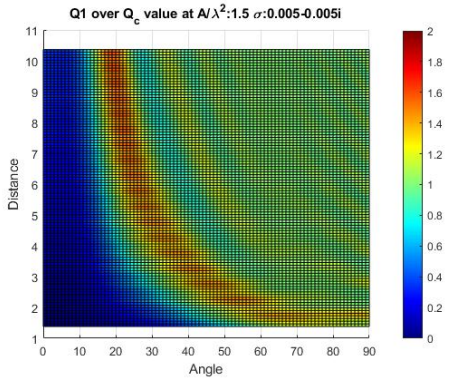
(b) $Q_1 + Q_2$ for $A = 5\lambda^2$



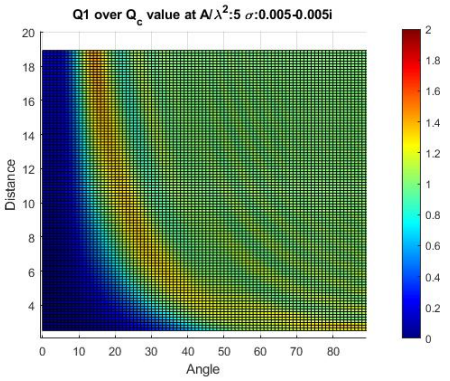
(c) $\frac{Q_1+Q_2}{2Q_c}$ for $A = 1.5\lambda^2$



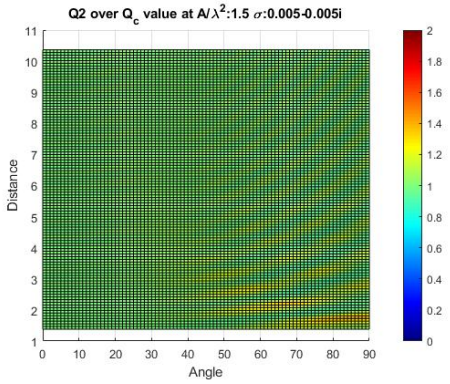
(d) $\frac{Q_1+Q_2}{2Q_c}$ for $A = 5.0\lambda^2$



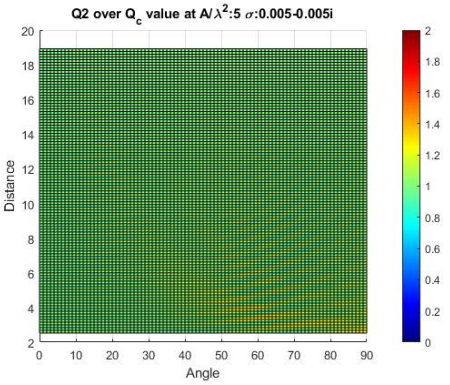
(e) $\frac{Q_1}{Q_c}$ for $A = 1.5\lambda^2$



(f) $\frac{Q_1}{Q_c}$ for $A = 5.0\lambda^2$



(g) $\frac{Q_2}{Q_c}$ for $A = 1.5\lambda^2$

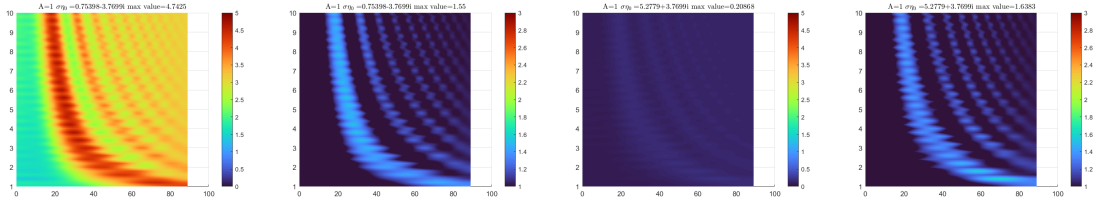


(h) $\frac{Q_2}{Q_c}$ for $A = 5.0\lambda^2$

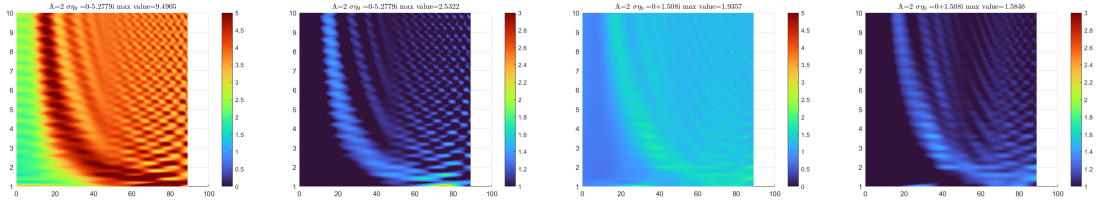
Figure 4.3: Parametric study results of electromagnetic field concentration for two cylindrical nanotubes with surface conductivity $\sigma = 0.05 - 0.05i$.

4.2.1 Extensive Parametric Study Results

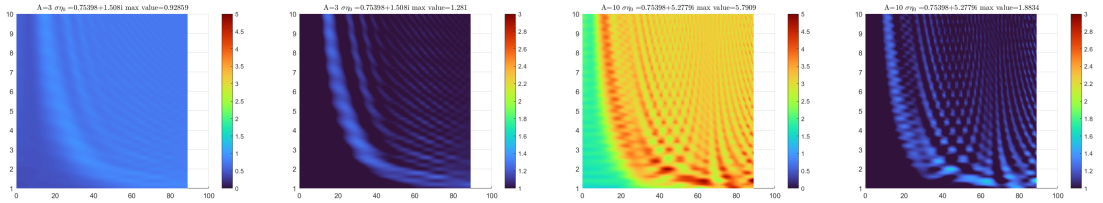
Figure 4.4 shows some of the results of the extensive study. The examples include electric field concentration absolute value and normalized value for nanotubes of normalized area equal 1, 2, 3, and 10. Also, every colormap has a different conductivity. As can be seen, these colormaps confirm the findings of the previous study. More examples of the extensive parametric study is included in Appendix A.



(a) Normalised Area = 1 (b) Normalised Area = 1, Normalised (c) Normalised Area = 1, Different Conductivity (d) Normalised Area = 1, Normalised, Different Conductivity



(e) Normalised Area = 2 (f) Normalised Area = 2, Normalised (g) Normalised Area = 2, Different Conductivity (h) Normalised Area = 2, Normalised, Different Conductivity

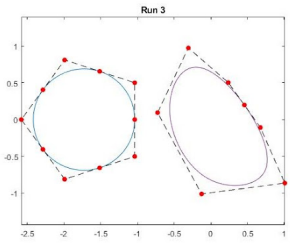


(i) Normalised Area = 3 (j) Normalised Area = 3, Normalised (k) Normalised Area = 10 (l) Normalised Area = 10, Normalised

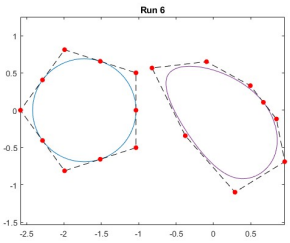
Figure 4.4: Extensive Parametric Study

4.3 Shape optimization of one nanotube in a pair

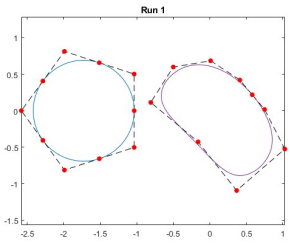
The optimization of a single nanotube, while the second one remains circular, with different numbers of DOFs was conducted as described in Section 2.5. The nanotube on the right was used for optimization since it showed lower concentration in the colormaps. The optimization was conducted with fmincon for a range of Dofs from 5 to 10 using random starting points. The optimized shapes are shown in Figure 4.5.



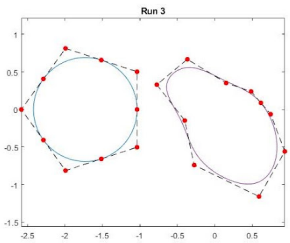
(a) 5 DoFs, $\max(\frac{Q_1+Q_2}{2Q_c}) = 2.791$



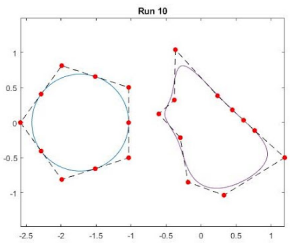
(b) 6 DoFs, $\max(\frac{Q_1+Q_2}{2Q_c}) = 3.2013$



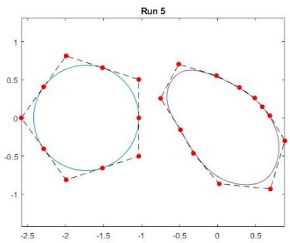
(c) 7 DoFs, $\max(\frac{Q_1+Q_2}{2Q_c}) = 2.8974$



(d) 8 DoFs, $\max(\frac{Q_1+Q_2}{2Q_c}) = 3.4176$

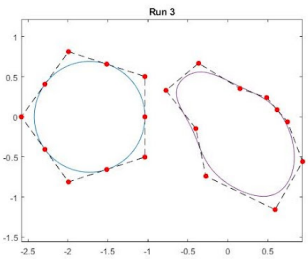


(e) 9 DoFs, $\max(\frac{Q_1+Q_2}{2Q_c}) = 3.001$

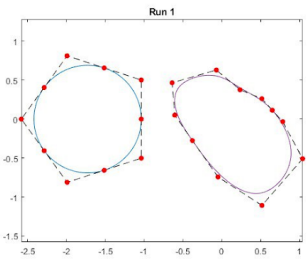


(f) 10 DoFs, $\max(\frac{Q_1+Q_2}{2Q_c}) = 3.1211$

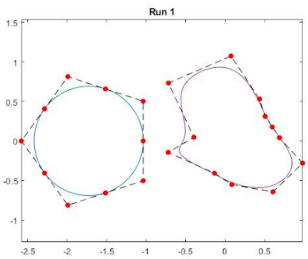
Figure 4.5: Fmincon optimized shapes of nanotubes of size $A = 1.5\lambda^2$ and conductivity $\sigma = 0.005 - 0.005i$ for various DoFs



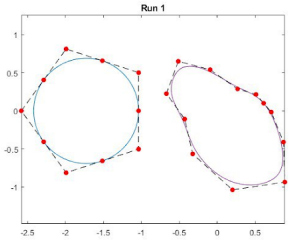
(a) Initial shape 8 DoFs, fmincon, $\frac{Q_1+Q_2}{2Q_c} = 3.4176$



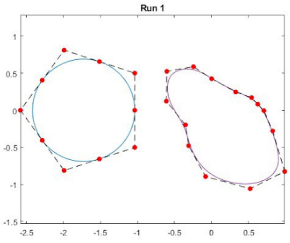
(b) increase to 9 DoFs, fmincon, $\frac{Q_1+Q_2}{2Q_c} = 3.2984$



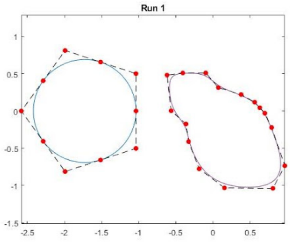
(c) increase to 10 DoFs, fmincon, $\frac{Q_1+Q_2}{2Q_c} = 1.575$



(d) increase to 10 Dofs, pat-ternsearch $\frac{Q_1+Q_2}{2Q_c} = 3.5307$



(e) increase to 12 DoFs, pat-ternsearch, $\frac{Q_1+Q_2}{2Q_c} = 3.6098$



(f) increase to 14 DoFs, pat-ternsearch, $\frac{Q_1+Q_2}{2Q_c} = 3.6843$

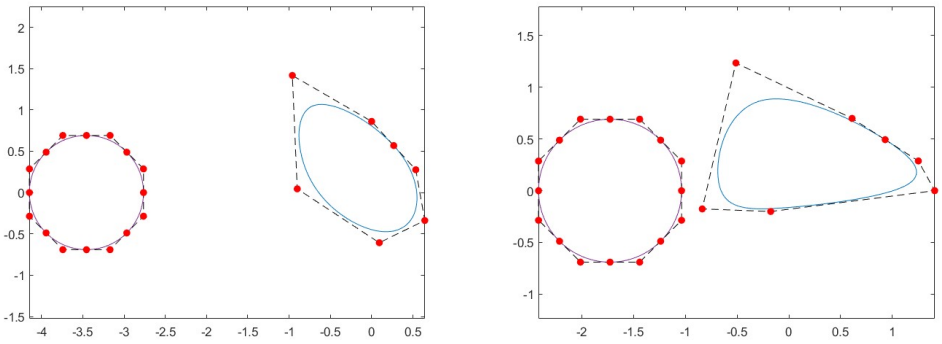
Figure 4.6: Shape optimization by increasing the number of DoFs of the best-optimized shape.

It is shown that with increasing shape DOFs the field concentration changes. After optimizing the shape in fmincon from 5 to 8 DOFs the concentration is raised to 3.4 times higher when compared to the circular tube. Using the shape resulted with the highest field concentration (Figure 4.5 (d)) and increasing its number of control points the optimization was repeated as shown in Figure 4.6. However, after that fmincon would show lower results, going down to 1.6 at 10 DOFs. One of the reasons might be that more parameters create more local maxima where the fmincon can get stuck and return false optima. Hence, afterward, the patternsearch method was used. The concentration was raised to 3.7 at 14 DOFs. The optimized shape is shown in Figure 4.6(f).

4.3.1 GA results

GA optimization breaks the ceiling of a three-fold increase in the concentration. The resulting improvement in concentration is $\frac{Q_1+Q_2}{2*Q_c} = 4.02$, which is about 4 times higher compared to the value from two circular nanotubes. The optimized shape is shown in Figure 4.7 a).

To investigate other areas of the colormap in Figure 4.3 the region that gives the best concentration with two circular tubes was taken for optimization. At this region, the distance is $D/\lambda = 2.5 \times R$ and the wave angle is $\alpha = 90^o$. Using these conditions, the new optimized shape resulted in the electric field concentration increase of $\frac{Q_1+Q_2}{2*Q_c} = 4.226$. The optimized shape is shown in 4.7 b).



(a) $\frac{Q_1+Q_2}{2*Q_c} = 4.02$ (b) $\frac{Q_1+Q_2}{2*Q_c} = 4.226$

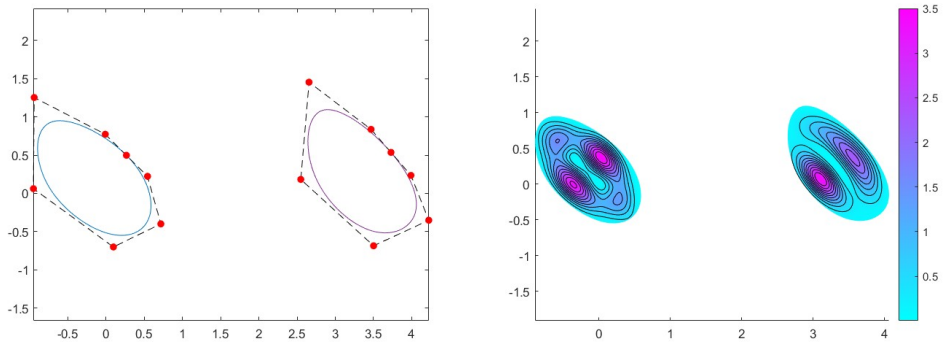
Figure 4.7: Optimized shapes obtained from GA

4.4 Optimization of two nanotubes

With the addition of the remaining tube to the maximization problem, the optimization results for two curves are presented. However, the results both from the qualitative and quantitative sides are not so rich since the time required for computing increased at least twice, and now one optimization requires a couple of days. The Genetic Algorithm optimization took more than four days to give a result. Note that this problem is addressed when using the C++ implementations.

4.4.1 GA and Patternsearch Results

The results from previous optimization were used to facilitate the process of optimization for both curves. The GA optimization in Figure 4.7 b) showed considerable results. Hence, the second curve was used as starting point for both shapes in patternsearch. The distance between nanotubes was set as $D/\lambda = 5 \times R$ and the wave angle is $\alpha = 45^o$. The results are shown in Figure 4.8 a).



(a) Two optimized shapes (b) Electric field concentration

Figure 4.8: GA and patternsearch Results

The optimized shape gives the normalized concentration of $\frac{Q_1+Q_2}{2 \cdot Q_c} = 6.216$

This result shows that optimizing both shapes gives an increase in the concentration of the electric field of 622%. Figure 4.8 b) shows the electric field inside the nanotubes. It is evident that even though the first curve is blocking the electromagnetic wave for the second curve the optimized shape helps to gather as much electromagnetic field as possible.

4.4.2 Sensitivity analysis

To check how the optimized shape behaves under different circumstances the concentration value of the optimized shape (optimized for the maximum concentration) of the nanotube was recalculated. The normalized distance and angle of the plane wave were the variables in the sensitivity analysis.

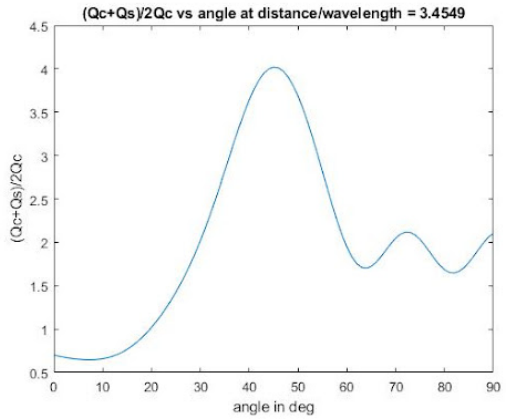


Figure 4.9: Sensitivity to Angle Adjustment

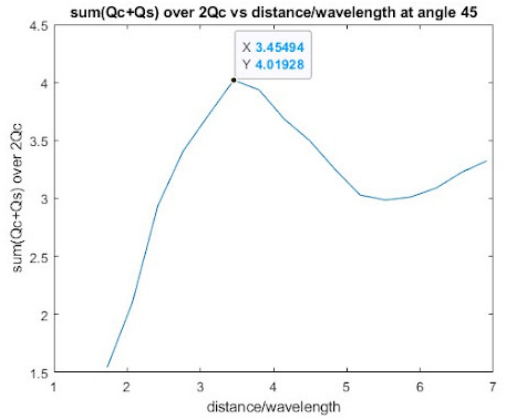


Figure 4.10: Sensitivity to Distance Adjustment

It is evident from Figures 4.9 and 4.10 that the optimized shape gives uplifting results. For the angle difference, the concentration for angles between 20 to 90 stays above 1, which means it is better than using circular nanotubes. Varying the distance between the nanotubes affects the concentration. However, for the range of normalized distance from 2 to 7, the concentration stays clearly above the concentrations achieved with the circular shapes.

4.5 Minimization by patternsearch optimizer

The minimization of total concentration inside nanotubes is also a part of the research. The method for finding the best shape is identical to the method described for maximization in sections 2.5. At the current moment, the optimization of a single nanotube of two was conducted only by patternsearch. The optimization was conducted 10 times with random starting points to find 10 local minima and take the solution with the least field concentration. The optimized shape is shown in Figure 4.11. The normalized total field concentration for the resulting shape is 0.552. The distance between nanotubes is $5 \times r$ and the incidence angle of the EM wave is 45 degrees.

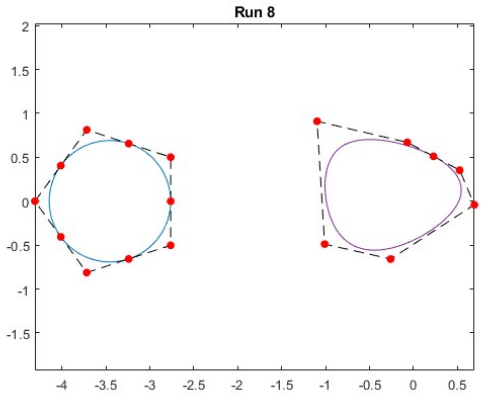


Figure 4.11: Optimized shape with 5 params and normalized concentration 0.552

4.6 Time and DOFs convergence study

In order to accurately estimate the required number of analysis DOFs for accurate calculations, an additional study was done. Logically, we can insert as many knots, as we wish, however this will result in an excessive computational cost. For this study the case with normalized concentration equal to 11.8 was chosen. These shapes have thin sections, that complicate the calculation, thus the study can accurately help to estimate number of DOFs and time consumed for calculation that can be assumed to be appropriate for most of the cases. From figure 4.12 it can be seen that the calculated value converges after 40 DOFs, while time consumption grows significantly from 1 second at 20-45 DOFs to 10 seconds at 160 DOFs. Using these results, all of the calculations were performed to ensure that the number of DOFs during calculation is above 60, computational time needed with these parameters and an entry level PC is around 2 seconds. These optimized values, allowed us to speed up the calculation process significantly, without using redundant DOFs.

The same study was performed with circular tubes, to estimate how many knots should be inserted to optimize calculation time. Circles calculation is done in order to obtain reference values with circular nanotubes. Calculation results can be seen in Figure 4.12. Using this result, the conclusion was drawn that 60 DOFs and above are convenient to use and the relative error is below 1%. Time consumed is around 4 seconds, compared to 10 seconds and above, where number of DOFs exceeds 120.

4.7 Optimization by IHS optimizer

After implementing the problem in C++, the optimization was conducted with the local optimizer "Cobyla" to get results in a short time and also verified with our Matlab code. In comparison with Matlab optimization, the computation time was reduced by more than ten times, providing relatively high quality results in a few hours. The main optimizer used in

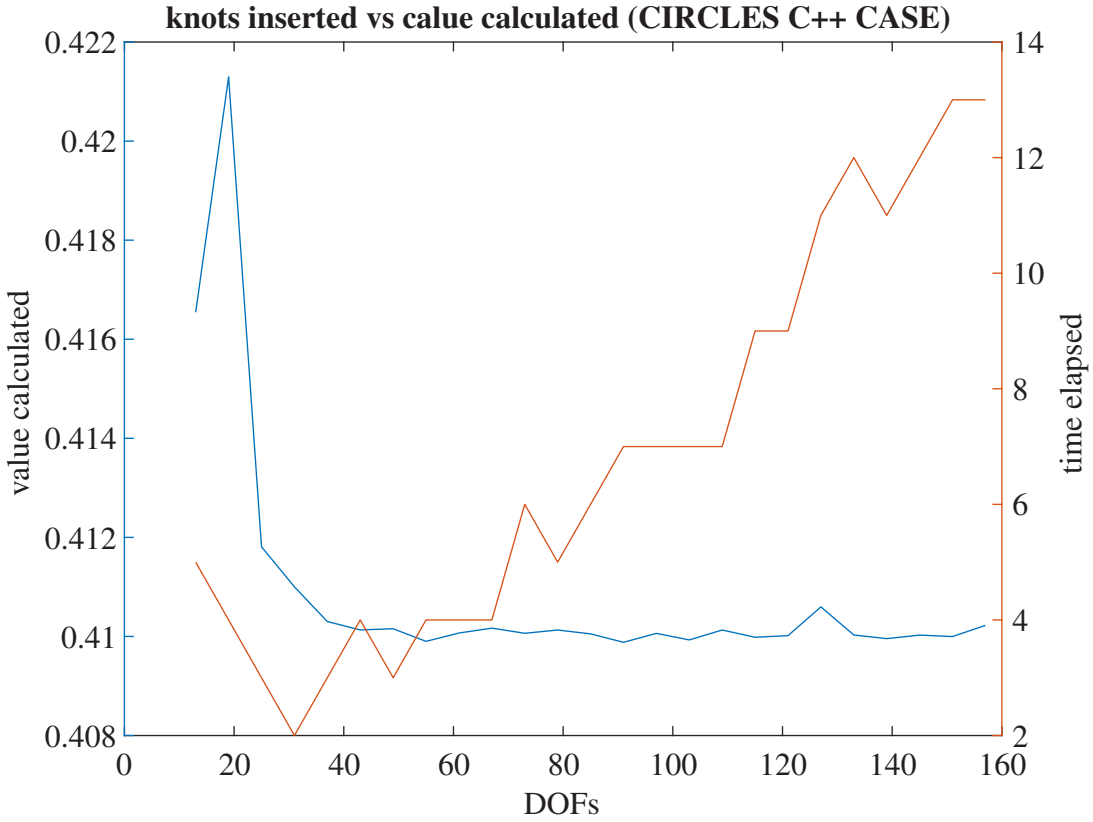


Figure 4.12: DOFs vs time consumed

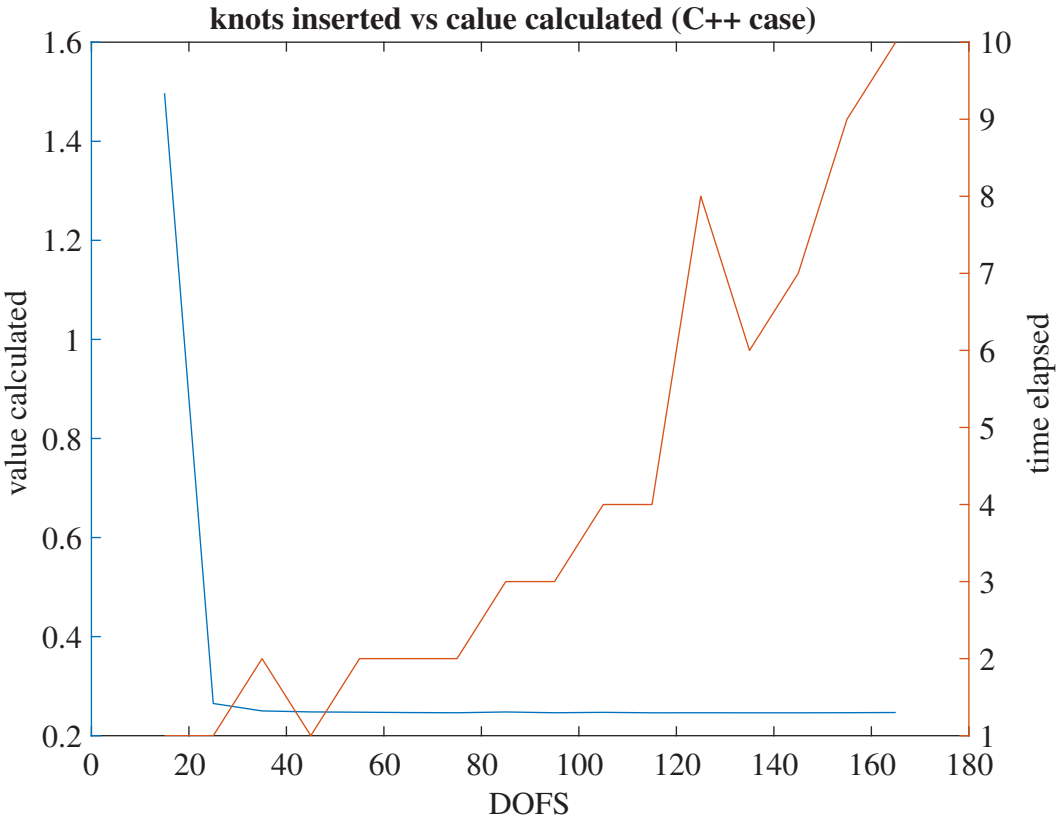


Figure 4.13: DOFs vs time consumed for circular tubes

C++ was the global optimizer "IHS", which also produced the results in a relatively short period. The optimization was conducted for non-symmetric and symmetric parametric models. In both cases, the resulting electric field concentration inside optimized shapes increased by more than 10 times relative to two circular nanotubes.

4.7.1 Optimization with a non-symmetric parametric model

The first optimization was conducted for 8-DoF shapes with a nanotube area of $A/\lambda^2 = 1.5$, surface conductivity of $\sigma = 0.005 - 0.005i$, distance of $d = 2.5 \times r$, and wave angle of $\alpha = 90^o$. The optimization resulted in a 6.6239 times larger concentration compared with two separate circular nanotubes and this result was obtained in about 3 hours, which is much less than the time spent on GA optimization for a similar result. The optimized shape is shown in Figure 4.14. The shapes are very similar since the angle of incidence of the wave is set in such a way that nanotubes interact with the electric field and affect each other equally.

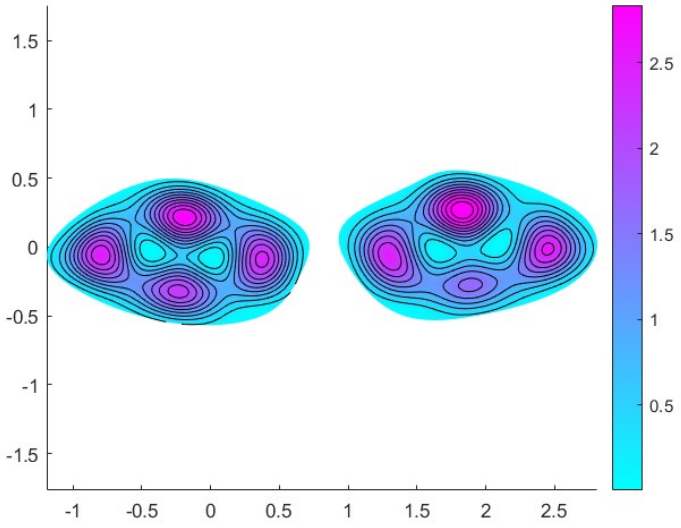
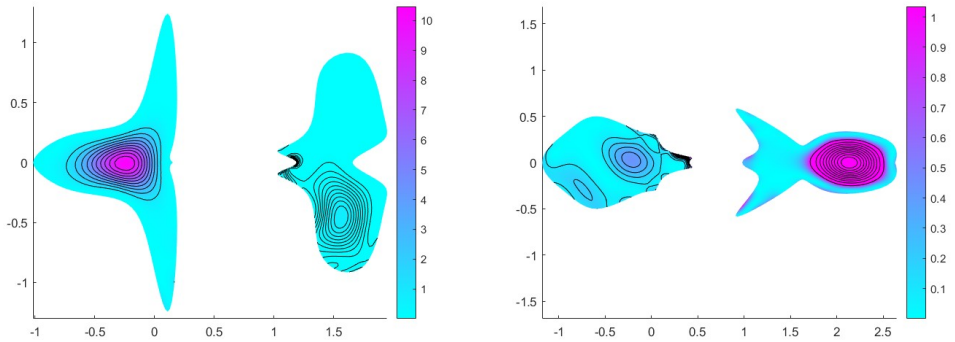


Figure 4.14: First optimized shape by IHS: $\frac{Q_1+Q_2}{2 \cdot Q_c} = 6.6239$

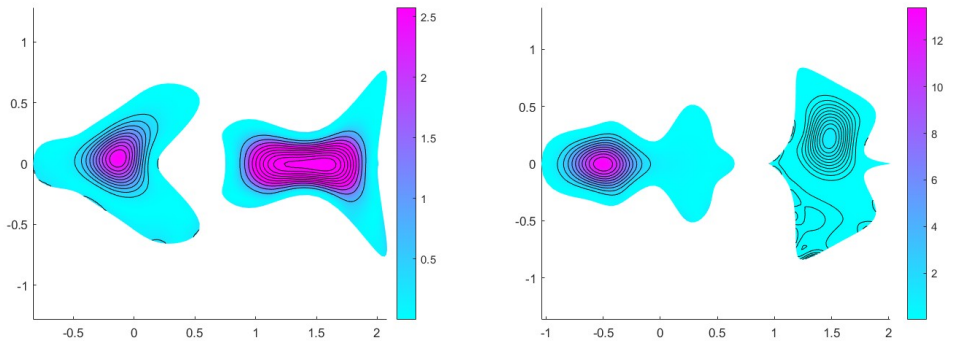
4.7.2 Optimization with a symmetric parametric model

The symmetric parametric model was used to optimize shapes using parameters from the three-dimensional parametric study. The conditions used for the first case are a nanotube area of $A/\lambda^2 = 1$, surface conductivity of $\sigma = 0.002 + 0.014i$, distance of $d = 2.8 \times r$, and wave angle of $\alpha = 70^o$. The number of DoFs used to create the shape is again 8. Several intermediate optimization results that give a 10-13-fold increase in concentration were obtained from this case. These shapes are shown in Figure 4.15. These figures have a similar pattern of pointing out a circular or oval shape on the left and an arc-like shape

on the right of a nanotube. In Figure 4.15 a) and d) the electric field is concentrated well only in one of the nanotubes, while the other nanotube, taking an arc-like shape, assists in converging the electromagnetic wave in the first nanotube. In Figure 4.15 c) it is seen that the electromagnetic field is concentrated in both nanotubes and both of them form an arc-like shape on the right, using it as an antenna concentrating field on the left side. This shape opens a possibility to make an array or matrix of such shapes to make up a metasurface concentrating electric field in all nanotubes.



(a) $\frac{Q_1+Q_2}{2*Q_c} = 10.3965$ (b) $\frac{Q_1+Q_2}{2*Q_c} = 10.8074$



(c) $\frac{Q_1+Q_2}{2*Q_c} = 11.7953$ (d) $\frac{Q_1+Q_2}{2*Q_c} = 12.6843$

Figure 4.15: Optimized shapes obtained from IHS using the symmetric parametric model (1st case)

The best result of a 17.7417-fold increase in concentration over two circular nanotubes was obtained with the same parameters. The shape that gives this result is shown in Figure 4.16. This shape also concentrates most of the electric field on the left side. Both nanotubes have an arc-like shape and act like two antennas for a point on the left.

The second case optimization was conducted in the region of the cubic colormap where the nanotube area is $A/\lambda^2 = 2$, surface conductivity is $\sigma = 0 + 0.008i$, distance is $d = 2.8 \times r$, and wave angle of $\alpha = 40^o$. The electric field concentration obtained from this optimization is increased about 9 times compared with circular nanotubes. Two shapes with different numbers of DoFs were optimized for this case (Figure 4.17). Due to a large surface area, the electric field concentrated region is also large, including the boundaries

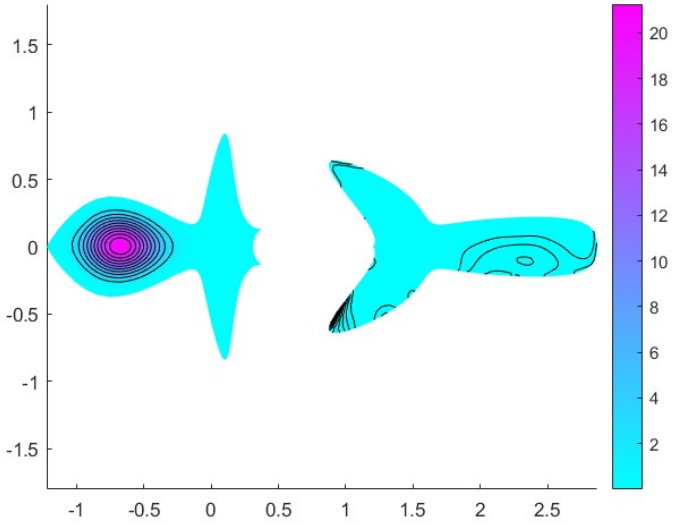
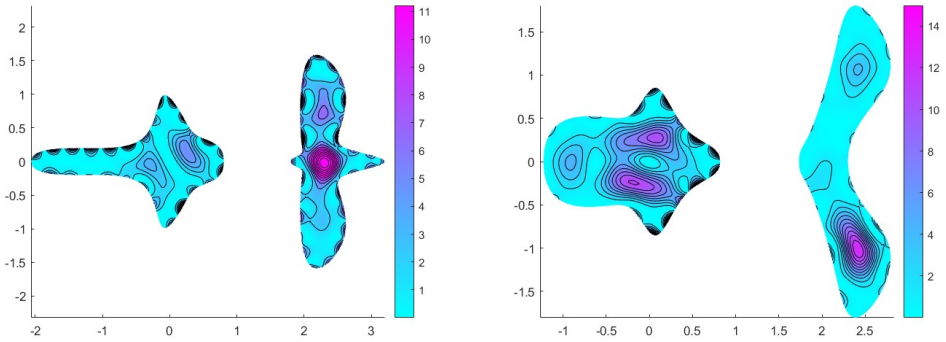


Figure 4.16: Optimized Shape by IHS: $\frac{Q_1+Q_2}{2*Q_c} = 17.7417$ (1st Case)

and the center of the nanotubes. In this case, the electric field is mostly concentrated in the nanotube located on the right, while in the previous case, it was on the left side. It might be due to the change in the angle of incidence of the wave.



(a) 8 DOFs: $\frac{Q_1+Q_2}{2*Q_c} = 8.7321$ (b) 6 DOFs: $\frac{Q_1+Q_2}{2*Q_c} = 8.8796$

Figure 4.17: Optimized shapes obtained from IHS using the symmetric parametric model (2nd Case)

4.8 Minimization by IHS optimizer

As another part of the research, an additional objective was to minimize the electric field concentration inside nanotubes. In Figure 4.18 two optimized shapes, used to minimize the electric field, are shown. Normalized concentration compared to circular nanotubes is equal to 0.0174 or 57.6 times lower than circular nanotubes. This study was done in C++ using the IHS optimizer with the distance between nanotubes set to be $d = 5.5 \times r$,

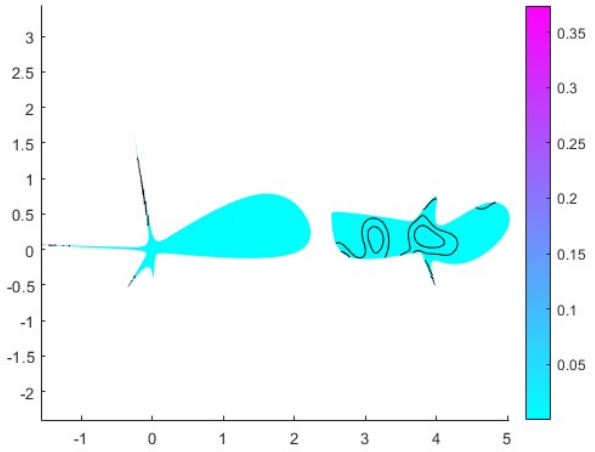


Figure 4.18: Optimized shape for Minimization in C++: $\frac{Q_1+Q_2}{2 \cdot Q_c} = 0.0174$

angle of TM wave $\alpha = 0^o$, area $A/\lambda^2 = 1.5$ and complex surface conductivities of both tubes $\sigma = 0.005 - 0.005i$. The result was obtained in approximately 12 hours, with more than 50000 iterations. To analyze how this shape performs in different conditions two sensitivity analyses were performed: one with variable distance and a second with variable angle.

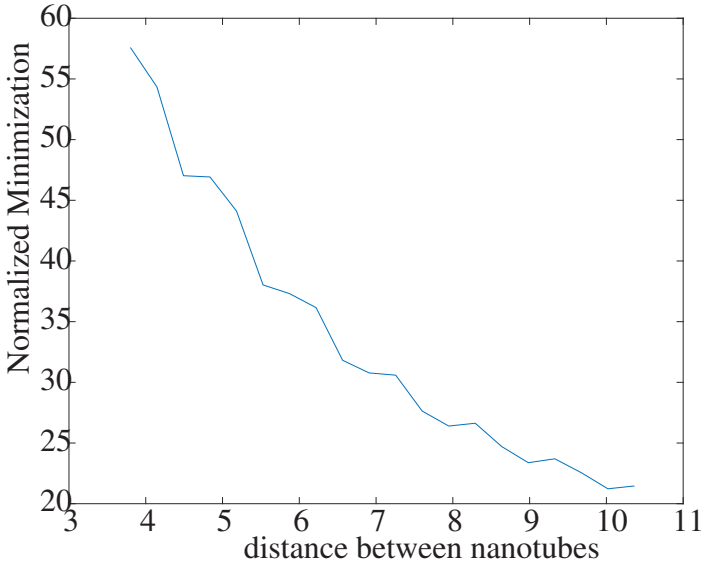


Figure 4.19: Sensitivity analysis of minimized case with variable distance

In figure 4.19 it can be seen that the overall trend is minimization properties decrease with an increase in distance between nanotubes, however, even with distance between nanotubes set to be $d = 15 \times r$ optimized nanotubes outperform circular tubes 22 times.

Contrary to distance variation, optimized shapes are far more sensitive to angle adjustment. From Figure 4.20 it can be seen that the optimized shape minimizes the electric field concentration inside better than circular up to an angle of 40 degrees, while

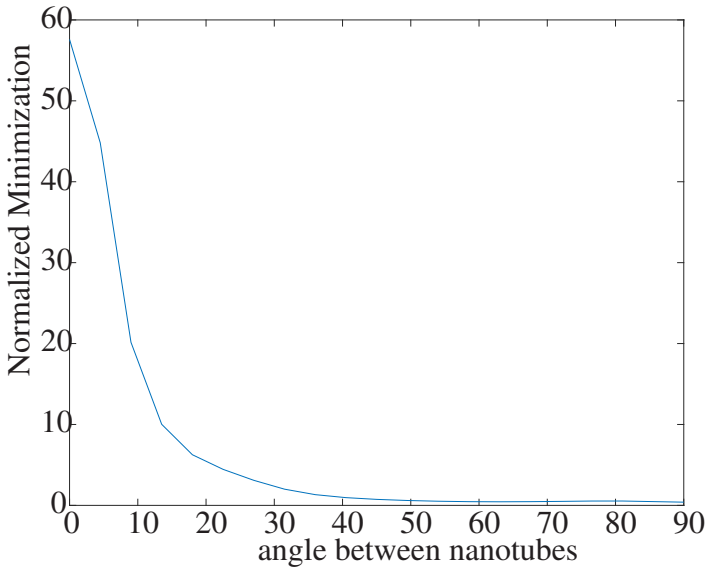


Figure 4.20: Sensitivity analysis of minimized case with variable angle

at angles higher circular nanotubes behave better. That is why for this case, it is important to locate nanotubes at a certain specified angle, but distance variation is not as important.

The minimization was also conducted for other regions of the extended parametric study, using an angle of incidence of the wave $\alpha = 89^o$, surface conductivity of nanotubes $\sigma = 0.012 + 0.014i$, area $A/\lambda^2 = 1$, and distance $d = 1.4 \times r$. As a result of the minimization, the electric field concentration of the optimized shapes was reduced to 9.38% of the concentration that would be obtained from two circular nanotubes. This means that the result is 10.7 times less. The optimized shape giving this result is shown in Figure 4.21.

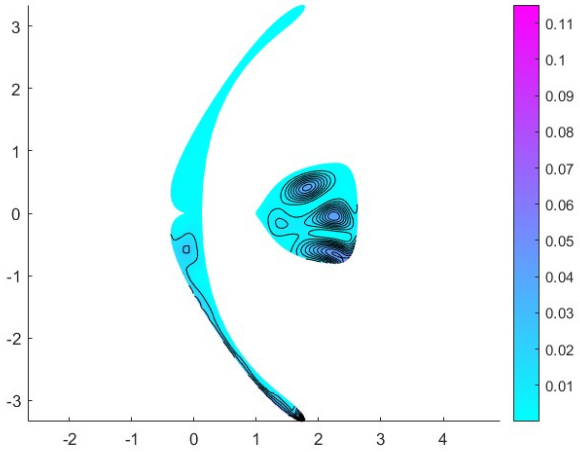


Figure 4.21: Optimized Shape for Minimization using parameters from the cubic colormap: $\frac{Q_1+Q_2}{2 \cdot Q_c} = 0.0938$

4.9 Limitations

One of the biggest limitations of this capstone project is that simulations done on the initial stage of the project were performed with the fmincon optimizer. As stated in the section 3.2.1, the fmincon optimizers are prone to getting stuck in the local minima and frequently cannot reach shapes that possess the best performance. After transitioning to patternsearch and genetic algorithm, optimizers' results increased notably with the same number of degrees of freedom. Another quite important note about the simulations and optimization is computational time in MATLAB. For the Genetic Algorithm to converge, a minimum time of 3 days is required for 1 run and up to 10 days, for the runs with a maximum number of parameters. To substantially reduce computational time C++ implementation of the code was done. In addition to that, after the transition to C++, a High-Performance Cluster was used to reduce computational time by employing special external libraries that parallelize calculations in CPUs with a high number of cores.

Another probable cause of small uncertainty in calculations is quantum effects, that are notable at nanoscale and temperature effects. Nanotubes subjected to electromagnetic waves can heat up and material properties can not be assumed uniform due to temperature gradients on the boundary.

Finally, it is difficult to validate theoretical results with large-scale metamaterial arrays due to the high complexity of current manufacturing technologies. In addition to that, a study of an array of nanotubes should be implemented in order to understand the interaction between multiple nanotubes, not only 2 of them.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The stated aim of this capstone project was to maximize and minimize the electric field concentration inside arbitrarily shaped nanotubes subjected to TM waves using different optimization methods and by employing Non-Uniform Rational B-Splines with the Isogeometric Analysis Boundary Element Method. The analysis started by solving the simple problem with two cylindrical-shaped nanotubes using the Boundary Element Method.

The parametric study involving the identification of the most promising regions was carried out with 3 main parameters: angle of incidence of TM wave, distance between nanotubes, and nanotubes' area. Next, gradient-based and guided random search optimization methods combined with the Isogeometric-Analysis-based Boundary Element Method were utilized in order to optimize the shape of single nanotube and nanotube pairs. Appropriate usage of a parametric model, NURBS, and IGS toolboxes in MATLAB was essential to accurately model free-form geometries, and evaluate the shapes' performances. Importantly, the advantage of the parametric model used is that no self-intersecting geometries are generated.

Transition to C++ allowed to significantly reduce computational time, use the extended list of modified optimizers, and generate more complicated and promising results with a new parametric model. Six external libraries were employed in C++ implementation allowing more flexible control over shape optimization.

The results demonstrate that a significant increase in the concentration of the electric field inside the optimized nanotubes is obtainable. A six-fold increase was achieved compared to circular nanotubes using the Genetic Algorithm. Optimizers implemented in C++ allowed us to obtain a 13-17 times increase in electric field concentration for two nanotubes.

Minimization of electric field concentration inside nanotubes is another objective of the project. 55 times decrease was achieved compared to circular nanotubes.

Additional studies of time and Degrees of Freedom convergence were performed in order to estimate the appropriate number of parameters required for an accurate and stable solution with minimal time consumption.

Optimized shapes of the nanotubes have a broad range of promising applications. They can be used in fiber optics and other data transfer technologies. The semiconductor industry

also puts nanotubes into service for micro-scale optical devices and field emission devices. Antennas and other equipment employed in wireless data transfer can utilize optimized shape nanotubes to improve signal concentration and security. Gas sensors, electrodes, and electric energy accumulators also can find use of shape-optimized nanotubes.

In conclusion, this capstone project has successfully achieved all stated objectives. Maximization and minimization of electric field concentration inside arbitrarily shaped nanotubes were implemented using NURBS, IGABEM, and optimization methods. The robust, accurate mathematical model was created and results show multifold improvement of optimized geometries that are readily can be employed in modern technologies.

5.2 Future Work

The future steps are as follows:

- Extend the framework to an array of nanotubes to see their interaction, not only a pair of tubes. This will allow us to estimate the feasibility of creating a meta-material with optimized nanotubes.
- The material choice is broad, and nanotubes nowadays can be manufactured from a variety of materials. Find the right materials with conductivities used in the study. Additionally, look into manufacturing methods that can generate nanotubes of arbitrary shape.
- Validate the results with experimental studies. Such experimental studies are currently being conducted in the context of the NU-funded CRP project WICAN coordinated by our supervisors.

5.3 Distribution of Tasks

Student name	Tasks
Aidana Faizulla	Extension of Code for One Nanotube to Two nanotubes, Parametric Study, Optimization of One Nanotube with fmincon and patternsearch, Sensitivity Analysis, Minimization of Field Concentration, Implementation of the code in C++, Optimization and Minimization using Cobyla and IHS
Ravil Ashirmametov	Implementation of Optimizers in MATLAB, Parametric Study, Genetic Algorithm implementation, Minimization of Field Concentration, Optimization of Two Nanotubes, Implementation of the code in C++, Minimization of concentration in C++, setting up C++ optimizers
Abzal Aznabayev	Extension of Code for one nanotube to two nanotubes. Implementation of Genetic Algorithm, patternsearch, and fmincon optimizers, Optimization of two nanotubes using various methods, Implementation of the code in C++. Optimization using compass search and IHS.

Table 5.1: Distribution of Tasks

Bibliography

[1] T. Kolda, R. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods. siam review 45, 385-482," *SIAM Review*, vol. 45, pp. 385–482, 09 2003.

[2] S. Iijima, "Helical microtubules of graphitic carbon," *Nature*, vol. 354, pp. 56–58, 1991.

[3] V. N. Popov, "Carbon nanotubes: properties and application," *Materials Science and Engineering: R: Reports*, vol. 43, no. 3, pp. 61–102, 2004.

[4] M. F. D. Volder, S. H. Tawfick, R. H. Baughman, and A. J. Hart, "Carbon nanotubes: Present and future commercial applications," *Science*, vol. 339, no. 6119, pp. 535–539, 2013.

[5] R. H. Baughman, A. A. Zakhidov, and W. A. de Heer, "Carbon nanotubes–the route toward applications," *Science*, vol. 297, no. 5582, pp. 787–792, 2002.

[6] N. Govindaraju and R. Singh, "Chapter 8 - synthesis and properties of boron nitride nanotubes," in *Nanotube Superfiber Materials* (M. J. Schulz, V. N. Shanov, and Z. Yin, eds.), pp. 243–265, Boston: William Andrew Publishing, 2014.

[7] G. Papanicolaou and D. Portan, "27 - carbon and titanium dioxide nanotube polymer composite manufacturing – characterization and interphase modeling," in *Structural Integrity and Durability of Advanced Composites* (P. Beaumont, C. Soutis, and A. Hodzic, eds.), Woodhead Publishing Series in Composites Science and Engineering, pp. 735–761, Woodhead Publishing, 2015.

[8] M. J. Schulz, B. Ruff, A. Johnson, K. Vemaganti, W. Li, M. M. Sundaram, G. Hou, A. Krishnaswamy, G. Li, S. Fialkova, S. Yarmolenko, A. Wang, Y. Liu, J. Sullivan, N. Alvarez, V. Shanov, and S. Pixley, "Chapter 2 - new applications and techniques for nanotube superfiber development," in *Nanotube Superfiber Materials* (M. J. Schulz, V. N. Shanov, and Z. Yin, eds.), pp. 33–59, Boston: William Andrew Publishing, 2014.

[9] J.-C. Charlier, "Defects in carbon nanotubes," *Accounts of Chemical Research*, vol. 35, no. 12, pp. 1063–1069, 2002. PMID: 12484794.

[10] K. Jiang, J. Wang, Q. Li, L. Liu, C. Liu, and S. Fan, "Superaligned carbon nanotube arrays, films, and yarns: A road to applications," *Advanced Materials*, vol. 23, no. 9, pp. 1154–1161, 2011.

[11] K. Kostas and C. Valagiannopoulos, "Optimally shaped nanotubes for field concentration," 10 2023.

[12] T. Hughes, J. Cottrell, and Y. Bazilevs, "Isogeometric analysis: Cad, finite elements, nurbs, exact geometry and mesh refinement," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 39, pp. 4135–4195, 2005.

[13] C. Politis, A. Ginnis, P. Kaklis, K. Belibassakis, and C. Feurer, "An isogeometric bem for exterior potential-flow problems in the plane," *Proceedings - SPM 2009: SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pp. 349–354, 10 2009.

[14] S. Piltan and D. Sievenpiper, "Field enhancement in plasmonic nanostructures," *Journal of Optics*, vol. 20, p. 055401, apr 2018.

[15] L. Lv, H. Jiang, Z. Ding, Q. Ye, N. Al-Dhahir, and J. Chen, "Secure non-orthogonal multiple access: An interference engineering perspective," *IEEE Network*, vol. 35, no. 4, pp. 278–285, 2021.

[16] T. Yang, "Optimizing electrode structure of carbon nanotube gas sensors for sensitivity improvement based on electric field enhancement effect of fractal geometry," *Scientific Reports*, 2021.

[17] S. Zhang, Y. Wang, X. Han, Y. Cai, and S. Xu, "Optimizing the fabrication of carbon nanotube electrode for effective capacitive deionization via electrophoretic deposition strategy," *Progress in Natural Science: Materials International*, vol. 28, no. 2, pp. 251–257, 2018.

[18] E. A. Bocharov GS, "Theory of carbon nanotube (cnt)-based electron field emitters.," *Nanomaterials*, 2013.

[19] S. Zhang, N. Nguyen, B. Leonhardt, C. Jolowsky, A. Hao, J. G. Park, and R. Liang, "Carbon-nanotube-based electrical conductors: Fabrication, optimization, and applications," *Advanced Electronic Materials*, vol. 5, no. 6, p. 1800811, 2019.

[20] D. Macri, "Using nurbs surfaces in real-time applications." https://www.intel.com/content/dam/develop/external/us/en/documents/40951-real-time-nurbs-142397.pdf, 2000. Accessed: 24.11.2023.

[21] K. Kostas, C. Politis, I. Zhanabay, and P. Kaklis, "A physics-informed parametrization and its impact on IGABEM analysis," 2023.

[22] W. J. Gordon and R. F. Riesenfeld, "B-spline curves and surfaces," in *Computer Aided Geometric Design* (R. E. BARNHILL and R. F. RIESENFELD, eds.), pp. 95–126, Academic Press, 1974.

[23] F. Biscani and D. Izzo, "A parallel global multiobjective framework for optimization: pagmo," *Journal of Open Source Software*, vol. 5, no. 53, p. 2338, 2020.

[24] M. Mahdavi, M. Fesanghary, and E. Damangir, "An improved harmony search algorithm for solving optimization problems," *Applied Mathematics and Computation*, vol. 188, no. 2, pp. 1567–1579, 2007.

[25] MathWorks, "What is the genetic algorithm?." https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html, n.d. Accessed: 19.11.2023.

[26] Sintef, "Gotools," 2009.

[27] J. Gough, "Gsl - gnu scientific library," 2021.

[28] Intel, "Intel® oneapi threading building blocks," 2021.

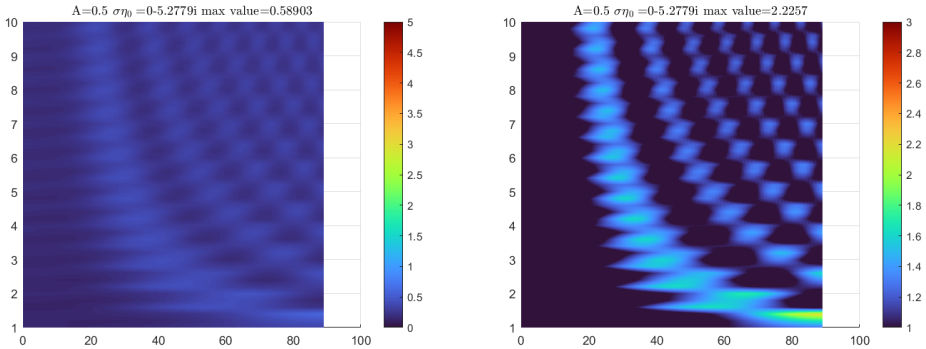
[29] R. Geuzaine, "A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities," 2009.

[30] O. M. Community, "Open mpi v5.0.x," 2024.

Appendices

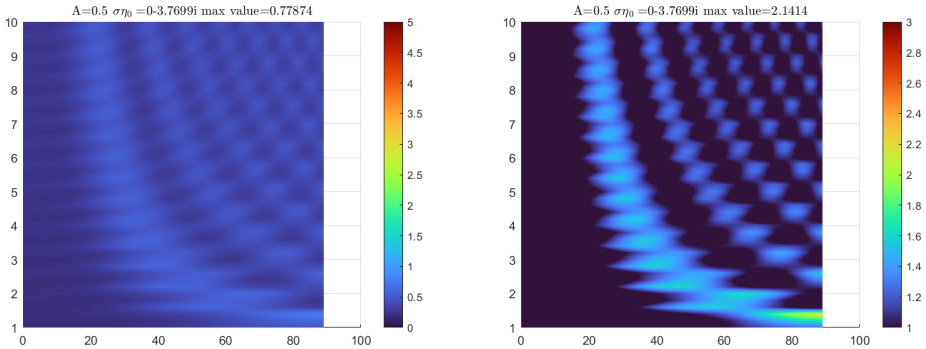
Appendix A

Extensive Parametric Study Results



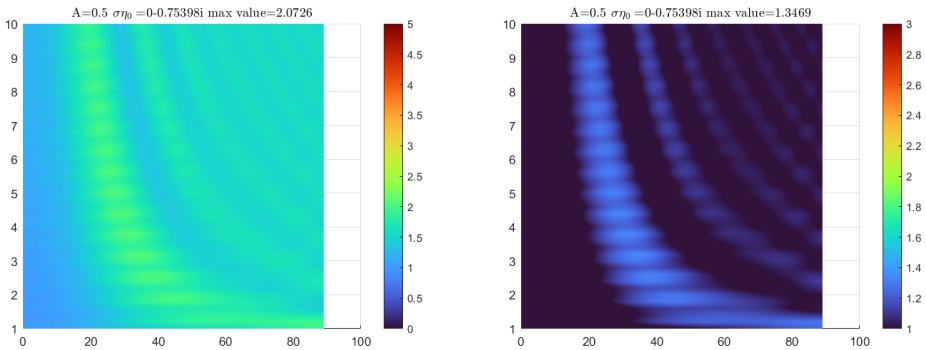
(a)

(b) Normalised



(c)

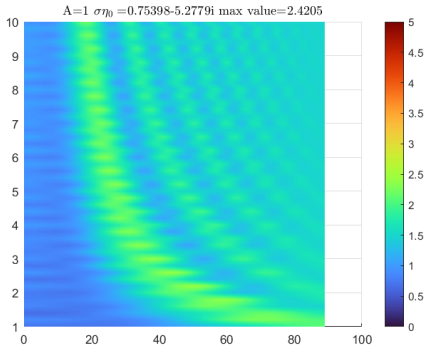
(d) Normalised



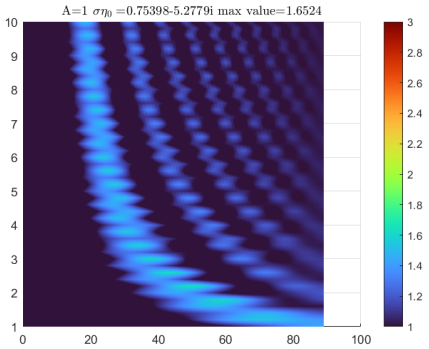
(e)

(f) Normalised

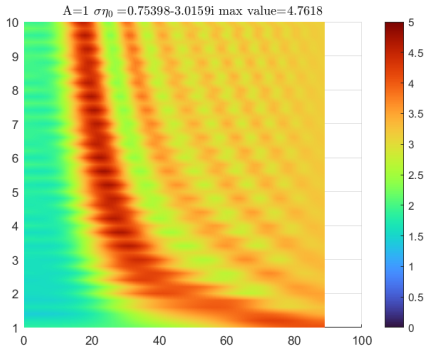
Figure A.1: Normalised Area = 0.5, Extensive Parametric Study



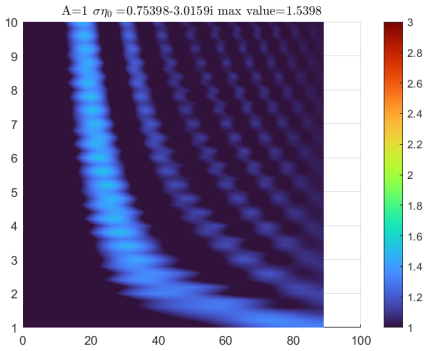
(a)



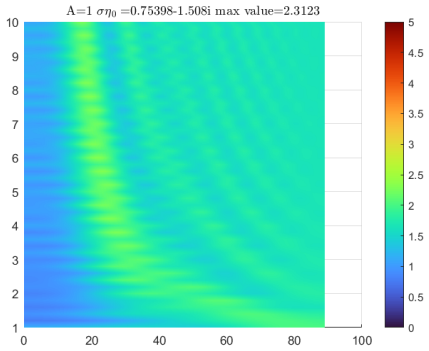
(b) Normalised



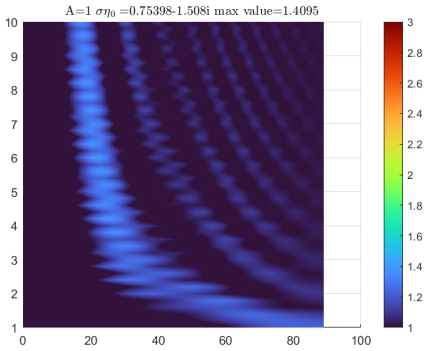
(c)



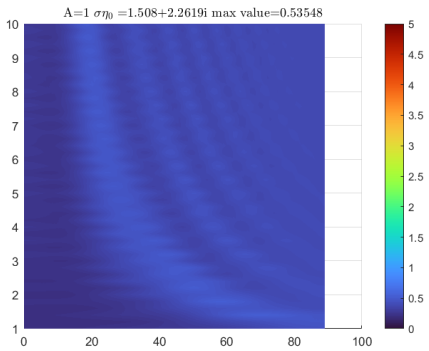
(d) Normalised



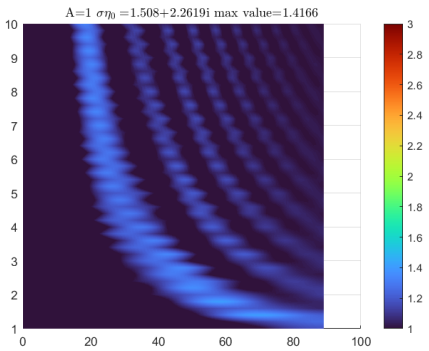
(e)



(f) Normalised

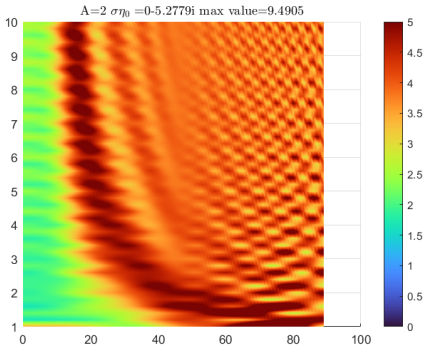


(g)

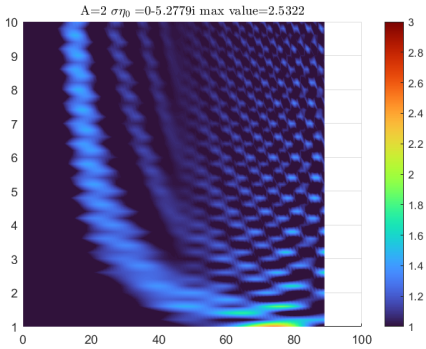


(h) Normalised

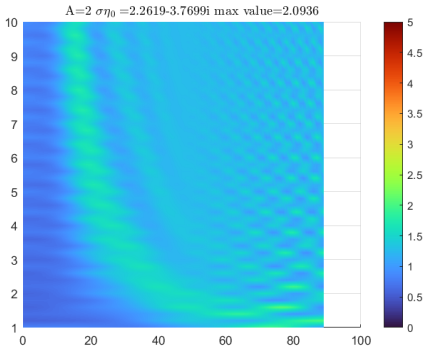
Figure A.2: Normalised Area = 1, Extensive Parametric Study



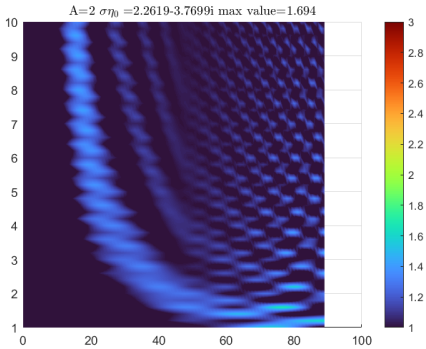
(a)



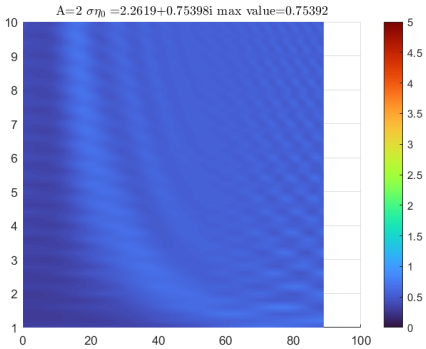
(b) Normalised



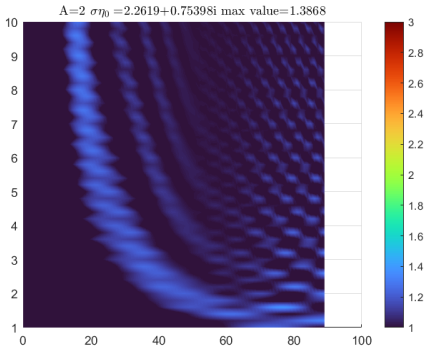
(c)



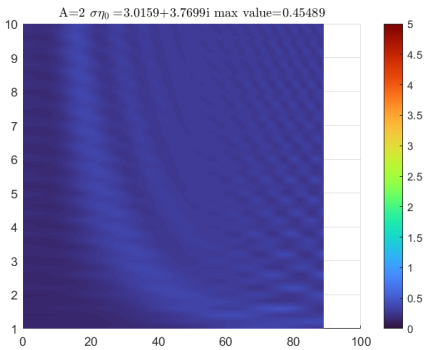
(d) Normalised



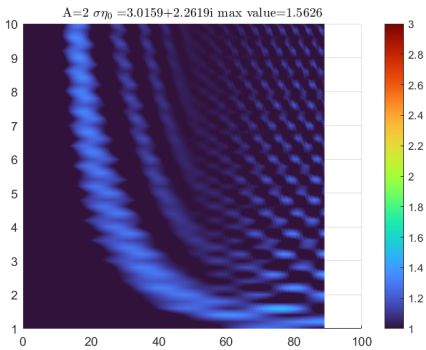
(e)



(f) Normalised

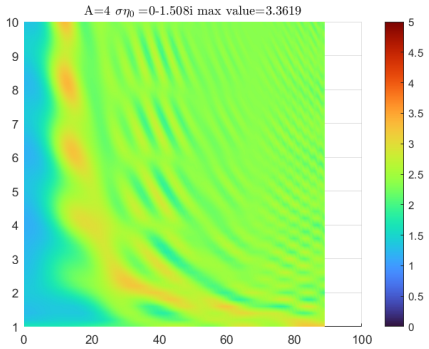


(g)

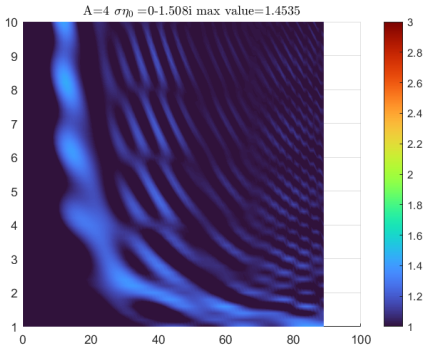


(h) Normalised

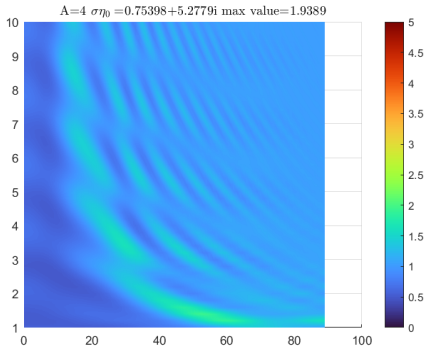
Figure A.3: Normalised Area = 2, Extensive Parametric Study



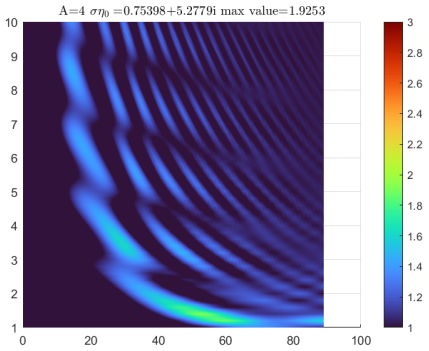
(a)



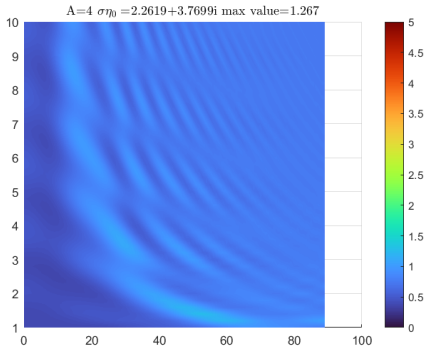
(b) Normalised



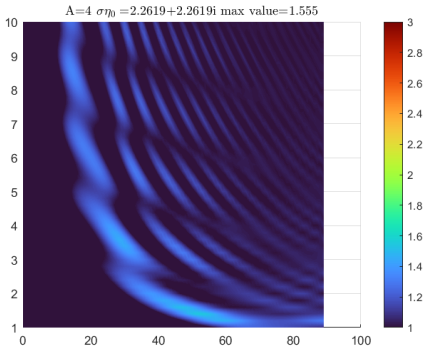
(c)



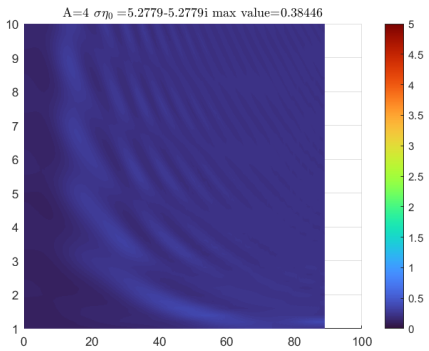
(d) Normalised



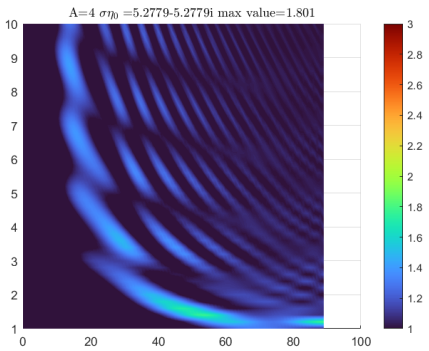
(e)



(f) Normalised

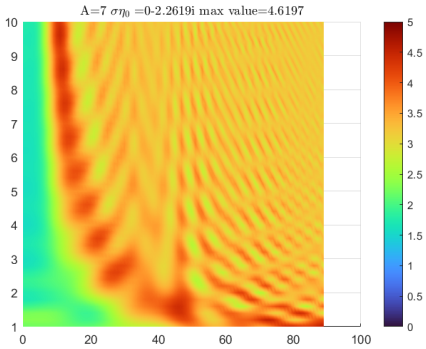


(g)

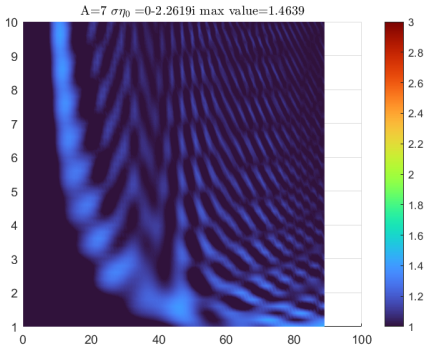


(h) Normalised

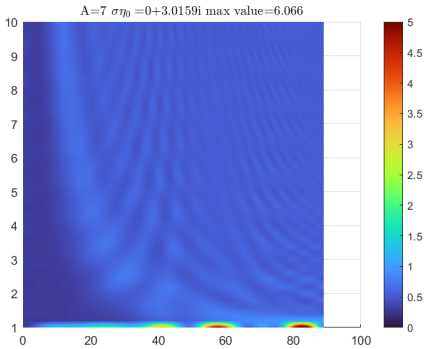
Figure A.4: Normalised Area = 4, Extensive Parametric Study



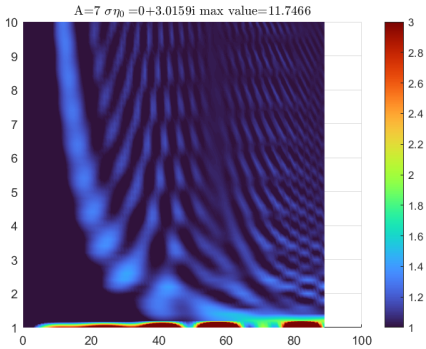
(a)



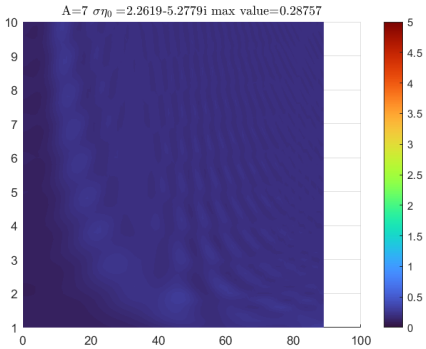
(b) Normalised



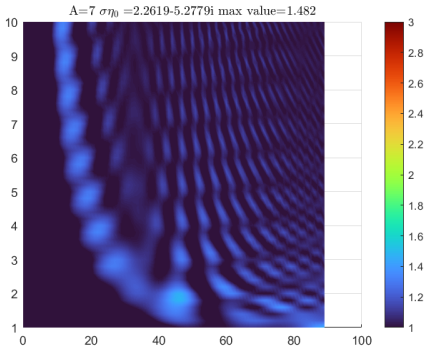
(c)



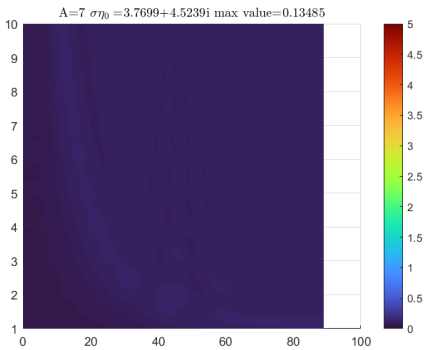
(d) Normalised



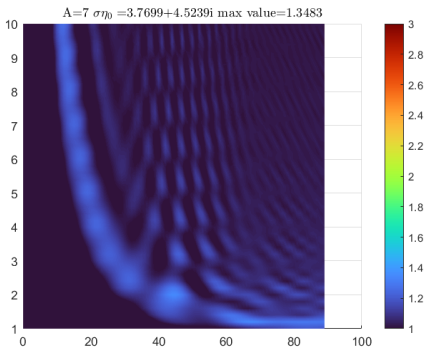
(e)



(f) Normalised

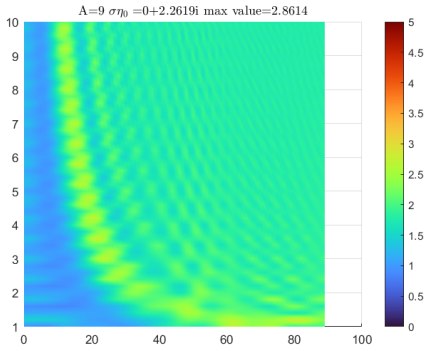


(g)

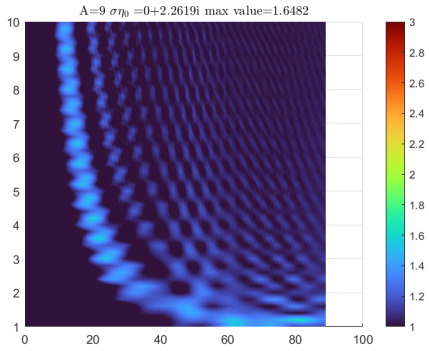


(h) Normalised

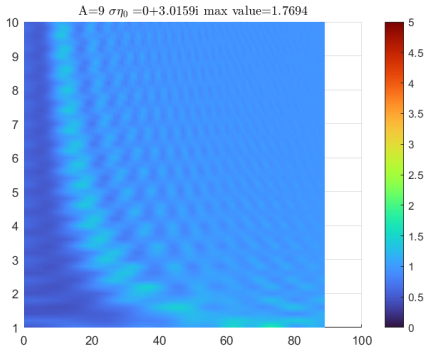
Figure A.5: Normalised Area = 7, Extensive Parametric Study



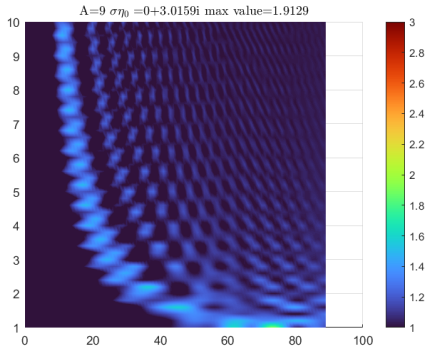
(a)



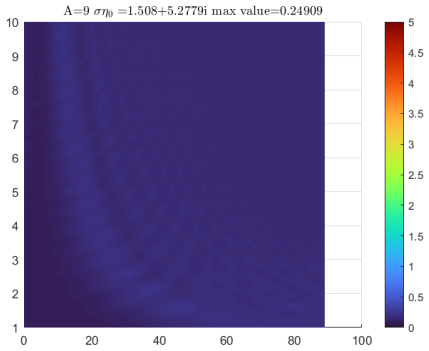
(b) Normalised



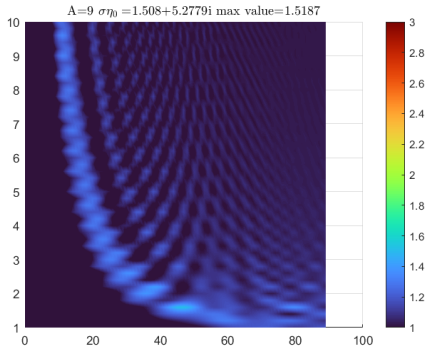
(c)



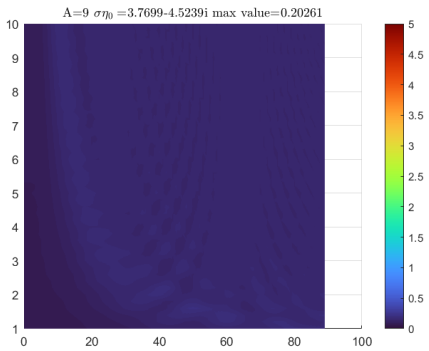
(d) Normalised



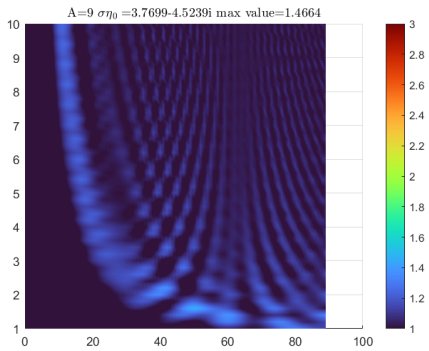
(e)



(f) Normalised



(g)



(h) Normalised

Figure A.6: Normalised Area = 9, Extensive Parametric Study

Appendix B

Code implemented in Matlab

The documentation of the code in Matlab is depicted on the next pages.

1 File Index

1.1 File List

Here is a list of all files with brief descriptions:

2 File Documentation

2.1 arc_length.m File Reference

Variables

- function [l]

2.2 area_C.m File Reference

Functions

- disp ('Done and dusted')
- end if isempty (A) A
- params1 (1,:)
- params1 (2,:)
- params2 (1,:)
- params2 (2,:)
- disp (['Area1:' num2str(Area1)])
- A/pi sqrt ()
- params1 findCentroid ()
- Area1 - A abs ()
- c (2)
- c (3)

Variables

- function [c, ceq]
- end if nargin
- param1 = params(1,1:l/2)
- param2 = params(1,l/2+1:end)
- l1 =length(param1)
- l2 =length(param2)
- Area1 = calc_Area(params1)
- Area2 = calc_Area(params2)
- dist = 5∗r
- cen2 = findCentroid(params2)
- nDist = sqrt((cen2(1)-cen1(1)).^2+(cen2(2)-cen1(2)).^2)
- ceq =[ ]

2.3 calc_Area.m File Reference

Functions

- disp ('Done and dusted')
- end if isempty (A) A
- if isnumeric (geom) % new model here when needed % geomNew(1
- if geomNew (2,:)

Variables

- function Area
- end if nargin
- end geom = convertNRB(geom)
- t0 =0:0.00001:1
- fnvals = fntlr(geom,2,t0)
- len = length(fnvals)

2.4 convertNRB.m File Reference

Variables

- function [curve_rep2]
- if curve_rep1 form

2.5 create_circle.m File Reference

2.6 curve_area.m File Reference

Variables

- function [val]
- end val = integral(@func,crv.knots(crv.order),crv.knots(end-crv.order+1))
- function y

2.7 curve_centroid.m File Reference

Variables

- function [c]
- end Mx = 0.5∗integral(@funcx,crv.knots(crv.order),crv.knots(end-crv.order+1))
- if ∼only_x My = -0.5∗integral(@funcy,crv.knots(crv.order),crv.knots(end-crv.order+1))
- c = [Mx,My]./area
- function y

2.8 domain_integral.m File Reference

Functions

- geometryFromEdges (m, g)

Variables

- function [value]
- persistent t0 = 0:step:1
- persistent sigma_val = sigma_value
- persistent step = step_val
- persistent coefs = [1, repmat([4,2],1,floor((nPoints-2)/2)), 4, 1]
- persistent lambda = lambda_value
- persistent k0 = 2∗pi/lambda
- persistent h0 = 120∗pi
- switch nargin case n = length(t)-1
- l = arc_length(domain)
- p =fnval(domain,t(1:n))
- g = [2,n,p(1,:),p(2,:)]'
- m = createpde
- try mesh = generateMesh(m,'GeometricOrder','linear','hmax',l/n)
- catch value = 0
- return
- end end end pnts_in = n+1:length(mesh.Nodes)
- area = 0
- nn = length(pnts_in)
- nm = length(mesh.Elements)
- Eback = E_back(mesh.Nodes(:,pnts_in),lambda)
- p0 =fntlr(domain,2,t0)
- e0 =fnval(Ein,t0)
- for i
- end vals = Green_vals.∗e0.∗sqrt(p0(3,:).^2+p0(4,:).^2)
- E_vals = Eback - 1i∗k0∗sigma_val∗h0∗step/3∗coefs∗transpose(vals)
- t = 0:0.01:1
- nPoints = length(t0)

2.9 E_back.m File Reference

Variables

- function [res]
- switch nargin case point source k = 2
- kind nu = 0
- order H0 = besselh(nu,k,k0∗sqrt((p(1,:)+L_plus_r).^2+p(2,:).^2))
- current = -4/(k0∗120∗pi)
- res = -k0∗120∗pi/4∗current∗H0

2.10 E_back_theta.m File Reference

Variables

- function [res]
- persistent k0 = 2∗pi/lambda∗[cos(theta),sin(theta)]
- otherwise TM plane with unitary amplitude res = exp(-1i∗k0∗p)

2.11 findCentroid.m File Reference

Functions

- params, 2.5 ∗r, 0 freeform_geom ()

Variables

- function c
- curve = convertNRB(curve)
- t0 =0:0.0001:1
- fnvals = fntlr(curve,2,t0)
- curveShape = polyshape(fnvals(1,:),fnvals(2,:),'Simplify',false)

2.12 findingDoF.m File Reference

Functions

- params1 (1,:)
- params1 (2,:)
- params2 (1,:)
- params2 (2,:)
- nrbctrlplot (curve2)
- hold on nrbctrlplot (curve1)
- hold on paramStudy2 ([ ], 1.5, dist)

Variables

- clear close all addpath C
- params = 0.3∗ones(1,12)
- l = length(params)
- param2 = params(1,1:l/2)
- param1 = params(1,l/2+1:end)
- l1 =length(param1)
- l2 =length(param2)
- r1 = 1.4
- r = sqrt(1.5/pi)
- dist = 5∗r
- curve1 = freeform_geom_withArea(params1,2.5∗r,1.5,[0;0])
- curve2 = freeform_geom_withArea(params2,2.5∗r,1.5,[dist;0])
- area
- hold on
- Q = paramStudy2(params)

2.13 freeform_geom.m File Reference

Functions

- if isempty (params(params >1|params< 0)) n
- r ∗ sin (phi)]+dist

Variables

- function [geom]
- r = max_r∗(0.99∗params(1,:)+0.01)
- phi = (params(2,:)+(0:n-1))∗2∗pi/n
- ctrl_pts = [r.∗cos(phi)
- geom = nrbmak(ctrl_pts,0:1/(n+6):1)
- normalize knot vector geom knots = (geom.knots-geom.knots(1))./(geom.knots(end)-geom.knots(1))

2.14 freeform_geom_withArea.m File Reference

Functions

- if isempty (params(params >1|params< 0)) &&max_r >0 n
- while abs (area-area0)>area_tol pts1
- r ∗ sin (phi)]+dist

Variables

- function [geom]
- area_tol =1e-4
- diff_tol =1e-6
- ctrl_pts = compute_ctrl_pts(params,max_r)
- tmpgeom = nrbmak(ctrl_pts,0:1/(n+6):1)
- area = curve_area(tmpgeom)
- pts2 = compute_ctrl_pts(params,max_r+diff_tol/2)
- g1 = nrbmak(pts1,0:1/(n+6):1)
- g2 = nrbmak(pts2,0:1/(n+6):1)
- darea = (curve_area(g2)-curve_area(g1))/diff_tol
- max_r = max_r - (area-area0)/darea
- end geom = tmpgeom
- normalize knot vector geom knots = (geom.knots-geom.knots(1))./(geom.knots(end)-geom.knots(1))
- phi = (params(2,:)+(0:n-1))∗2∗pi/n
- res = [r.∗cos(phi)

2.15 freeform_x_sym_withArea.m File Reference

Functions

- if isempty (params(params >1|params< 0)) &&max_r >0 n
- while abs (area-area0)>area_tol[pts1
- r (2:end-1).∗sin(phi)]
- if res (2, 2)< res(2
- if && res (1, 1) > r(1) a
- if params (2, 1) ∗res(2

Variables

- function [geom, cx]
- cx = 0
- x coord of area centroid area_tol =1e-4
- diff_tol =1e-6
- tmpgeom = nrbmak(ctrl_pts,knots)
- area = curve_area(tmpgeom)
- while knots1 = compute_ctrl_pts(params,max_r-diff_tol/2)
- g1 = nrbmak(pts1,knots1)
- g2 = nrbmak(pts2,knots2)
- darea = (curve_area(g2)-curve_area(g1))/diff_tol
- max_r = max_r - (area-area0)/darea
- end geom = tmpgeom
- n0 = length(params)
- phi = (params(2,2:end-1)+(0:n-3))∗(pi/(n-2) - pi/180)+pi/180
- res = [r(2:end-1).∗cos(phi)

2.16 Green.m File Reference

Variables

- function [res]
- k = 2
- kind nu = 0
- order H0 = besselh(nu,k,k0∗sqrt((p(1,:)-p0(1)).^2+(p(2,:)-p0(2)).^2))
- res = -1i/4∗H0

2.17 igabem_single_geometry.m File Reference

Functions

- LHS (i,:)
- if greville (i) -0 >0 %quad1
- @integ_fun, greville(i true ()
- end LHS (i, k)
- else b (i)
- end end cond (RBases+LHS) e_in_b
- if ∼isempty (tv) if tv<
- else g (i)

Variables

- function [e_in, solution]
- end k0 = 2∗pi/lambda
- h0 = 120∗pi
- omega = 3e8 ∗ 2∗pi/lambda / 1e-9
- sigma_val = sigma(omega)
- greville = aveknt(geometry.knots,geometry.order)
- DoFs = geometry.number
- N = spcol(geometry.knots,geometry.order,greville)
- RBases = (geometry.coefs(3,:).∗N)./(N∗geometry.coefs(3,:)')
- Rational bases greville LHS = zeros(DoFs,DoFs)
- b = zeros(DoFs,1)
- if mode t0 =0:step:1
- nPoints = length(t0)
- Rational bases greville Nt = spcol(geometry.knots,geometry.order,t0)
- Rt = (geometry.coefs(3,:).∗Nt)./(Nt∗geometry.coefs(3,:)')
- fnvals = fntlr(geometry,2,t0)
- coefs = [1, repmat([4,2],1,floor((nPoints-2)/2)), 4, 1]
- end fngvals = fntlr(geometry,1,greville)
- for i
- if mode vals = Green(g,fnvals(1:2,:),lambda).∗Rt'.∗sqrt(fnvals(3,:).^2+fnvals(4,:).^2)
- else for k
- quad2 = 0
- quad1 =quadgk(@integ_fun,0,greville(i))
- end end if mode
- e_in = rsmak(geometry.knots,[transpose(e_in_b).∗geometry.coefs(3,:);geometry.coefs(3,:)])
- if mode solution = Rt∗e_in_b
- res = Green(g,vals(1:2,:),lambda).∗Rt(:,k)'.∗sqrt(vals(3,:).^2+vals(4,:).^2)

2.18 igabem_two_tubes.m File Reference

Typedefs

- using nPoints = length(t0)

Functions

- end DoFs (i)
- if find (curves(i).coefs(3,:)∼=1) RBases
- LHS (i, 1:DoFs(1))
- LHS (i, DoFs(1)+1:DoFsum)
- else b (i)
- LHS (i+DoFs(1), 1:DoFs(1))
- LHS (i+DoFs(1), DoFs(1)+1:DoFsum)
- else b (DoFs(1)+i)
- end end LHS (1:DoFs(1), 1:DoFs(1))
- LHS (DoFs(1)+1:DoFsum, DoFs(1)+1:DoFsum)
- if ∼isempty (tv) if tv<
- else g (j)

Variables

- function [e, solutions]
- for i
- end end k0 = 2∗pi/lambda
- h0 = 120∗pi
- t0 =0:step:1
- used only when mode
- if mode greville {i} = shift_greville(greville{i},t0)
- N {i} = spcol(curves(i).knots,curves(i).order,greville{i})
- else RBases {i} = (curves(i).coefs(3,:).∗N{i})./(N{i}∗curves(i).coefs(3,:)')
- Rational bases greville end fngvals {i} = fntlr(curves(i),1,greville{i})
- end DoFsum = sum(DoFs)
- LHS = zeros(DoFsum,DoFsum)
- b = zeros(DoFsum,1)
- else Rt {i} = (curves(i).coefs(3,:).∗Nt{i})./(Nt{i}∗curves(i).coefs(3,:)')
- end fnvals {i} = fntlr(curves(i),2,t0)
- end coefs = [1, repmat([4,2],1,floor((nPoints-2)/2)), 4, 1]
- if mode g1_1 =Green(g,fnvals{1}(1:2,:),lambda)
- g1_2 =Green(g,fnvals{2}(1:2,:),lambda)
- chlenix = sqrt(fnvals{1}(3,:).^2+fnvals{1}(4,:).^2)
- vals1 = Green(g,fnvals{1}(1:2,:),lambda).∗Rt{1}'.∗sqrt(fnvals{1}(3,:).^2+fnvals{1}(4,:).^2)
- vals2 = Green(g,fnvals{2}(1:2,:),lambda).∗Rt{2}'.∗sqrt(fnvals{2}(3,:).^2+fnvals{2}(4,:).^2)
- if mode g2_1 =Green(g,fnvals{1}(1:2,:),lambda)
- g2_2 =Green(g,fnvals{2}(1:2,:),lambda)
- e_b12 = LHS\b
- e_b {1} = e_b12(1:DoFs(1))
- if mode solutions {i} = Rt{i}∗e_b{i}
- for j

2.19 insertKnots.m File Reference

Variables

- function new_curve
- uniqueKnots =unique(curve.knots(k:end-k+1))
- l = length(uniqueKnots)-1
- knotsIn = zeros(1,l∗n)
- for i
- for j

2.20 main.m File Reference

Functions

- area_C ([ ], A)
- calc_Area ([ ], A)
- paramStudy ([ ], A, dist)

Variables

- clear clc addpath C
- r = sqrt(A/pi)
- dist = 5∗r
- fun = @paramStudy
- i = 16
- x0 = rand(2,i)
- A = [ ]
- b = [ ]
- Aeq = [ ]
- beq = [ ]
- lb = zeros(2,i)
- ub = ones(2,i)
- nonlcon = @area_C
- options = optimoptions('ga','Display','iter','UseParallel',true,PopulationSize=320)
- Q = ga(fun,32,A,b,Aeq,beq,lb,ub,nonlcon,options)

2.21 parametric_study.m File Reference

Functions

- Simpson s rule with varying signal angle electrical conductivities sigma (2)=0.005 - 1i ∗0.005
- sigma (1)
- domain_integral (ref_circle,[ ], lambda, sigma(1), step) ref_value
- circle (2)
- curves (j)
- E_back_theta ([ ], lambda, angles(j))
- Q2 (i, j)
- integrals (i, j)
- figure surf (X, Y, integrals, 'FaceColor', 'interp')
- xlabel('Angle') % ylabel('Distance') % title(['Q1+Q2 value at A/\lambda^2 colorbar view ([0, 0, 1]) % figure % surf(X

Variables

- function [integrals]
- integration step step = 1e-4
- integration scheme mode = 3
- circle area over lambda squared A = [1.5,5]
- circle radius for a
- r = sqrt(A(a)/pi)/lambda
- distance between circles dist = 2.1∗r:0.5∗r:10∗r
- nDist = length(dist)
- angles angles = (0:45:90)∗pi/180
- nAngle = length(angles)
- integrals = zeros(nDist,nAngle)
- Q1 = integrals
- Q2 = integrals
- ref_circle = create_circle(r,[0,0],8,true)
- e = igabem_single_geometry(ref_circle,lambda,sigma(1),step,1)
- for i
- for j
- e1 =igabem_two_tubes(curves,lambda,sigma,step,mode)
- xlabel('Angle') % ylabel('Distance') % title(['Q1+Q2 value at A/\lambda^2 colorbar Y
- xlabel('Angle') % ylabel('Distance') % title(['Q1+Q2 value at A/\lambda^2 colorbar integrals ref_value
- xlabel('Angle') % ylabel('Distance') % title(['Q1+Q2 value at A/\lambda^2 colorbar integrals FaceColor
- xlabel('Angle') % ylabel('Distance') % title(['Q1+Q2 value at A/\lambda^2 colorbar integrals interp

2.22 paramStudy.m File Reference

Functions

- params1 (1,:)
- params1 (2,:)
- params2 (1,:)
- params2 (2,:)
- Simpson s rule with varying signal angle electrical conductivities sigma (2)=0.002+1i ∗0.004
- sigma (1)
- angle pi ()
- angles length ()
- else circles (i)
- domain_integral (ref_circle,[ ], lambda, sigma(1), step)
- circle (2)
- circle (1)
- curves (j)
- end disp (['Calculating distance:' num2str(dist) ' at angle:' num2str(angles)])

Variables

- function [sum]
- param1 = params(1,1:l/2)
- param2 = params(1,l/2+1:end)
- l1 =length(param1)
- l2 =length(param2)
- wavelength lambda = 1
- integration scheme mode = 3
- circle radius r = sqrt(A_lambda/pi)
- distance between circles CHANGE HERE distance between circles nDist = length(dist)
- integrals = zeros(nDist,nAngle)
- for i
- end ref_circle = create_circle(r,[dist,0],5,true)
- e = igabem_single_geometry(ref_circle,lambda,sigma(1),step,1)
- ref_value
- e2 =igabem_two_tubes(circles,lambda,sigma,step,mode)
- ref1
- for j
- e1 =igabem_two_tubes(curves,lambda,sigma,step,mode)
- Q1

2.23 plotSolutionIn.m File Reference

Variables

- function [objvals, areas]

2.24 run_code.m File Reference

Functions

- catch disp (['Error processing IGES file:' filename])
- plot the curves figure nrbctrlplot (curve1) hold on nrbctrlplot(curve2) %assume also a wavelength equal to 100nm %lambda
- you can calculate its value at a point by calling the function E_back as E_back (p, lambda) lambda
- used for integration with Simpson s rule (when mode is 1) omega

2.25 test.m File Reference

Functions

- params1 (1,:)
- params1 (2,:)
- params2 (1,:)
- params2 (2,:)
- nrbctrlplot (curve2)
- nrbctrlplot (curve1)

Variables

- clear close all addpath C
- params = rand(1,64)
- l = length(params)
- param1 = params(1,1:l/2)
- param2 = params(1,l/2+1:end)
- l1 =length(param1)
- l2 =length(param2)
- r1 = 1.4
- Area = 3
- r = sqrt(Area/pi)
- dist = 1.4∗r
- angle = 60
- curve1 = freeform_geom_withArea(params1,2.5∗r,Area,[0;0])
- curve2 = freeform_geom_withArea(params2,2.5∗r,Area,[dist;0])
- area
- hold on
- t0 =0:0.0001:1
- fnvals1 = fntlr(curve1,2,t0)
- fnvals2 = fntlr(curve2,2,t0)
- curveShape1 = polyshape(fnvals1(1,:),fnvals1(2,:),'Simplify',false)
- curveShape2 = polyshape(fnvals2(1,:),fnvals2(2,:),'Simplify',false)
- normal

2.26 test_circle.m File Reference

Functions

- tiledlayout (3, 3) for i
- nexttile nrbctrlplot (circle)

Variables

- clear clc adjust this path to the location of your NURBS toolbox folder addpath C

2.27 test_new_pmodel.m File Reference

Functions

- Assume you need a distance d between the centroids of the two shapes g (1).coefs(1
- Assume you need a distance d between the centroids of the two shapes g (2).coefs(1
- Assume you need a distance d between the centroids of the two shapes nrbctrlplot (convertNRB(g(1)))
- hold on nrbctrlplot (convertNRB(g(2)))

Variables

- replace the path with your own path to NURBStoolbox addpath C
- replace the path with your own path to NURBStoolbox addpath end d = 3

2.28 testIGES.m File Reference

Functions

- 'new1.igs' iges2matlab ()
- Plot the IGES object plotIGES (ParameterData, 1)

Variables

- clear clc addpath C

2.29 testSym.m File Reference

Functions

- params1 (1,:)
- params1 (2,:)
- params2 (1,:)
- params2 (2,:)
- nrbctrlplot (curve2)

Variables

- clear close all addpath C
- params = [0.98554 0.907933 0.0181202 0.72593 0.249302 0.301766 0.979765 0.945459 0.8729 0.↩ 0478817 0.024112 0.0358854 0.524259 0.0356903 0.274254 0.568405 0.710389 0.828933 0.656368 0.218204 0.898869 0.0572068 0.977653 0.0344837 0.529439 0.0876993 0.920246 0.729349 0.↩ 855365 0.393696 0.373365 0.473583 0.957865 0.760406 0.0334215 0.000840704 0.984036 0.0038241 0.00949617 0.9757 0.0156119 0.0135596 0.384359 0.00385976 0.238079 0.00874129 0.0269137 0.980579 0.635922 0.439442 0.310683 0.901094 0.409907 0.241837 0.877043 0.898044 0.338573 0.323907 0.53254 0.397702 0.0527111 0.848642 0.516472 0.748854]
- l = length(params)
- param1 = params(1,1:l/2)
- param2 = params(1,l/2+1:end)
- l1 =length(param1)
- l2 =length(param2)
- r1 = 1.4
- Area = 1
- r = sqrt(Area/pi)
- dist = 1.4∗r
- angle = 88
- curve1 = freeform_x_sym_withArea(params1,2.5∗r,Area,[0;0])
- curve2 = freeform_x_sym_withArea(params2,2.5∗r,Area,[dist;0])
- area
- hold on
- hold on Q = paramStudy_sym(params,Area,dist,angle)

Appendix C

Code implemented in C++

The documentation of the code in C++ is depicted on the next pages.

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Class Documentation

4.1 analysis_curves Class Reference

`#include <analysis_NURBSCurve.h>`

Public Member Functions

- analysis_curves ()
  default constructor
- ~analysis_curves ()
  default destructor
- unsigned int nCurves () const
  get number of surfaces in model
- vector< unsigned int > DoFs (bool CAD=false) const
  get number of DoFs
- const SplineCurve & operator[ ] (unsigned int i) const
  get the ith curve from the collection of curves in the CAD model
- const SplineCurve & getAnalysisCurve (unsigned int i) const
  get the ith curve from the collection of curves in the analysis model
- const SplineCurve & getSolution (unsigned int i) const
  get the solution for the ith curve
- void translateCurve (Point translation_vector, unsigned int curve_index)
  translate the curve in position curve_index for both CAD and analysis models
- void addCurve (SplineCurve &curve)
  add a curve in the CAD model
- void addAnalysisCurve (SplineCurve &curve)
  add analysis curve in the analysis model
- void addGrevilleAbscissae (double *vals, int size)
  add greville Abscissae
- double GetGrevilleAbscissa (unsigned int i, unsigned int curve_index=0)
  get greville Abscissa
- bool isCurveClosed (double tolerance=1e-6, unsigned int curve_index=0)
  check if curve at curve_index is closed
- double CurveArea (unsigned int curve_index=0)
  get the area of closed curved (return -1, if curve is not closed)

- double CurveArea (const SplineCurve &curve)

  get the area of closed curved stroed in a GoTools SplineCurve structure (return -1, if curve is not closed)
- double CurveLength (unsigned int curve_index=0)

  get length of curve at curve_indexS
- double CurveArcLength (double start_param=0., double end_param=1., unsigned int curve_index=0)

  get arc length between start_param and end_param for curve at curve_index
- Point CurveCentroid (const SplineCurve &curve)

  get closed curve centroid
- void Circle (unsigned int ns=3, double r=1, vector< double > v={ 0, 0 }, unsigned int curve_index=0)

  generate a circle represented as a quadratic NURBS curve with ns (ns>2) number of circular segments; r corresponds to circle's radius and v to the translation vector
- void CurveFromParameters_pm1 (vector< double > parameters, double r_max, vector< double > v={ 0, 0 }, unsigned int curve_index=0)

  generate a closed curve of arbitrary shape using the 1st parametric model at position curve_index
- double CurveFromParameters_pm2 (vector< double > parameters, double r_max, double area0, vector< double > v={ 0, 0 }, unsigned int curve_index=0)

  generate a closed curve of arbitrary shape using the 2nd parametric model (with given enclosed area) at position curve_index
- double CurveFromParameters_pm3 (vector< double > parameters, double r_max, double area0, vector< double > v={ 0, 0 }, unsigned int curve_index=0)
- void CurveFromParameters_pm3_m (vector< double > parameters, double r_max, double area0, vector< double > v={ 0, 0 }, unsigned int curve_index=0)

  Meruyert's implementation of pm3; not fully tested.
- void RefineCurve (unsigned int knots_per_span, unsigned int curve_index=0)
- double ComputeBasisValue (double param, unsigned int b_id, unsigned int curve_index=0)

  Analysis model: compute basis value at 'param' parametric value (return rational basis value, if curve is rational)
- valarray< valarray< double > > ComputeBasisValues (valarray< double > params, unsigned int curve_index=0)

  Analysis model: compute basis values at 'params' parametric values (return rational basis values, if curve is rational)
- vector< vector< Point > > ComputeCurvePoints (valarray< double > params, unsigned int derivs=0, unsigned int curve_index=0)

  CAD model: compute curve's points and derivatives at 'params' parametric values.
- valarray< complex< double > > ComputeSolutionValues (valarray< double > params, unsigned int curve_index=0)

  Analysis model: compute solution at 'params' parametric values.
- void ComputeFieldOnSingleCurve (double wavelength, double angle, complex< double > sigma, unsigned int curve_index=0, double step=1e-4)

  compute electric field on single curve from a given background field using Simpson's integration
- void ComputeFieldOnSingleCurve_TBB (double wavelength, double angle, complex< double > sigma, unsigned int curve_index=0, double step=1e-4)

  compute electric field on single curve from a given background field using threads and Simpson's integration
- void ComputeFieldOnTwoCurves_TBB (double wavelength, double angle, vector< complex< double > >sigma, vector< unsigned int > curve_index, double step=1e-4)

  compute electric field on two curves from a given background field using threads and Simpson's integration
- void ComputeFieldOnSingleCurve_GSL (double wavelength, double angle, complex< double > sigma, unsigned int curve_index=0)

  compute electric field on single curve from a given background field using GSL integration
- void ComputeFieldOnSingleCurve_MPI (double wavelength, double angle, complex< double > sigma, unsigned int curve_index=0, double step=1e-4)

  compute electric field on single curve from a given background field with OpenMPI and Simpson's integration
- bool read_iges_file (const char *filename)

  read curve from IGES file
- bool save_analysis_model (const char *filename, bool IGES=true)

save analysis curves in IGES file

- string PrintSolution (unsigned int curve_index=0)

  return a string with the solution's control values on the curve boundary
- int memory_size ()

  compute memory used for curves storage
- double GetGreville (unsigned int i, unsigned int j)

  get i greville Abscissa from analysis model j
- string PrintCurveInfo (unsigned int i, bool analysis=false)

  print curve info
- void CalculateGreville ()

  calculate Greville Abscissae and Points
- double CalculateQuantityOnMesh (double wavelength, double angle, complex< double > sigma, unsigned int curve_index=0, double step=1e-4, string filename="")

  calculate integral of E inside a single tube when a single tube is present (using Simpson's intergration and TBB)
- double CalculateQuantityOnMesh_GSL (double wavelength, double angle, complex< double > sigma, unsigned int curve_index=0, double tolerance=1e-4, string filename="")

  calculate integral of E inside a single tube when a single tube is present (using GSL adaptive integration intergration and TBB)
- double CalculateQuantityOnMeshFromTwoTubes (double wavelength, double angle, vector< complex< double > > sigma, vector< unsigned int > curve_index, double step=1e-4, string filename="")

  calculate integral of E inside a single tube when two tubes are present (using Simpson's intergration and TBB)
- double CalculateQuantityOnMeshFromTwoTubes_GSL (double wavelength, double angle, vector< complex< double > > sigma, vector< unsigned int > curve_index, double tolerance=1e-3, string filename="")

  calculate integral of E inside a single tube when two tubes are present (using GSL adaptive intergration and TBB)

Private Member Functions

- SplineCurve ComputeSplineCurve (vector< double > params, double rv, vector< double > knots, unsigned int k, unsigned int n, vector< double > v={ 0, 0 })

  create a spline curve using the pm1 parametric model (params are parametric model's parameters, rv the maximum radius and n should be equal to params.size()/2; k:order, knots:knotvector)
- SplineCurve ComputeSymSplineCurve (vector< double > params, double rv, unsigned int k, vector< double > v={ 0, 0 })

  create a spline curve using the symmetric parametric model (params are parametric model's parameters, rv the maximum radius and n should be equal to params.size()/2; k:order, knots:knotvector)
- SplineCurve ComputeSplineCurve_pm3 (vector< double > params, double rv, vector< double > knots, unsigned int k, unsigned int n, vector< double > v={ 0, 0 })

  Meruyert's implementation of pm3 NR function; not fully tested.
- void updateAnalysisModel ()
- void shiftGreville (valarray< double > t, unsigned int curve_index)

  shift Greville Abscissae to avoid singularities when using Simpson's integration

Private Attributes

- vector< SplineCurve > m_cad_model

  initial NURBS CAD model
- vector< SplineCurve > m_analysis_model

  curve model after refinement used for analysis
- vector< SplineCurve > m_e

  solution vector (electric field on analysis_model)
- vector< valarray< double > > m_greville

  Greville abscissae.
- vector< vector< Point > > m_grevillePoints

  Greville points.

4.1.1 Constructor & Destructor Documentation

4.1.1.1 analysis_curves()

```
analysis_curves::analysis_curves ( )  [inline]
```

default constructor

4.1.1.2 ~analysis_curves()

```
analysis_curves::~analysis_curves ( )  [inline]
```

default destructor

4.1.2 Member Function Documentation

4.1.2.1 addAnalysisCurve()

```
void analysis_curves::addAnalysisCurve (
              SplineCurve & curve )  [inline]
```

add analysis curve in the analysis model

4.1.2.2 addCurve()

```
void analysis_curves::addCurve (
              SplineCurve & curve )  [inline]
```

add a curve in the CAD model

4.1.2.3 addGrevilleAbscissae()

```
void analysis_curves::addGrevilleAbscissae (
              double * vals,
              int size )  [inline]
```

add greville Abscissae

4.1.2.4 CalculateGreville()

```
void analysis_curves::CalculateGreville ( )
```

calculate Greville Abscissae and Points

4.1.2.5 CalculateQuantityOnMesh()

```
double analysis_curves::CalculateQuantityOnMesh (
              double wavelength,
              double angle,
              complex< double > sigma,
              unsigned int curve_index = 0,
              double step = 1e-4,
              string filename = "" )
```

calculate integral of E inside a single tube when a single tube is present (using Simpson's intergration and TBB)

4.1.2.6 CalculateQuantityOnMesh_GSL()

```
double analysis_curves::CalculateQuantityOnMesh_GSL (
              double wavelength,
              double angle,
              complex< double > sigma,
              unsigned int curve_index = 0,
              double tolerance = 1e-4,
              string filename = "" )
```

calculate integral of E inside a single tube when a single tube is present (using GSL adaptive integration intergration and TBB)

4.1.2.7 CalculateQuantityOnMeshFromTwoTubes()

```
double analysis_curves::CalculateQuantityOnMeshFromTwoTubes (
              double wavelength,
              double angle,
              vector< complex< double > > sigma,
              vector< unsigned int > curve_index,
              double step = 1e-4,
              string filename = "" )
```

calculate integral of E inside a single tube when two tubes are present (using Simpson's intergration and TBB)

4.1.2.8 CalculateQuantityOnMeshFromTwoTubes_GSL()

```
double analysis_curves::CalculateQuantityOnMeshFromTwoTubes_GSL (
              double wavelength,
              double angle,
              vector< complex< double > > sigma,
              vector< unsigned int > curve_index,
              double tolerance = 1e-3,
              string filename = "" )
```

calculate integral of E inside a single tube when two tubes are present (using GSL adaptive intergration and TBB)

4.1.2.9 Circle()

```
void analysis_curves::Circle (
              unsigned int ns = 3,
              double r = 1,
              vector< double > v = { 0,0 },
              unsigned int curve_index = 0 )
```

generate a circle represented as a quadratic NURBS curve with ns (ns>2) number of circular segments; r corresponds to circle's radius and v to the translation vector

4.1.2.10 ComputeBasisValue()

```
double analysis_curves::ComputeBasisValue (
              double param,
              unsigned int b_id,
              unsigned int curve_index = 0 )
```

Analysis model: compute basis value at 'param' parametric value (return rational basis value, if curve is rational)

4.1.2.11 ComputeBasisValues()

```
valarray< valarray< double > > analysis_curves::ComputeBasisValues (
              valarray< double > params,
              unsigned int curve_index = 0 )
```

Analysis model: compute basis values at 'params' parametric values (return rational basis values, if curve is rational)

4.1.2.12 ComputeCurvePoints()

```
vector< vector< Point > > analysis_curves::ComputeCurvePoints (
              valarray< double > params,
              unsigned int derivs = 0,
              unsigned int curve_index = 0 )
```

CAD model: compute curve's points and derivatives at 'params' parametric values.

4.1.2.13 ComputeFieldOnSingleCurve()

```
void analysis_curves::ComputeFieldOnSingleCurve (
              double wavelength,
              double angle,
              complex< double > sigma,
              unsigned int curve_index = 0,
              double step = 1e-4 )
```

compute electric field on single curve from a given background field using Simpson's integration

4.1.2.14 ComputeFieldOnSingleCurve_GSL()

```
void analysis_curves::ComputeFieldOnSingleCurve_GSL (
              double wavelength,
              double angle,
              complex< double > sigma,
              unsigned int curve_index = 0 )
```

compute electric field on single curve from a given background field using GSL integration

4.1.2.15 ComputeFieldOnSingleCurve_MPI()

```
void analysis_curves::ComputeFieldOnSingleCurve_MPI (
              double wavelength,
              double angle,
              complex< double > sigma,
              unsigned int curve_index = 0,
              double step = 1e-4 )
```

compute electric field on single curve from a given background field with OpenMPI and Simpson's integration

4.1.2.16 ComputeFieldOnSingleCurve_TBB()

```
void analysis_curves::ComputeFieldOnSingleCurve_TBB (
              double wavelength,
              double angle,
              complex< double > sigma,
              unsigned int curve_index = 0,
              double step = 1e-4 )
```

compute electric field on single curve from a given background field using threads and Simpson's integration

4.1.2.17 ComputeFieldOnTwoCurves_TBB()

```
void analysis_curves::ComputeFieldOnTwoCurves_TBB (
              double wavelength,
              double angle,
              vector< complex< double > > sigma,
              vector< unsigned int > curve_index,
              double step = 1e-4 )
```

compute electric field on two curves from a given background field using threads and Simpson's integration

4.1.2.18 ComputeSolutionValues()

```
valarray< complex< double > > analysis_curves::ComputeSolutionValues (
              valarray< double > params,
              unsigned int curve_index = 0 )
```

Analysis model: compute solution at 'params' parametric values.

4.1.2.19 ComputeSplineCurve()

```
SplineCurve analysis_curves::ComputeSplineCurve (
              vector< double > params,
              double rv,
              vector< double > knots,
              unsigned int k,
              unsigned int n,
              vector< double > v = { 0,0 } )  [private]
```

create a spline curve using the pm1 parametric model (params are parametric model's parameters, rv the maximum radius and n should be equal to params.size()/2; k:order, knots:knotvector)

4.1.2.20 ComputeSplineCurve_pm3()

```
SplineCurve analysis_curves::ComputeSplineCurve_pm3 (
              vector< double > params,
              double rv,
              vector< double > knots,
              unsigned int k,
              unsigned int n,
              vector< double > v = { 0,0 } )  [private]
```

Meruyert's implementation of pm3 NR function; not fully tested.

4.1.2.21 ComputeSymSplineCurve()

```
SplineCurve analysis_curves::ComputeSymSplineCurve (
              vector< double > params,
              double rv,
              unsigned int k,
              vector< double > v = { 0,0 } )  [private]
```

create a spline curve using the symmetric parametric model (params are parametric model's parameters, rv the maximum radius and n should be equal to params.size()/2; k:order, knots:knotvector)

4.1.2.22 CurveArcLength()

```
double analysis_curves::CurveArcLength (
              double start_param = 0.,
              double end_param = 1.,
              unsigned int curve_index = 0 )
```

get arc length between start_param and end_param for curve at curve_index

4.1.2.23 CurveArea() [1/2]

```
double analysis_curves::CurveArea (
              const SplineCurve & curve )
```

get the area of closed curved stroed in a GoTools SplineCurve structure (return -1, if curve is not closed)

4.1.2.24 CurveArea() [2/2]

```
double analysis_curves::CurveArea (
            unsigned int curve_index = 0 )
```

get the area of closed curved (return -1, if curve is not closed)

4.1.2.25 CurveCentroid()

```
Point analysis_curves::CurveCentroid (
            const SplineCurve & curve )
```

get closed curve centroid

4.1.2.26 CurveFromParameters_pm1()

```
void analysis_curves::CurveFromParameters_pm1 (
            vector< double > parameters,
            double r_max,
            vector< double > v = { 0,0 },
            unsigned int curve_index = 0 )
```

generate a closed curve of arbitrary shape using the 1st parametric model at position curve_index

4.1.2.27 CurveFromParameters_pm2()

```
double analysis_curves::CurveFromParameters_pm2 (
            vector< double > parameters,
            double r_max,
            double area0,
            vector< double > v = { 0,0 },
            unsigned int curve_index = 0 )
```

generate a closed curve of arbitrary shape using the 2nd parametric model (with given enclosed area) at position curve_index

4.1.2.28 CurveFromParameters_pm3()

```
double analysis_curves::CurveFromParameters_pm3 (
            vector< double > parameters,
            double r_max,
            double area0,
            vector< double > v = { 0,0 },
            unsigned int curve_index = 0 )
```

generate an x-symmetric closed curve of arbitrary shape using the 3rd parametric model (with given enclosed area) at position curve_index. It also returns the x-coordinate of the generated curve's centroid. Note: parameters.size() >= 6

4.1.2.29 CurveFromParameters_pm3_m()

```
void analysis_curves::CurveFromParameters_pm3_m (
              vector< double > parameters,
              double r_max,
              double area0,
              vector< double > v = { 0,0 },
              unsigned int curve_index = 0 )
```

Meruyert's implementation of pm3; not fully tested.

4.1.2.30 CurveLength()

```
double analysis_curves::CurveLength (
              unsigned int curve_index = 0 )
```

get length of curve at curve_indexS

4.1.2.31 DoFs()

```
vector< unsigned int > analysis_curves::DoFs (
              bool CAD = false ) const
```

get number of DoFs

4.1.2.32 getAnalysisCurve()

```
const SplineCurve & analysis_curves::getAnalysisCurve (
              unsigned int i ) const
```

get the ith curve from the collection of curves in the analysis model

4.1.2.33 GetGreville()

```
double analysis_curves::GetGreville (
              unsigned int i,
              unsigned int j )  [inline]
```

get i greville Abscissa from analysis model j

4.1.2.34 GetGrevilleAbscissa()

```
double analysis_curves::GetGrevilleAbscissa (
              unsigned int i,
              unsigned int curve_index = 0 )
```

get greville Abscissa

4.1.2.35 getSolution()

```
const SplineCurve & analysis_curves::getSolution (
              unsigned int i ) const
```

get the solution for the ith curve

4.1.2.36 isCurveClosed()

```
bool analysis_curves::isCurveClosed (
              double tolerance = 1e-6,
              unsigned int curve_index = 0 )
```

check if curve at curve_index is closed

4.1.2.37 memory_size()

```
int analysis_curves::memory_size ( )
```

compute memory used for curves storage

4.1.2.38 nCurves()

```
unsigned int analysis_curves::nCurves ( ) const  [inline]
```

get number of surfaces in model

4.1.2.39 operator[]()

```
const SplineCurve & analysis_curves::operator[ ] (
              unsigned int i ) const
```

get the ith curve from the collection of curves in the CAD model

4.1.2.40 PrintCurveInfo()

```
string analysis_curves::PrintCurveInfo (
              unsigned int i,
              bool analysis = false )
```

print curve info

4.1.2.41 PrintSolution()

```
string analysis_curves::PrintSolution (
              unsigned int curve_index = 0 )
```

return a string with the solution's control values on the curve boundary

4.1.2.42 read_iges_file()

```
bool analysis_curves::read_iges_file (
            const char * filename )
```

read curve from IGES file

4.1.2.43 RefineCurve()

```
void analysis_curves::RefineCurve (
            unsigned int knots_per_span,
            unsigned int curve_index = 0 )
```

refine the analysis model placing knots_per_span knots uniformly distributed knots for each knot span for the curve at curve_index; if no curve_index is specified, the first curve is refined.

4.1.2.44 save_analysis_model()

```
bool analysis_curves::save_analysis_model (
            const char * filename,
            bool IGES = true )
```

save analysis curves in IGES file

4.1.2.45 shiftGreville()

```
void analysis_curves::shiftGreville (
            valarray< double > t,
            unsigned int curve_index )  [private]
```

shift Greville Abscissae to avoid singularities when using Simpson's integration

4.1.2.46 translateCurve()

```
void analysis_curves::translateCurve (
            Point translation_vector,
            unsigned int curve_index )
```

translate the curve in position curve_index for both CAD and analysis models

4.1.2.47 updateAnalysisModel()

```
void analysis_curves::updateAnalysisModel ( )  [inline], [private]
```

4.1.3 Member Data Documentation

4.1.3.1 m_analysis_model

```
vector<SplineCurve> analysis_curves::m_analysis_model  [private]
```

curve model after refinement used for analysis

4.1.3.2 m_cad_model

```
vector<SplineCurve> analysis_curves::m_cad_model  [private]
```

initial NURBS CAD model

4.1.3.3 m_e

```
vector<SplineCurve> analysis_curves::m_e  [private]
```

solution vector (electric field on analysis_model)

4.1.3.4 m_greville

```
vector<valarray<double> > analysis_curves::m_greville  [private]
```

Greville abscissae.

4.1.3.5 m_grevillePoints

```
vector<vector<Point> > analysis_curves::m_grevillePoints  [private]
```

Greville points.

The documentation for this class was generated from the following files:

- analysis_NURBSCurve.h
- analysis_NURBSCurve.cpp

4.2 BIE_gsl_f_params Struct Reference

```
#include <analysis_NURBSCurve.h>
```

Public Attributes

- SplineCurve c
- SplineCurve e
- Point q
- double w
- bool isReal

4.2.1 Member Data Documentation

4.2.1.1 c

```
SplineCurve BIE_gsl_f_params::c
```

4.2.1.2 e

```
SplineCurve BIE_gsl_f_params::e
```

4.2.1.3 isReal

```
bool BIE_gsl_f_params::isReal
```

4.2.1.4 q

```
Point BIE_gsl_f_params::q
```

4.2.1.5 w

```
double BIE_gsl_f_params::w
```

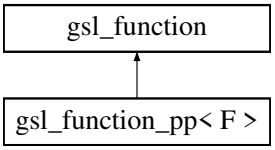
The documentation for this struct was generated from the following file:

- analysis_NURBSCurve.h

4.3 gsl_function_pp< F > Class Template Reference

```
#include <integration.h>
```

Inheritance diagram for gsl_function_pp< F >:



Public Member Functions

- gsl_function_pp (const F &f)
- operator gsl_function * ()

Static Private Member Functions

- static double invoke (double x, void *params)

Private Attributes

- const F func

4.3.1 Constructor & Destructor Documentation

4.3.1.1 gsl_function_pp()

```
template<typename F >
gsl_function_pp< F >::gsl_function_pp (
              const F & f )  [inline]
```

4.3.2 Member Function Documentation

4.3.2.1 invoke()

```
template<typename F >
static double gsl_function_pp< F >::invoke (
              double x,
              void * params )  [inline], [static], [private]
```

4.3.2.2 operator gsl_function *()

```
template<typename F >
gsl_function_pp< F >::operator gsl_function * ( )  [inline]
```

4.3.3 Member Data Documentation

4.3.3.1 func

```
template<typename F >
const F gsl_function_pp< F >::func  [private]
```

The documentation for this class was generated from the following file:

- integration.h

4.4 IntegrationWorkspace Class Reference

```
#include <integration.h>
```

Public Member Functions

- IntegrationWorkspace (const size_t n=1000)
- ~IntegrationWorkspace ()
- operator gsl_integration_workspace * ()

Private Attributes

- gsl_integration_workspace * wsp

4.4.1 Constructor & Destructor Documentation

4.4.1.1 IntegrationWorkspace()

```
IntegrationWorkspace::IntegrationWorkspace (
              const size_t n = 1000 )  [inline]
```

4.4.1.2 ~IntegrationWorkspace()

```
IntegrationWorkspace::~IntegrationWorkspace ( )  [inline]
```

4.4.2 Member Function Documentation

4.4.2.1 operator gsl_integration_workspace *()

```
IntegrationWorkspace::operator gsl_integration_workspace * ( )  [inline]
```

4.4.3 Member Data Documentation

4.4.3.1 wsp

```
gsl_integration_workspace* IntegrationWorkspace::wsp  [private]
```

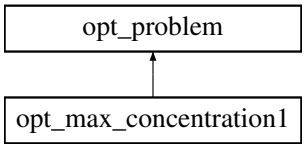
The documentation for this class was generated from the following file:

- integration.h

4.5 opt_max_concentration1 Class Reference

```
#include <opt_problems.h>
```

Inheritance diagram for opt_max_concentration1:



Public Member Functions

- opt_max_concentration1 (unsigned int problem_size=0)
- ~opt_max_concentration1 ()
- opt_max_concentration1 (unsigned int problem_size, double area)
- opt_max_concentration1 (unsigned int problem_size, double area, opt_problem_data d)
- opt_max_concentration1 (const opt_max_concentration1 &o)
- void set_problem_data (opt_problem_data d)
- void set_area_constraint (double area)
- double get_area_constraint ()
- vector_double fitness (const vector_double &v) const
- vector_double batch_fitness (const vector_double &v) const
- vector_double::size_type get_nic () const

Public Member Functions inherited from opt_problem

- opt_problem (unsigned int problem_size=0)
- ~opt_problem ()
- virtual vector_double::size_type get_nec () const
- pair< vector_double, vector_double > get_bounds () const
- void set_bounds (vector< double > lb, vector< double > ub)
- void set_problem_size (unsigned int size)

Private Attributes

- double m_area
- opt_problem_data m_data

Additional Inherited Members

Protected Attributes inherited from opt_problem

- unsigned int m_problem_size
- vector< double > m_lb
- vector< double > m_ub

4.5.1 Constructor & Destructor Documentation

4.5.1.1 opt_max_concentration1() [1/4]

```
opt_max_concentration1::opt_max_concentration1 (
            unsigned int problem_size = 0 )  [inline]
```

4.5.1.2 ~opt_max_concentration1()

```
opt_max_concentration1::~opt_max_concentration1 ( )  [inline]
```

4.5.1.3 opt_max_concentration1() [2/4]

```
opt_max_concentration1::opt_max_concentration1 (
              unsigned int problem_size,
              double area )  [inline]
```

4.5.1.4 opt_max_concentration1() [3/4]

```
opt_max_concentration1::opt_max_concentration1 (
              unsigned int problem_size,
              double area,
              opt_problem_data d )  [inline]
```

4.5.1.5 opt_max_concentration1() [4/4]

```
opt_max_concentration1::opt_max_concentration1 (
              const opt_max_concentration1 & o )
```

4.5.2 Member Function Documentation

4.5.2.1 batch_fitness()

```
vector_double opt_max_concentration1::batch_fitness (
              const vector_double & v ) const  [virtual]
```

Reimplemented from opt_problem.

4.5.2.2 fitness()

```
vector_double opt_max_concentration1::fitness (
              const vector_double & v ) const  [virtual]
```

m_data.circles;

Reimplemented from opt_problem.

4.5.2.3 get_area_constraint()

```
double opt_max_concentration1::get_area_constraint ( )  [inline]
```

4.5.2.4 get_nic()

```
vector_double::size_type opt_max_concentration1::get_nic ( ) const  [inline], [virtual]
```

Reimplemented from opt_problem.

4.5.2.5 set_area_constraint()

```
void opt_max_concentration1::set_area_constraint (
              double area )  [inline]
```

4.5.2.6 set_problem_data()

```
void opt_max_concentration1::set_problem_data (
              opt_problem_data d )  [inline]
```

4.5.3 Member Data Documentation

4.5.3.1 m_area

```
double opt_max_concentration1::m_area  [private]
```

4.5.3.2 m_data

```
opt_problem_data opt_max_concentration1::m_data  [private]
```

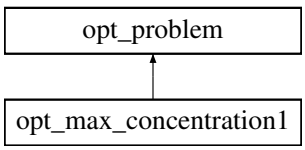
The documentation for this class was generated from the following files:

- opt_problems.h
- opt_problems.cpp

4.6 opt_problem Class Reference

`#include <opt_problems.h>`

Inheritance diagram for opt_problem:



Public Member Functions

- opt_problem (unsigned int problem_size=0)
- ~opt_problem ()
- virtual vector_double fitness (const vector_double &v) const
- virtual vector_double batch_fitness (const vector_double &v) const
- virtual vector_double::size_type get_nec () const
- virtual vector_double::size_type get_nic () const
- pair< vector_double, vector_double > get_bounds () const
- void set_bounds (vector< double > lb, vector< double > ub)
- void set_problem_size (unsigned int size)

Protected Attributes

- unsigned int m_problem_size
- vector< double > m_lb
- vector< double > m_ub

4.6.1 Constructor & Destructor Documentation

4.6.1.1 opt_problem()

```
opt_problem::opt_problem (
              unsigned int problem_size = 0 )  [inline]
```

4.6.1.2 ~opt_problem()

```
opt_problem::~opt_problem ( )  [inline]
```

4.6.2 Member Function Documentation

4.6.2.1 batch_fitness()

```
virtual vector_double opt_problem::batch_fitness (
              const vector_double & x ) const  [inline], [virtual]
```

Reimplemented in opt_max_concentration1.

4.6.2.2 fitness()

```
virtual vector_double opt_problem::fitness (
              const vector_double & x ) const  [inline], [virtual]
```

Reimplemented in opt_max_concentration1.

4.6.2.3 get_bounds()

```
pair< vector_double, vector_double > opt_problem::get_bounds ( ) const
```

4.6.2.4 get_nec()

```
virtual vector_double::size_type opt_problem::get_nec ( ) const  [inline], [virtual]
```

4.6.2.5 get_nic()

```
virtual vector_double::size_type opt_problem::get_nic ( ) const  [inline], [virtual]
```

Reimplemented in opt_max_concentration1.

4.6.2.6 set_bounds()

```
void opt_problem::set_bounds (
              vector< double > lb,
              vector< double > ub )
```

4.6.2.7 set_problem_size()

```
void opt_problem::set_problem_size (
              unsigned int size )  [inline]
```

4.6.3 Member Data Documentation

4.6.3.1 m_lb

```
vector<double> opt_problem::m_lb  [protected]
```

4.6.3.2 m_problem_size

```
unsigned int opt_problem::m_problem_size  [protected]
```

4.6.3.3 m_ub

```
vector<double> opt_problem::m_ub  [protected]
```

The documentation for this class was generated from the following files:

- opt_problems.h
- opt_problems.cpp

4.7 opt_problem_data Struct Reference

```
#include <opt_problems.h>
```

Public Attributes

- double wavelength
- double waveangle
- double area0
- double circles
- double dist
- unsigned int refinement_level

4.7.1 Member Data Documentation

4.7.1.1 area0

```
double opt_problem_data::area0
```

4.7.1.2 circles

```
double opt_problem_data::circles
```

4.7.1.3 dist

```
double opt_problem_data::dist
```

4.7.1.4 refinement_level

```
unsigned int opt_problem_data::refinement_level
```

4.7.1.5 waveangle

```
double opt_problem_data::waveangle
```

4.7.1.6 wavelength

```
double opt_problem_data::wavelength
```

The documentation for this struct was generated from the following file:

- opt_problems.h

Chapter 5

File Documentation

5.1 analysis_NURBSCurve.cpp File Reference

```
#include <mpi.h>
#include <numeric>
#include "analysis_NURBSCurve.h"
#include "integration.h"
#include "GoTools/igeslib/IGESconverter.h"
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_linalg.h>
#include <tbb/parallel_for.h>
#include <tbb/blocked_range2d.h>
```

5.2 analysis_NURBSCurve.h File Reference

```
#include <sstream>
#include <iterator>
#include <fstream>
#include <iostream>
#include <set>
#include <gmsh.h>
#include "special_functions.h"
#include "GoTools/geometry/SplineCurve.h"
```

Classes

- struct BIE_gsl_f_params
- class analysis_curves

Macros

- #define ANALYSIS_NURBSCURVE_H
- #define GOTOOLS_BASIS_VERSION

Functions

- double BIE_gsl_f (double t, void *vars)

5.2.1 Macro Definition Documentation

5.2.1.1 ANALYSIS_NURBSCURVE_H

```
#define ANALYSIS_NURBSCURVE_H
```

5.2.1.2 GOTOOLS_BASIS_VERSION

```
#define GOTOOLS_BASIS_VERSION
```

5.2.2 Function Documentation

5.2.2.1 BIE_gsl_f()

```
double BIE_gsl_f (
              double t,
              void * vars )
```

5.3 analysis_NURBSCurve.h

Go to the documentation of this file.

```
00001 #pragma once
00002 #ifndef ANALYSIS_NURBSCURVE_H
00003 #define ANALYSIS_NURBSCURVE_H
00004
00005 #include <sstream>
00006 #include <iterator>
00007 #include <fstream>
00008 #include <iostream>
00009 #include <sstream>
00010 #include <set>
00011 #include <gmsh.h>
00012 #include "special_functions.h"
00013 #include "GoTools/geometry/SplineCurve.h"
00014
00015
00016 #define GOTOOLS_BASIS_VERSION
00017 using namespace std;
00018 using namespace Go;
00019
00020 struct BIE_gsl_f_params { SplineCurve c; SplineCurve e; Point q; double w; bool isReal; };
00021
00022 double BIE_gsl_f(double t, void* vars)
00023 {
00024     struct BIE_gsl_f_params* params = (BIE_gsl_f_params*)vars;
00025     SplineCurve c = (params->c);
00026     SplineCurve e = (params->e);
00027     Point q = (params->q);
00028     double w = (params->w);
00029     bool isReal = (params->isReal);
00030     vector<Point> p(2);
00031     Point pe;
00032     c.point(p, t, 1);
00033     e.point(pe, t);
00034     complex<double> z = Green(q, p[0], w) * complex<double>(pe[0], pe[1]) * p[1].length();
00035     return isReal ? z.real() : z.imag();
00036 }
00037
```

```
00038
00039
00040
00041 class analysis_curves
00042 {
00043 public:
00045     analysis_curves() { }
00047     ~analysis_curves() { }
00049     unsigned int nCurves() const{return m_cad_model.size();}
00051     vector<unsigned int> DoFs(bool CAD = false) const;
00053     const SplineCurve& operator[](unsigned int i) const;
00055     const SplineCurve& getAnalysisCurve(unsigned int i) const;
00057     const SplineCurve& getSolution(unsigned int i) const;
00059     void translateCurve(Point translation_vector, unsigned int curve_index);
00061     void addCurve(SplineCurve& curve) { m_cad_model.push_back(curve); }
00063     void addAnalysisCurve(SplineCurve& curve) { m_analysis_model.push_back(curve); }
00065     void addGrevilleAbscissae(double* vals, int size) {m_greville.push_back(valarray<double>(vals, size));}
00067     double GetGrevilleAbscissa(unsigned int i, unsigned int curve_index = 0);
00069     bool isCurveClosed(double tolerance=1e-6, unsigned int curve_index = 0);
00071     double CurveArea(unsigned int curve_index = 0);
00073     double CurveArea(const SplineCurve& curve);
00075     double CurveLength(unsigned int curve_index = 0);
00077     double CurveArcLength(double start_param = 0., double end_param = 1., unsigned int curve_index = 0);
00079     Point CurveCentroid(const SplineCurve& curve);
00081     void Circle(unsigned int ns=3, double r = 1, vector<double> v = { 0,0 }, unsigned int curve_index = 0);
00083     void CurveFromParameters_pm1(vector<double> parameters, double r_max, vector<double> v = { 0,0 }, unsigned int
      curve_index = 0);
00085     double CurveFromParameters_pm2(vector<double> parameters, double r_max, double area0, vector<double> v = { 0,0
      }, unsigned int curve_index = 0);
00088     double CurveFromParameters_pm3(vector<double> parameters, double r_max, double area0, vector<double> v = { 0,0
      },unsigned int curve_index = 0);
00090     void CurveFromParameters_pm3_m(vector<double> parameters, double r_max, double area0, vector<double> v = { 0,0
      },unsigned int curve_index = 0);
00093     void RefineCurve(unsigned int knots_per_span, unsigned int curve_index = 0);
00095     double ComputeBasisValue(double param, unsigned int b_id, unsigned int curve_index = 0);
00097     valarray<valarray<double> > ComputeBasisValues(valarray<double> params, unsigned int curve_index = 0);
00099     vector<vector<Point> > ComputeCurvePoints(valarray<double> params, unsigned int derivs = 0, unsigned int
      curve_index = 0);
00101     valarray<complex<double> > ComputeSolutionValues(valarray<double> params, unsigned int curve_index = 0);
00103     void ComputeFieldOnSingleCurve(double wavelength, double angle, complex<double> sigma, unsigned int
      curve_index = 0, double step = 1e-4);
00105     void ComputeFieldOnSingleCurve_TBB(double wavelength, double angle, complex<double> sigma, unsigned int
      curve_index = 0, double step = 1e-4);
00107     void ComputeFieldOnTwoCurves_TBB(double wavelength, double angle, vector<complex<double> >sigma,
      vector<unsigned int> curve_index, double step = 1e-4);
00109     void ComputeFieldOnSingleCurve_GSL(double wavelength, double angle, complex<double> sigma, unsigned int
      curve_index = 0);
00111     void ComputeFieldOnSingleCurve_MPI(double wavelength, double angle, complex<double> sigma, unsigned int
      curve_index = 0, double step = 1e-4);
00113     bool read_iges_file(const char* filename);
00115     bool save_analysis_model(const char* filename, bool IGES = true);
00117     string PrintSolution(unsigned int curve_index = 0);
00119     int memory_size();
00121     double GetGreville(unsigned int i /*curve*/, unsigned int j /*Greville Abscissa index*/) { return
      m_greville[i][j];}
00123     string PrintCurveInfo(unsigned int i, bool analysis = false);
00125     void CalculateGreville();
00127     double CalculateQuantityOnMesh(double wavelength, double angle, complex<double> sigma, unsigned int
      curve_index=0, double step = 1e-4, string filename="");
00129     double CalculateQuantityOnMesh_GSL(double wavelength, double angle, complex<double> sigma, unsigned int
      curve_index = 0, double tolerance = 1e-4, string filename = "");
00131     double CalculateQuantityOnMeshFromTwoTubes(double wavelength, double angle, vector<complex<double> > sigma,
      vector<unsigned int> curve_index, double step = 1e-4, string filename = "");
00133     double CalculateQuantityOnMeshFromTwoTubes_GSL(double wavelength, double angle, vector<complex<double> >
      sigma, vector<unsigned int> curve_index, double tolerance = 1e-3, string filename = "");
00134 private:
00136     SplineCurve ComputeSplineCurve(vector<double> params, double rv, vector<double> knots, unsigned int k,
      unsigned int n, vector<double> v = { 0,0 });
00138     SplineCurve ComputeSymSplineCurve(vector<double> params, double rv, unsigned int k, vector<double> v = { 0,0
      });
00140     SplineCurve ComputeSplineCurve_pm3(vector<double> params, double rv, vector<double> knots, unsigned int k,
      unsigned int n, vector<double> v = { 0,0 });
00141     void updateAnalysisModel() { m_analysis_model = m_cad_model; CalculateGreville(); }
00143     void shiftGreville(valarray<double> t, unsigned int curve_index);
00145     vector<SplineCurve> m_cad_model;
00147     vector<SplineCurve> m_analysis_model;
00149     vector<SplineCurve> m_e;
00151     vector<valarray<double> > m_greville;
00153     vector<vector<Point> > m_grevillePoints;
00154 };
00155 #endif
```

5.4 integration.h File Reference

```
#include <gsl/gsl_integration.h>
```

Classes

- class IntegrationWorkspace
- class gsl_function_pp< F >

Macros

- #define INTEGRATION_H

Functions

- template<typename F >
  gsl_function_pp< F > make_gsl_function (const F &func)

5.4.1 Macro Definition Documentation

5.4.1.1 INTEGRATION_H

```
#define INTEGRATION_H
```

5.4.2 Function Documentation

5.4.2.1 make_gsl_function()

```
template<typename F >
gsl_function_pp< F > make_gsl_function (
              const F & func )
```

5.5 integration.h

Go to the documentation of this file.

```
00001 #pragma once
00002 #ifndef INTEGRATION_H
00003 #define INTEGRATION_H
00004 #include <gsl/gsl_integration.h>
00005
00006 class IntegrationWorkspace {
00007     gsl_integration_workspace* wsp;
00008
00009 public:
00010     IntegrationWorkspace(const size_t n = 1000) :
00011         wsp(gsl_integration_workspace_alloc(n)) {}
00012     ~IntegrationWorkspace() { gsl_integration_workspace_free(wsp); }
00013
00014     operator gsl_integration_workspace* () { return wsp; }
00015
00016 };
00017 // Build gsl_function from lambda
00018 template <typename F>
00019 class gsl_function_pp : public gsl_function {
00020     const F func;
00021     static double invoke(double x, void* params) {
00022         return static_cast<gsl_function_pp*>(params)->func(x);
00023     }
00024 public:
00025     gsl_function_pp(const F& f) : func(f) {
00026         function = &gsl_function_pp::invoke; //inherited from gsl_function
00027         params = this;                       //inherited from gsl_function
00028     }
00029     operator gsl_function* () { return this; }
00030 };
00031
00032 // Helper function for template construction
00033 template <typename F>
00034 gsl_function_pp<F> make_gsl_function(const F& func) {
00035     return gsl_function_pp<F>(func);
00036 }
00037
00038 #endif
```

5.6 opt_problems.cpp File Reference

```
#include "opt_problems.h"
#include "analysis_NURBSCurve.h"
#include <iostream>
#include <tbb/parallel_for.h>
#include <numeric>
#include "GoTools/geometry/GoIntersections.h"
#include <chrono>
```

5.7 opt_problems.h File Reference

```
#include <cmath>
#include <complex>
#include <initializer_list>
#include <iostream>
#include <utility>
#include <pagmo/problem.hpp>
#include <pagmo/types.hpp>
```

Classes

- struct opt_problem_data
- class opt_problem
- class opt_max_concentration1

Macros

- #define OPT_PROBLEMS_H

Functions

- string vector2str (vector< double > v)

5.7.1 Macro Definition Documentation

5.7.1.1 OPT_PROBLEMS_H

```
#define OPT_PROBLEMS_H
```

5.7.2 Function Documentation

5.7.2.1 vector2str()

```
string vector2str (
              vector< double > v )
```

5.8 opt_problems.h

Go to the documentation of this file.

```
00001 #pragma once
00002 #ifndef OPT_PROBLEMS_H
00003 #define OPT_PROBLEMS_H
00004
00005 #include <cmath>
00006 #include <complex>
00007 #include <initializer_list>
00008 #include <iostream>
00009 #include <utility>
00010
00011 #include <pagmo/problem.hpp>
00012 #include <pagmo/types.hpp>
00013 //#include "message_utils.h"
00014
00015 using namespace pagmo;
00016 using namespace std;
00017
00018 //problem data structure
00019 struct opt_problem_data
00020 {
00021     double wavelength;
00022     double waveangle;
00023     double area0;
00024     double circles;
00025     double dist;
00026     unsigned int refinement_level;
00027
00028 };
```

```
00029
00030 //General optimization problem with a dummy objective function and side constraints
00031 class opt_problem
00032 {
00033 public:
00034     opt_problem(unsigned int problem_size = 0) : m_problem_size(problem_size) {}
00035     ~opt_problem() {}
00036     virtual vector_double fitness(const vector_double& v) const { return { 0 }; }
00037     virtual vector_double batch_fitness(const vector_double& v) const { return { 0 }; }
00038     virtual vector_double::size_type get_nec() const { return 0; }
00039     virtual vector_double::size_type get_nic() const { return 0; }
00040     pair<vector_double, vector_double> get_bounds() const;
00041     void set_bounds(vector<double> lb, vector<double> ub);
00042     void set_problem_size(unsigned int size) { m_problem_size = size; m_lb.resize(0); m_ub.resize(0); }
00043 protected:
00044     unsigned int m_problem_size;
00045     vector<double> m_lb;
00046     vector<double> m_ub;
00047 };
00048
00049 //Single nanotube optimization problem target maximization of interior concentration subject to an area constraint
00050 class opt_max_concentration1 : public opt_problem
00051 {
00052 public:
00053     opt_max_concentration1(unsigned int problem_size = 0) : opt_problem(problem_size), m_area(0) {}
00054     ~opt_max_concentration1() {}
00055     opt_max_concentration1(unsigned int problem_size, double area) : opt_problem(problem_size), m_area(fabs(area))
    {}
00056     opt_max_concentration1(unsigned int problem_size, double area, opt_problem_data d) :
    opt_problem(problem_size), m_area(fabs(area)), m_data(d) {}
00057     opt_max_concentration1(const opt_max_concentration1& o);
00058     void set_problem_data(opt_problem_data d) { m_data = { d.wavelength, d.waveangle, d.area0, d.circles, d.dist,
    d.refinement_level}; }
00059     void set_area_constraint(double area) { m_area = fabs(area); }
00060     double get_area_constraint() { return m_area; }
00061     vector_double fitness(const vector_double& v) const;
00062     vector_double batch_fitness(const vector_double& v) const;
00063     vector_double::size_type get_nic() const { return 1; }
00064 private:
00065     double m_area;
00066     opt_problem_data m_data;
00067 };
00068
00069 string vector2str(vector<double> v) {
00070   ostringstream ss;
00071   ss « "(";
00072   unsigned int i;
00073   for (i = 0; i < v.size() - 1; i++)
00074     ss « v[i] « ",";
00075   ss « v[i] « ")";
00076   return ss.str();
00077 }
00078
00079 #endif
```

5.9 special_functions.cpp File Reference

```
#include "special_functions.h"
```

Functions

- complex< double > E_plane_back (Point p, double l, double a)
- complex< double > Green (Point p0, Point p, double wavelength)
- valarray< complex< double > > Green (Point p0, vector< Point > p, double wavelength)
- valarray< complex< double > > toComplexArray (valarray< double > a)
- valarray< valarray< double > > transposeArray (valarray< valarray< double > >a)
- valarray< complex< double > > NormOfPointsInVector (vector< Point > a)
- double TriangleArea (Point a, Point b, Point c)

5.9.1 Function Documentation

5.9.1.1 E_plane_back()

```
complex< double > E_plane_back (
              Point p,
              double l,
              double a )
```

5.9.1.2 Green() [1/2]

```
complex< double > Green (
              Point p0,
              Point p,
              double wavelength )
```

5.9.1.3 Green() [2/2]

```
valarray< complex< double > > Green (
              Point p0,
              vector< Point > p,
              double wavelength )
```

5.9.1.4 NormOfPointsInVector()

```
valarray< complex< double > > NormOfPointsInVector (
              vector< Point > a )
```

5.9.1.5 toComplexArray()

```
valarray< complex< double > > toComplexArray (
              valarray< double > a )
```

5.9.1.6 transposeArray()

```
valarray< valarray< double > > transposeArray (
              valarray< valarray< double > > a )
```

5.9.1.7 TriangleArea()

```
double TriangleArea (
              Point a,
              Point b,
              Point c )
```

5.10 special_functions.h File Reference

```
#include <cmath>
#include <complex>
#include <valarray>
#include "GoTools/utils/Point.h"
#include <gsl/gsl_complex.h>
#include <gsl/gsl_complex_math.h>
```

Macros

- #define SPECIAL_FUNCTIONS_H

Functions

- const complex< double > i_unit (0.0, 1.0)
- valarray< complex< double > > toComplexArray (valarray< double > a)
- valarray< valarray< double > > transposeArray (valarray< valarray< double > >a)
- valarray< complex< double > > NormOfPointsInVector (vector< Point > a)
- gsl_complex toGSLcomplex (complex< double > c)
- complex< double > E_plane_back (Point a, double wavelength, double theta)
- string point2str (Point a, bool addCR=false)
- complex< double > Green (Point p0, Point p, double wavelength)
- valarray< complex< double > > Green (Point p0, vector< Point > p, double wavelength)
- double TriangleArea (Point a, Point b, Point c)

5.10.1 Macro Definition Documentation

5.10.1.1 SPECIAL_FUNCTIONS_H

```
#define SPECIAL_FUNCTIONS_H
```

5.10.2 Function Documentation

5.10.2.1 E_plane_back()

```
complex< double > E_plane_back (
            Point a,
            double wavelength,
            double theta )
```

5.10.2.2 Green() [1/2]

```
complex< double > Green (
            Point p0,
            Point p,
            double wavelength )
```

5.10.2.3 Green() [2/2]

```
valarray< complex< double > > Green (
              Point p0,
              vector< Point > p,
              double wavelength )
```

5.10.2.4 i_unit()

```
const complex< double > i_unit (
              0.  0,
              1.  0 )
```

5.10.2.5 NormOfPointsInVector()

```
valarray< complex< double > > NormOfPointsInVector (
              vector< Point > a )
```

5.10.2.6 point2str()

```
string point2str (
              Point a,
              bool addCR = false )
```

5.10.2.7 toComplexArray()

```
valarray< complex< double > > toComplexArray (
              valarray< double > a )
```

5.10.2.8 toGSLcomplex()

```
gsl_complex toGSLcomplex (
              complex< double > c )
```

5.10.2.9 transposeArray()

```
valarray< valarray< double > > transposeArray (
              valarray< valarray< double > > a )
```

5.10.2.10 TriangleArea()

```
double TriangleArea (
              Point a,
              Point b,
              Point c )
```

5.11 special_functions.h

Go to the documentation of this file.

```
00001 #pragma once
00002 #ifndef SPECIAL_FUNCTIONS_H
00003 #define SPECIAL_FUNCTIONS_H
00004
00005 #include <cmath>
00006 #include <complex>
00007 #include <valarray>
00008 #include "GoTools/utils/Point.h"
00009 #include <gsl/gsl_complex.h>
00010 #include <gsl/gsl_complex_math.h>
00011
00012 using namespace std;
00013 using namespace Go;
00014 const complex<double> i_unit(0.0, 1.0);
00015 valarray<complex<double> > toComplexArray(valarray<double> a);
00016 valarray<valarray<double> > transposeArray(valarray<valarray<double> >a);
00017 valarray<complex<double> > NormOfPointsInVector(vector<Point> a);
00018 gsl_complex toGSLcomplex(complex<double> c) { gsl_complex z; GSL_SET_COMPLEX(&z, c.real(), c.imag()); return z; }
00019 complex<double> E_plane_back(Point a, double wavelength, double theta);
00020 string point2str(Point a, bool addCR = false) { ostringstream ss; string s = (addCR) ? "\n" : ""; ss « "(" «
      to_string(a[0]) « "," « to_string(a[1]) « "," « to_string(a[2]) « ")" « s; return ss.str(); }
00021 complex<double>  Green(Point p0, Point p, double wavelength);
00022 valarray<complex<double> > Green(Point p0, vector<Point> p, double wavelength);
00023 double TriangleArea(Point a, Point b, Point c);
00024 #endif
```

5.12 test_mesh.cpp File Reference

```
#include <iostream>
#include <random>
#include "analysis_NURBSCurve.h"
#include "GoTools/geometry/SplineCurve.h"
#include <utility>
#include <stdlib.h>
#include "tbb/tick_count.h"
```

Functions

- int main (int argc, char *argv[ ])

5.12.1 Function Documentation

5.12.1.1 main()

```
int main (
              int argc,
              char * argv[ ] )
```

5.13 test_opt.cpp File Reference

```
#include "opt_problems.h"
#include <pagmo/archipelago.hpp>
#include <pagmo/algorithm.hpp>
#include <pagmo/algorithms/compass_search.hpp>
#include <pagmo/algorithms/ihs.hpp>
#include <pagmo/algorithms/nlopt.hpp>
#include <pagmo/population.hpp>
#include "tbb/tick_count.h"
#include "analysis_NURBSCurve.h"
#include <complex>
```

Functions

- int main (int argc, char *argv[ ])

5.13.1 Function Documentation

5.13.1.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

5.14 test_splineCurve.cpp File Reference

```
#include <mpi.h>
#include <iostream>
#include "analysis_NURBSCurve.h"
#include <stdlib.h>
```

Functions

- void broadcastCurveData (analysis_curves &curves, int buf_size, int rank, int root, MPI_Comm communicator)
- int main (int argc, char *argv[ ])

5.14.1 Function Documentation

5.14.1.1 broadcastCurveData()

```
void broadcastCurveData (
            analysis_curves & curves,
            int buf_size,
            int rank,
            int root,
            MPI_Comm communicator )
```

5.14.1.2 main()

```
int main (
            int argc,
            char * argv[ ] )
```

Index