NAZARBAYEV UNIVERSITY

School of Engineering and Digital Sciences

Bachelor of Engineering in
Mechanical and Aerospace Engineering

Investigation of EM scattering behavior of weakly interacting nanotubes

(Final Capstone Project Report)

by

Nurkeldi Iznat, Madeniyet Bespayev, and Yerassyl Turarov

Lead Supervisor: Prof. Konstantinos Kostas

Co-Supervisor: Prof. Konstantinos Valagiannopoulos

April, 2024

Declaration

We, Nurkeldi Iznat, Madeniyet Bespayev, and Yerassyl Turarov, hereby declare that this report, entitled “Investigation of EM scattering behavior of weakly interacting 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: Nurkeldi Iznat
Date: April 28, 2024

Signature:

Name: Madeniyet Bespayev
Date: April 28, 2024

Signature:

Name: Yerassyl Turarov
Date: April 28, 2024

Abstract

The problem of electromagnetic (EM) waves scattering by circular and arbitrary-shaped nanotubes is considered in this work. An Iso-Geometric Analysis enabled Boundary Element Method (IGABEM) is developed to estimate the scattering amount in the corresponding cases. Our research studies the scattering of EM waves caused by a nanotube and aims to optimize the shape of the nanotube so that the scattering value is maximized. The problem is of particular interest when the electrical conductivity of the nanotube is very low. This is because by only changing the shape of the nanotube, its weakly interacting (almost transparent) behavior could be altered and it would be changed into a super-scatterer of EM waves solely because of its geometry. In this work, the optimal shapes that maximize scattering were found and checked, with respect to their sensitivity, for varying wavelengths, wave angles, and distances from the source (when a point source was considered). The dependence of the scattering behavior of the nanotube on the DoFs needed for geometry representation as well as the computational accuracy as a function of the number of analysis DoFs were also examined. The best result, received by shape optimization in the context of this study, indicates that an arbitrarily shaped nanotube can scatter EM waves about 85 times higher than a circular nanotube with the same electircal conductivity and the same cross-sectional area. Such superscattering nanotubes can find applications in electromagnetic interference shielding, optical nanoantennas, biomedical imaging, spectroscopy and nanoplasmonic sensing.

Contents

Abstract ii

Contents iii

1 Introduction 1

2 Methodology 4

2.1 Formulation of Electromagnetic Scattering 4

2.2 Iso-Geometric Analysis Boundary Elements Method 6

2.3 Shape optimization 7

3 Results and Discussion 13

3.1 Results 13

3.2 Applications 36

3.3 Limitations 37

4 Conclusions and Future Work 38

4.1 Conclusions 38

4.2 Future Work 39

4.3 Distribution of Tasks 39

Bibliography 40

Appendices 44

A Source Code - C++ Code 45

B Source Code MatLAB Code 95

Chapter 1

Introduction

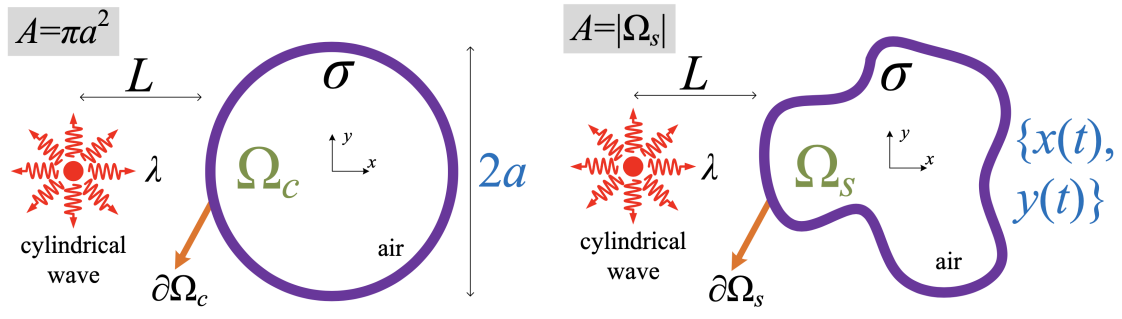
Nanotubes are cylindrical structures made out of 2D materials with a diameter in the nanometer scale. They can be obtained by rolling up a sheet of material that is a single layer of atoms of a certain material arranged in a two-dimensional lattice [1]. The most common material for nanotubes is graphene, which is extracted from graphite, and is a pure carbon structure [2]. In 1991, the carbon nanotube (CNT) was discovered and gained its popularity due to its excellent mechanical and electrochemical properties, and up to now, CNTs have a wide range of application prospects, such as manufacturing of bicycle components, creation of composite materials, material for electrodes in batteries, etc. [3]. There are also boron-nitride (BNNT) and silicon (SNT) nanotubes, among others, with similar and/or other more specialized applications. For example, BNNTs are used for neutron shielding, polymer composite reinforcement [4], whereas SNTs find their applications in biomedical imaging, bioseparation and drug delivery [5].

Scattering is a phenomenon that occurs when the electromagnetic (EM) wave is being absorbed by an object and then emitted in different directions with the same or altered wavelength. It plays a crucial role in fields such as telecommunication, biomedicine, nano technologies, etc. [6, 7]. Since 1991, when CNTs were discovered by Iijima [3], there have been many researchers who have meticulously studied the properties of CNTs and found a wide range of applications for this invention. Studies that drew our attention were addressing the electromagnetic (EM) wave scattering properties of CNTs [8, 9, 10]. Nanotubes usually have circular cross sections (see Figure 1.1(a)) since this shape is naturally occurring during the manufacturing process and admits analytical modelling [11]. In turn, we decided to study a nanotube that is not circular, but is arbitrarily shaped (see Figure 1.1(b)). Note that although it might be hard or infeasible to produce nanotubes with arbitrarily complex free-form shapes, studying the shape-dependence of scattering is still meaningful as these shapes can be potentially materialized via metamaterials comprising nanotubes. This work focuses on studying how the shape affects the way a nanotube interacts with EM waves. Namely, this research studies the scattering of the EM waves caused by the geometry of the nanotube and aims to optimize the shape of the nanotube so that the scattering value is enlarged. In our study, two cases are considered: a) EM waves being emitted from a point source, located at a certain distance from the nanotube's centroid, and b) cases of nanotubes interacting with a transverse EM plane wave.

An Iso-Geometric Analysis enabled Boundary Element Method (IGABEM) is

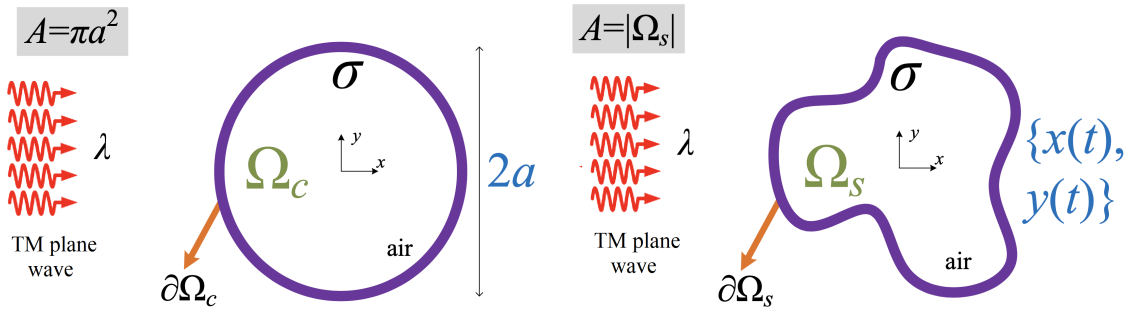
implemented in our work to estimate the relevant scattering for a TM plane wave or EM waves emanating from a point source, which is either placed at a finite distance from the nanotube or it is infinitely far away. This is because the IGA approach enables the incorporation of computer-aided models (CAD) in the analysis stage (CAE), without the need of an approximate mesh model. This is achieved by representing the unknown solution field with the same basis functions used for the geometrical representation of the design. This allows to perform analysis without relying on mesh approximations when IGABEM or IGA-enabled finite element methods (IGAFEM) are employed. Furthermore, integrating BEM with CAD results in a much faster analysis and design in terms of computational time as high-convergence rates with significantly fewer degrees of freedom can be achieved.

There are 2 main parts in this report: Methodology, which includes problem definition, tools and implementation, and Results and Discussion, where the optimized shapes and behavior of the nanotubes are discussed. The objective function used in our optimization examples considers the ratio involving the scattering from a circular nanotube (see Figure 1.1 (a) and Figure 1.2 (a)) and that from an arbitrarily shaped nanotube (see Figure 1.1 (b) and Figure 1.2 (b)). Specifically, the ratio of the scattering value of the arbitrarily shaped nanotube over the value of the circular one of the same area and surface conductivity is considered. Therefore, this ratio corresponds to the normalized scattering enhancement which is solely due to its shape. Obviously, the remaining parameters, i.e., wavelength, distance to the point source and relative angle, remain fixed for each optimization case we consider.



(a) (b)

Figure 1.1: Nanotubes excited by a circular wave emanating from a point source. (a) Circular nanotube with area A and surface conductivity σ at $L + a$ distance from the point source. (b) Arbitrarily-shaped nanotube with same A, L and σ [11].



(a) (b)

Figure 1.2: Nanotubes excited by a TM plane wave with a wavelength λ. (a) Circular nanotube with area A and surface conductivity σ. (b) Arbitrarily-shaped nanotube with same A and σ [11].

Chapter 2

Methodology

2.1 Formulation of Electromagnetic Scattering

The two-dimensional Green's function of free-space at point (r, φ) with the singular source placed at (P, Φ), both expressed in cylindrical coordinates, is given by:

$$G(r, \varphi, P, \Phi) = -\frac{i}{4} H_0^{(2)} \left(k_0 \sqrt{r^2 + P^2 - 2rP\cos(\varphi - \Phi)} \right), \tag{2.1}$$

where $k_0 = 2\pi/\lambda$ is the wavenumber, and $H_0(\cdot)^{(2)}$ is the zeroth order Hankel function of the second kind.

Our intention is to find an asymptotic expression of Equation 2.1 for $\mathbf{r} \to +\infty$ while keeping P fixed. It is well-known that, for large arguments, the Hankel function possesses the following large-argument expansion [12]:

$$H_\nu^{(2)} \sim \sqrt{\frac{2}{\pi z}} e^{-i(z - \frac{1}{2}\nu\pi - \frac{1}{4}\pi)}. \tag{2.2}$$

In the amplitude, we can assume roughly that: $\sqrt{r^2 + P^2 - 2rP\cos(\varphi - \Phi)} \cong r$ but in the phase we should employ the Newton approximation. In particular, we have:

$$(2.1) \implies G(r, \varphi, P, \Phi) \cong -\frac{i}{4} \sqrt{\frac{2i}{\pi k_0 r}} e^{-ik_0 r \sqrt{1 + \frac{P^2}{r^2} - 2\frac{P}{r}\cos(\varphi - \Phi)}} \xrightarrow{P \ll r}$$

$$G(r, \varphi, P, \Phi) \cong -\frac{i}{4} \sqrt{\frac{2i}{\pi}} \frac{e^{-ik_0 r}}{\sqrt{k_0 r}} e^{-ik_0 P\cos(\varphi - \Phi)}. \tag{2.3}$$

We know that the (sole) z component of the scattering electric field from an arbitrary-shaped nanotube is given by (constant σ):

$$E_{z,\text{scat}}(r, \varphi) = -ik_0 \sigma \eta_0 \int_{\partial\Omega} G(r, \varphi, P, \Phi) E(P, \Phi) dL \xrightarrow{(2.3)}$$

$$E_{z,\text{scat}}(r, \varphi) \cong -\frac{k_0 \sigma \eta_0 \sqrt{2i}}{4\sqrt{\pi}} \frac{e^{-ik_0 r}}{\sqrt{k_0 r}} U(\varphi), \tag{2.4}$$

where $\eta_0 = 120\pi$, which is the wave impedance into vacuum. Also, it is worthy to note that $U(\varphi) = \int_{\partial\Omega} e^{-ik_0 P\cos(\varphi - \Phi)} E(P, \Phi) dL$ is independent from the source point. Now that we have (2.3) to get the electric field into vacuum, we get the magnetic field from Faraday law:

$$H_\varphi(r, \varphi) = \frac{1}{ik_0 \eta_0} \frac{\partial E_z(r, \phi)}{\partial r} \xrightarrow{(2.4)}$$

$$H_\varphi(r,\varphi) = \frac{1}{ik_0\eta_0}\left(-\frac{k_0\sigma\eta_0\sqrt{2i}}{4\sqrt{\pi}}\right)U(\varphi)\left[-ik_0\frac{e^{-ik_0r}}{\sqrt{k_0r}} - \frac{k_0e^{-ik_0r}}{2(k_0r)^{3/2}}\right] \xrightarrow{r \text{ large}}$$

$$H_\varphi(r,\varphi) = \frac{1}{\eta_0}\frac{k_0\sigma\eta_0\sqrt{2i}}{4\sqrt{pi}}\frac{e^{-ik_0r}}{\sqrt{k_0r}}U(\varphi) \tag{2.5}$$

From Poynting theorem, we have for the power carried by an electromagnetic wave passing through a surface (S), which we take as an infinite circle, since we have far-field expressions:

$$P_{\text{scat}} = -\frac{1}{2}r\int_0^{2\pi}\text{Re}[E_z(r,\varphi)H_\varphi^*(r,\varphi)]\,d\varphi \implies P_{\text{scat}} = \frac{1}{2}\frac{1}{\eta_0}\frac{k_0^2|\sigma\eta_0|^2}{8\pi}r\int_0^{2\pi}\frac{|U(\varphi)|^2}{k_0r}\,d\varphi \implies$$

$$P_{\text{scat}} = \frac{1}{2}\frac{1}{\eta_0}\frac{k_0|\sigma\eta_0|^2}{8\pi}\frac{r}{k_0r}\int_0^{2\pi}|U(\varphi)|^2\,d\varphi \implies P_{\text{scat}} = \frac{k_0}{16\pi\eta_0}|\sigma\eta_0|^2\int_0^{2\pi}|U(\varphi)|^2\,d\varphi, \tag{2.6}$$

where P_{scat} is the scattered power by the nanotube per unit length of $\mathbf{z}$ axis.

From Equation 2.6, we can see that P_{scat} is proportional to the square of the relative complex conductivity $|\sigma\eta_0|^2$. This is intuitive and natural since the scattering is an outcome of the textural contrast that the nanotube possesses. On top of that, from the equation, we can see that the scattering is also proportional to the squared voltage $|U(\varphi)|^2$:

$$U(\varphi) = \int_{\partial\Omega} e^{-ik_0P\cos(\varphi-\Phi)}E(P,\Phi)\,d\ell \tag{2.7}$$

Here $E(P,\Phi)$, with $(P,\Phi) \in \partial\Omega$, is the electric field on the boundary of the metasurface, which can be expressed in the following way:

$$E(P,\Phi) = E_{\text{back}}(P,\Phi) + E_{\text{scat}}(P,\Phi) \tag{2.8}$$

From (2.8), it is obvious that in order to determine the electric field on the boundary, we need to compute the background electric field, E_{back}, and the scattered electric field from the nanotube, E_{scat}.

The background electric field by a point source, in the absence of the tube, is expressed as:

$$E_{\text{back}}(x,y) = H_0\left(k_0\sqrt{(x+a+L)^2+y^2}\right), \tag{2.9}$$

where H_0 is the zeroth order Hankel function of the second kind. On the other hand, the background electric field by a planar wave is expressed as:

$$E_{\text{back}}(x,y) = \exp(-\text{i}k_0x). \tag{2.10}$$

The scattered electric field at an arbitrary point, $\mathbf{P}$, is as follows [11]:

$$E_{\text{scat}}(\mathbf{P}) = -ik_0\sigma\eta_0\oint_{\partial\Omega}G(\mathbf{P},\mathbf{P}')E(\mathbf{P}')\,d\ell', \tag{2.11}$$

where, in free space, $\mathbf{P}' \in \partial\Omega$ and $G(\mathbf{P},\mathbf{P}') = -\frac{i}{4}H_0(\|\mathbf{P}-\mathbf{P}'\|)$. Just like in (2.10), H_0 is the zeroth order Hankel function of the second kind.

Now, if we plug (2.10) and (2.11) into (2.8), we can compute the electric field on the boundary. The values of the electric field on the boundary, in turn, allows us to calculate the voltage, see (2.7). And, finally, plugging the voltage into (2.6) gives us the value we are trying to maximize, which is the scattered powered by the nanotube per unit length.

2.2 Iso-Geometric Analysis Boundary Elements Method

2.2.1 Mathematical Representaion of Nanotube Shape

The arbitrarily shaped curve $\partial\Omega$ is represented with a regular parametric NURBS curve $r(t), t \in I$. If we assume that $\mathcal{I} = \{t_0, t_1, \dots, t_{n+k}\}$ corresponding to the so-called knot vector, a NURBS curve is expressed in the following way:

$$r(t) \coloneqq \sum_{t=0}^{n} \mathbf{b}_i N_{i,k}(t), \quad t \in I = [t_{k-1}, t_{n-1}] \tag{2.12}$$

where $n+1$ is the overall number of control points and k is its so-called order, which equals to the polynomial degree plus 1 ($k = d + 1$).

The control points for a NURBS curve in (2.12) are in the following form: $\mathbf{b}_i = (x_i \cdot w_i, y_i \cdot w_i, w_i)^T, \quad i = 0, \dots, n$ with $\overline{\mathbf{b}}_i = (\frac{x_i}{w_i}, \frac{y_i}{w_i})^T$ being the corresponding NURBS curve control point in the Euclidean space. Furthermore, the B-spline basis functions, denoted as $\{N_{i,k}(t)\}_{i=0}^{n}$, are recursively defined over a knotvector $\mathcal{I}$ as follows:

$$N_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t)$$

where the initial functions for $k = 1$ are:

$$N_{i,1} = \begin{cases} 1 & \text{if } t_i \leq t < t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Now, if we consider the projection of $r(t)$ onto the $\mathbb{E}^2$ space, we get:

$$\overline{r}(t) = \sum_{i=0}^{n} \overline{\mathbf{b}}_i \frac{w_i N_{i,k}(t)}{\sum_{j=0}^{n} w_j N_{j,k}(t)} = \sum_{i=0}^{n} \overline{\mathbf{b}}_i R_{i,k}(t)$$

where $\{R_{i,k}(t)\}_{i=0}^{n}$ represent the rational basis functions for NURBS curves.

2.2.2 Mathematical Representation of Electric Field

As the Iso-Geometric Analysis approach couples the design with analysis, we need to express the unknown, $E(P, \Phi)$, in (2.7) with the same basis functions we used for the geometry, which is $R_{i,k}(t)$. To do that, we express the approximation of the electric field values, $\tilde{E}(P, \Phi)$, as follows:

$$\tilde{E}(P, \Phi) = \tilde{E}[x(t), y(t)] \coloneqq \sum_{i=0}^{n} e_i R_{i,k}(t), \quad t \in I.$$

A sequence of nested finite-dimensional spline spaces $\mathcal{S}_k(\mathcal{I}^{(0)}) \subset \ldots \subset \mathcal{S}_k(\mathcal{I}^{(m-1)}) \subset \mathcal{S}_k(\mathcal{I}^{(m)})$ with $m \in \mathbb{N}^*$, denoting the number of knots inserted in $\mathcal{I} = \mathcal{I}^{(0)}$, can be used to create a set of richer spline spaces which allow the more accurate approximation of $E(P, \Phi)$, which can be written as:

$$\tilde{E}[x(t), y(t)] \coloneqq \sum_{i=0}^{n+m} e_i R_{i,k}^{(m)}(t), \tag{2.13}$$

where $\{R_{i,k}^{(m)}(t)\}_{i=0}^{n+m}$ is the basis of spline space $\mathcal{S}_k(\mathcal{I}^{(m)})$.

Moreover, if we use a collocation method that employs the Greville abscissae for the knot vector $\mathcal{I}^{(m)}$, we get a better convergence [11, 13]. Greville abscissae are knot averages such that $\hat{t}_j = \frac{1}{k-1}(t_{i+1} + t_{i+2} + \ldots + t_{i+k-1})$ [11].

Finally, we get the following linear system:

$$E_{\text{back}}(\hat{t}_j) - \sum_{i=0}^{n+m} e_i R_{i,k}^{(m)}(\hat{t}_j) = ik_0\sigma\eta_0 \sum_{i=0}^{n+m} e_i \oint_I G(\hat{t}_j, \tau) R_{i,k}^{(m)}(\tau) \|\bar{\mathbf{r}}'(\tau)\| \, d\tau, \tag{2.14}$$

where $j = 0, 1, \ldots, n+m$. Solving this linear system yields us $\{e_i\}_{i=0}^{n+m}$, from which we can get the electric field on the boundary of the metasurface using (2.13). Now, since the boundary of the metasurface is parametrically represented as $\partial\Omega_s(t) = \{x(t), y(t)\}$ for $t \in I \subset \mathbb{R}$, the voltage in (2.7) must be represented in the cartesian coordinate system:

$$U(\varphi) = \int_0^1 e^{-ik_0\sqrt{[x(t)]^2+[y(t)]^2}\cos\left[\varphi - \arctan\left(\frac{y(t)}{x(t)}\right)\right]} E[x(t), y(t)] \sqrt{[x'(t)]^2 + [y'(t)]^2} \, dt. \tag{2.15}$$

This expression, then, can be plugged into (2.6), which becomes easy to numerically integrate to get the scattered power for the nanotube.

2.3 Shape optimization

As a reference, a perfectly circular nanotube with a radius, *r*, and an area, $A = |\Omega_c| = \pi r^2$ is considered (see Figure 1.1(a)). Next, we consider another nanotube defined by an arbitrary closed smooth curve, S, the centroid of which is at the origin (see Figure 1.1(b)) and seek to solve the following constraint optimization problem:

$$\begin{array}{ll} \text{Maximization:} & P_{scat,S}/P_{scat,C}, \\ \text{Area constraint:} & |A(\Omega_C) - A(\Omega_S)| < \epsilon, \\ \text{Distance from source:} & L + a, \\ \text{(when a point source was considered)} & \end{array} \tag{2.16}$$

where ϵ is an arbitrarily small positive number and where $P_{scat,S}$ and $P_{scat,C}$ can be computed using (2.6) for the arbitrarily-shaped and circular nanotube, respectively.

We focus on optimizing the shape of the arbitrarily shaped nanotube so that its scattering property is much larger compared to the scattering of the circular reference nanotube.

The arbitrarily shaped nanotube will have about the same area (namely, $A \pm \epsilon$) as the circular one. Both the circular and the arbitrarily shaped nanotubes' centroids are located at distance $L + a$ from the point source (if applicable). Given that the nanotubes have the same surface complex conductivity σ, the only difference between the two nanotubes is their shapes. Thus, we can compare the scattering from the arbitrarily shaped nanotube $P_{scat,S}$ to that of the circular nanotube $P_{scat,C}$ and see how we can improve scattering by only modifying the shape.

Geometric Parametric Models

In IGA shape optimization, using NURBS control points or weights as direct design variables to parameterize a shape can be really cumbersome and may require complicated constraints, especially with shapes that have high numbers of degrees of freedom [11]. Failing to address these issues may result in self-intersecting or other invalid shapes [11]. Thus, we used appropriate geometric parametric models that produce valid NURBS curves without any complicated mathematical constraints. In this work so far, we have used three families of parametric models:

1. Equal-sector-based parametric model (see [11])
   Given a $2 \times n$ matrix of parameters V such that $\forall\, V_{i,j} \in [0; 1]$, the location of each auxiliary point can be expressed as [11]:

   $$r_i = V_{1,i}(r_{max} - \varepsilon) + \varepsilon, \quad \varphi_i = \frac{2\pi(V_{2,i} + i - 1)}{n},$$

   where n denotes the number of sectors and ultimately determines the DoFs of the NURBS curve, r_{max} is the maximum allowable radius, and ϵ is an arbitrarily small positive number to avoid degenerate curves with zero area. The location of the auxiliary points can also be written in homogeneous coordinates as follows [11]:

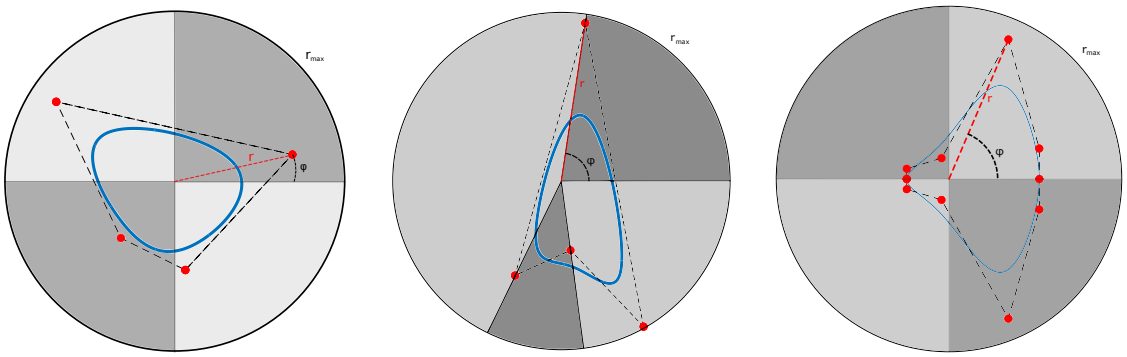
   $$\mathbf{q}_i = [r_i \cos(\varphi_i),\ r_i \sin(\varphi_i),\ 1]^T$$

2. Angle-percentage-based parametric model
   With the same $2 \times n$ matrix V, the location of each auxiliary point for this parametric model is:

   $$r_i = V_{1,i}(r_{max} - \varepsilon) + \varepsilon, \quad \begin{cases} \varphi_1 = 2\pi V_{2,1} \\ \varphi_i = V_{2,i}(2\pi - \varphi_{i-1}) + \varphi_{i-1}, \quad \text{for } i = 2, 3, \dots, n \end{cases}$$

   $$\mathbf{q}_i = [r_i \cos(\varphi_i),\ r_i \sin(\varphi_i),\ 1]^T$$

3. Symmetric parametric model
   This model is almost the same as the first one, which is the equal-sector-based parametric model. However, in the symmetric parametric model, the control points



(a) (b) (c)

Figure 2.1: The parametric models implemented in the study. (a) Equal-sector-based parametric model. (b) Angle-percentage-based parametric model. (c) Symmetric parametric model.

are only allowed to be placed in the first 180 degrees or upper half of the imaginary circle with the radius r_{max}. After that, the same control points are mirrored with respect to the x-axis and geometry is created.

The main difference of the three families is that in the first case, the circular sectors are of the same size, whereas in the second case, they depend on ϕ_i. While the third parametric also creates equally sized circular sectors, but only for the initial 180 degrees. After that, symmetrical sectors are created with respect to the x-axis. In the end, a symmetrical shape is obtained. The equal-sector-based parametric model ensures that the geometry created is enclosed and without self-intersections. The angle-percentage-based parametric model grants more freedom to the nanotube's geometry, which produces more variety of shapes with low DoFs. However, this model may lead to self-intersections. The symmetric parametric model is mainly designed to reduce the computational time required for the optimization process, allowing more DoFs to be used for the nanotube representation compared to the other two models. In Figure 2.1, the examples for each parametric model's outcome can be seen.

Shape Optimization using MATLAB

In order to optimize the shape of a nanotube in *MATLAB*, we used the following optimizers from the Optimization Toolbox and Global Optimization Toolbox:

- A gradient-based search, ***fmincon()***
- A direct, derivative-free search, ***patternsearch()***
- Genetic Algorithm, ***ga()***

fmincon() starts from an initial estimate and then seeks a constrained minimum of a scalar function of many variables. This is commonly known as nonlinear programming

or restricted nonlinear optimization [14]. One of the main inputs we give to *fmincon()* are the parameters (§2.2.1) through which the geometry of the nanotube is constructed. Since this function finds the minima, equation (2.6) multiplied by *-1* was used as the objective function in *fmincon()* in order to find the maximum scattering. By changing the parameters, *fmincon()* is seeking for the parameters that will create a shape with higher scattering compared to the circular nanotube. Additional parameters given to *fmincon()* were *lower (lb)* and *upper (ub)* boundaries (that correspond to the side constraints of the values of the parameters that define the auxiliary points in the parametric model) [14], that are equal to 0 and 1, respectively, and *areaconstaint*, a nonlinear constraint that restricts the search to shapes that have the same cross-sectional area with the reference circular tube.

patternsearch() is similar to *fmincon()* and tries to minimize the value of the function given as input. All the inputs are the same for both MATLAB functions. However, *patternsearch()* is better in terms of its ability to reach a global optimum, since *patternsearch()*, contrary to *fmincon()*, is not a gradient-based approach, but a heuristic method that has a generally good expected behavior (but with no rigorous mathematical guarantees in the general case) [15]. The chances for the output of the *patternsearch()* to be a global solution are much higher compared to *fmincon()*, in any other cases, the solution is local. That is the reason we were using it more frequently in our study.

The last optimization function is *ga()*, the genetic algorithm. Unlike two previous optimization functions, genetic algorithm does not require an initial starting design vector, while *patternsearch()* and *fmincon()* are highly dependent on these initial parameters. GA creates a random initial population which evolves towards the optimum solution via a large number of generations created via the selection, crossover and mutation mechanisms [16]. Selection process aims to multiply the number of population that have greater fitness (parameters that lead to better solutions of the input problem, see Equation 2.6) compared to others. Each new generation can include mutated individuals with random changes, that may also lead to an increase in fitness [16]. The process terminates when the population converges to a single individual (i.e., the sought for design) or the number of maximum iterations is reached. The results from the genetic algorithm were used in order to find good starting points (initial parameters) for *patternsearch()* and *fmincon()* so that the chances of finding global solution are maximized.

Transition to C++

After processing first results via MATLAB, several drawbacks of the platform were drawn out: large running times, lack of a variety of optimization tools, low control over the execution of the program, and difficulties in scaling up by code parallelization on several CPU nodes to achieve lower computational time. Therefore, in order to increase code

execution efficiency, it was decided to transfer MATLAB code to the C++ environment while parallelizing its execution at the same time.

As C++ libraries are distinctively more efficient than what MATLAB can offer, new libraries were chosen for the same working goals:

- CMake - for systematization of code compilation and organizing used libraries [17];
- GNU Scientific Library (GSL) - numerical library for C and C++ for providing wide range of mathematical routines, mainly related to linear algebra and numerical quadrature [18];
- IGES (GoTools) - IGES files reading and writing utility [19];
- SISL (GoTools) - The SINTEF Spline Library for the modeling and interrogation of NURBS [19];
- oneTBB by Intel - allows multi-threaded applications with a shared memory model (improving execution speed for multi-core CPUs) [20];
- OpenMPI - parallel programming supporting both multi-threaded and multi-processor systems [21];
- PAGMO by European Space Agency - scientific library for massively parallel optimization [22].

C++ Optimizers

In C++ implementation, optimizers from the Parallel Global Multi-objective Optimizer (PAGMO) library [23] that was developed by European Space Agency were utilized:

- A direct search method used for optimization, ***compass search (CS)***
- A local derivative-free optimization, ***constrained optimization by linear approximations (COBYLA)***
- genetic algorithm's alternative, ***improved harmony search (IHS)***

Compass Search was built on a concept of incremental adjustment of parameters until improvements stopped being achieved. The algorithm starts with an initial step size for parameter's adjustment and after each iteration the step size is decreased by a half, or a similar updating rule depending on its implementation [24]. This optimizer ensures slow but trustworthy convergence to the optimum of the problem being solved.

COBYLA is a local optimization algorithm that deals mainly with constrained problems. This optimizer is based on linear approximations of the objective and does not require derivatives like gradient based methods [25]. It uses linear constraints to direct the search

toward regions that are highly likely to contain a solution. The algorithm has a dynamic feature to adapt the step size during each iteration.

IHS is a metaheuristic algorithm which was inspired by the improvisation and musical instrument's tuning process done by musicians. At the beginning of the optimization process there is an initial population of variables that create generations or values that evolve to the global optimum [26]. *IHS* can control the probability of choosing variables and the rate of mutation per iteration. It is used for various types of engineering problems and can handle both constrained and unconstrained problems.

Finally, approximately 2500+ lines of code were written to fully substitute MATLAB from the optimization routine, and after the first results were drawn out, all of the various advantages of the C++ language were proven.

Chapter 3

Results and Discussion

3.1 Results

3.1.1 Verification of the Numerical Solution

Before doing any shape optimization, it is crucial that the use of the IGABEM procedure is justified. This can be done by exploiting the fact that the respective boundary value problem possesses an analytical solution for a circular nanotube; see [27]. Namely, it is done by evaluating the deviation of the numerical solution from the analytical solution for different levels of refinements, via knot insertion, of a circle. By doing so, the accuracy and the convergence rate of the approach can be shown in order to confirm its superiority over ordinary boundary element methods.

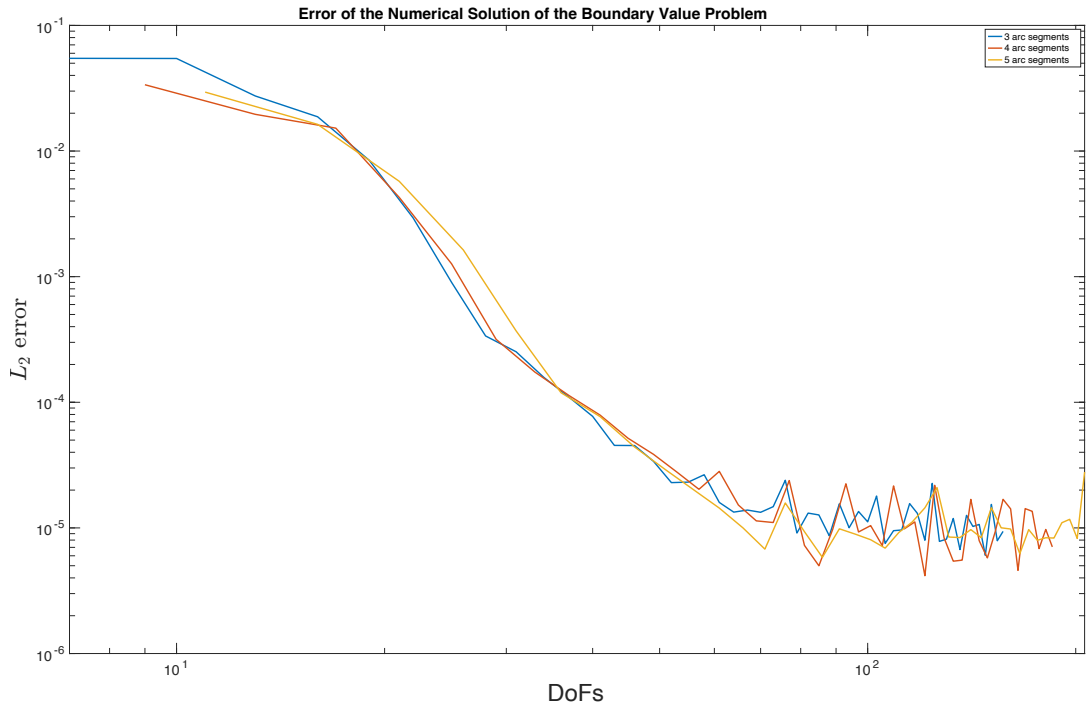


Figure 3.1: Error of the numerical solution against the analytical solution of the boundary value problem

.

As can be seen in Figure 3.1, the error drops exponentially for all the three cubic circles with different arc segments. More importantly, the figure shows that the convergence of the solution that is obtained using the IGABEM approach is much faster than the solution convergence of the ordinary boundary element methods; see the comparisons in

[28, 29, 30, 31]. This result not only verifies the numerical solution obtained from the IGABEM procedure but also justifies the use of the approach over the other methods due to its fast solution convergence rate.

3.1.2 Preliminary Shape Optimization

Using random shapes from the 1st parametric model described in §2.3, as initial designs for the gradient-based algorithm (fmincon), locally optimal designs were computed for a number of auxiliary points (Degrees of freedom) ranging from 5 to 10; (see Figure 3.2 and 3.3 below). Calculations were performed using the following values for the remaining parameters:

- Surface conductivity, $\sigma\eta_0 = 0.12\pi(1+i)$;
- Wavelength, $\lambda = 1$ unit;
- Area of curve, $A = 2\lambda^2$;
- Distance from point source, $L = 10\lambda$;
- Frequency, $k_0 = 2\pi/\lambda$.

Theoretically, as the number of degrees of freedom of the shape is increased, the value of the objective function goes up (or at least stays the same if the global optimum is achieved). However, as can be seen from Figure 3.2, we do not necessarily see an increase in the scattering as we increase the number of degrees of freedom of the shapes generated. This can be explained by the simple fact that the optimized shapes are the local optima of our search space and not the global optima, which is the price we pay for using local optimizers like "fmincon" and "patternsearch."

While reviewing the parametric model for the optimized nanotube shape with 10 control points, most of the parameters for the angle and the distance values were very close to the boundaries of the constraints of the parametric model, namely 0.99 and 0.01, which indicated that a new parametric model with more freedom is needed to be able to generate more diverse shapes for the nanotube optimization.

The “patternsearch” optimization algorithm was also used to verify results from "fmincon". Initial parameters for the "patternsearch" algorithm were similar to those for the “fmincon” algorithm. With randomly parameterized initial curves and degrees of freedom from 5 to 10, "patternsearch" resulted in shapes structures and scattering values, i.e., not significantly better to the ones produced by circular nanotubes.

At this point, the second parametric model, angle-percentage-based parametric model (see 2.3), was employed, which allowed for the sectors’ angles to vary. Resulting geometries began to be more flexible, however only with low DoFs. With high DoFs, the curves were not valid and the close distance of the control points and their increased

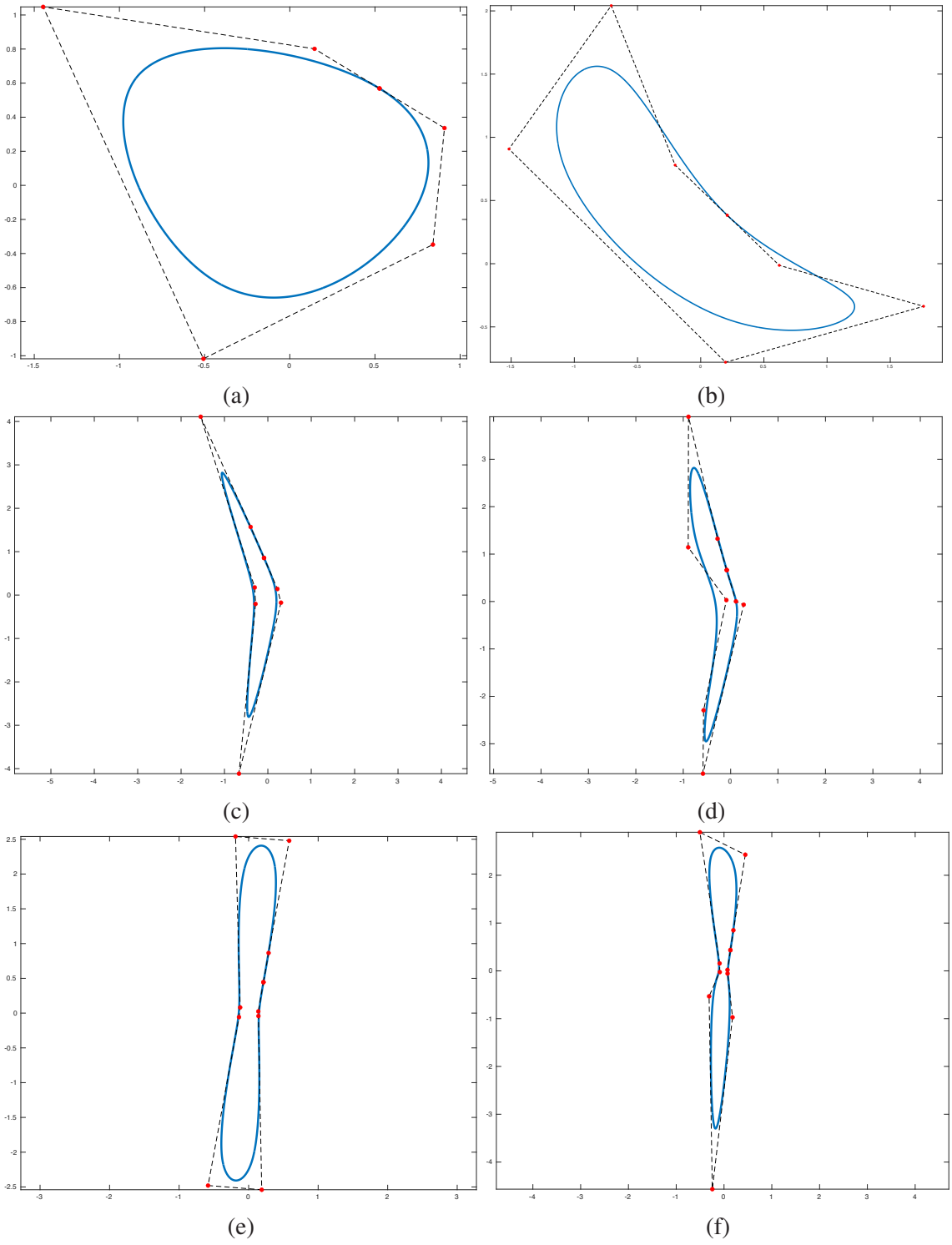
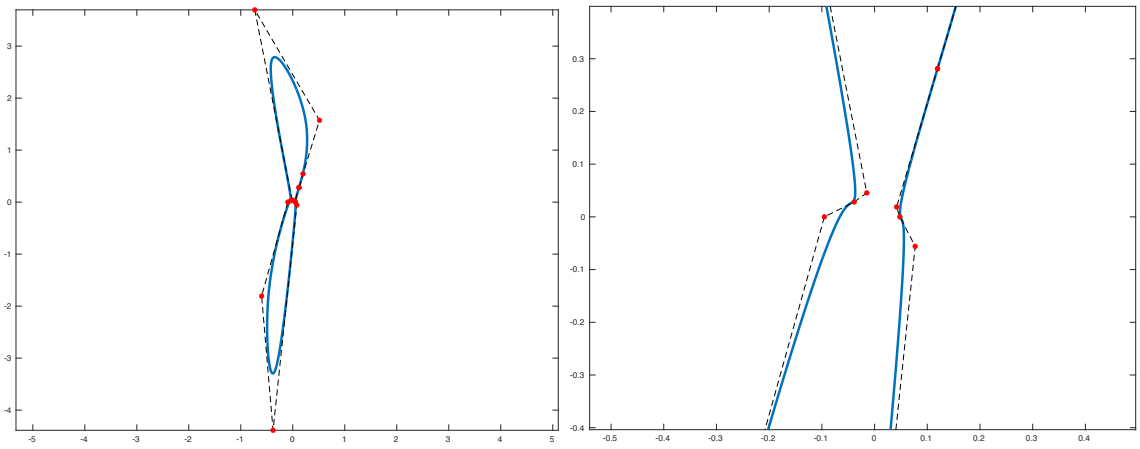


Figure 3.2: Optimized nanotube shapes with DoFs from 4 to 9 (a) 4 DoFs, scattering ratio = 1.1157 (b) 5 DoFs, scattering ratio = 1.3049 (c) 6 DoFs, scattering ratio = 2.4585 (d) 7 DoFs, scattering ratio = 2.4656 (e) 8 DoFs, scattering ratio = 2.0316 (f) 9 DoFs, scattering ratio = 2.1432

flexibility resulted in self-intersections. This eliminated the possibility of using the genetic algorithm with this specific parametric model since, unlike the equal-sector-based parametric model, the angle-percentage-based parametric model did not guarantee a



(a) (b)

Figure 3.3: Optimized nanotube shape with 10 DoFs. (a) Nanotube shape with scattering ratio = 2.4476 (b) Closer look to a cluttered region

valid (non self-intersecting) NURBS curve. Thus, even for "fmincon" and "patternsearch," the initial parameters under this parametric model could not be random. That is why, knowing that these two algorithms do not change the initial parameters as dramatically as say the genetic algorithm would, the initial parameters that result in valid geometries were manually entered. Furthermore, to address the same issue, the optimized parameters with lower DoFs were used as starting points for optimization with higher DoFs.

Analysis of angle of incidence of point source to the geometry was performed, and for the circular nanotube, the scattering value obviously remains constant, but this may not be true for an arbitrarily shaped nanotube (see Figure 3.4 below). Resulting graphs showed that, in most cases, fmincon missed these more optimal rotated shapes with better scattering values. Interestingly, presence of approximate symmetry within the scattering vs incidence angle graph was found near maximum scattering value. Therefore, such symmetry may define maxima for given free-form geometries and underlines the necessity of symmetry presence within geometry for the maximization.

Since there is no mathematical basis for both “fmincon” and “patternsearch” to be exactly or near the global maxima, with so many variables and geometry shapes it is nearly impossible to identify the global maximum with these algorithms. Necessity of a new algorithm for finding global maxima points (or good approximations) was clear, and the genetic algorithm was applied with the safer equal-sector-based parametric model. High DoF values have been set, with as many as 20 control points. For computational reasons, the convergence tolerance for the genetic algorithm was set to a relatively large number as it was our goal to be at least in the vicinity of the global maximum, at which point we could use computationally faster algorithms, “fmincon” and "patternsearch". Resultant optimized geometries are shown in Figure 3.5 and 3.4. It is clear that with such method of consecutive usage of both genetic algorithm and "fmincon"/"patternsearch", we obtain better results than using just the genetic algorithm on its own. This is especially true if

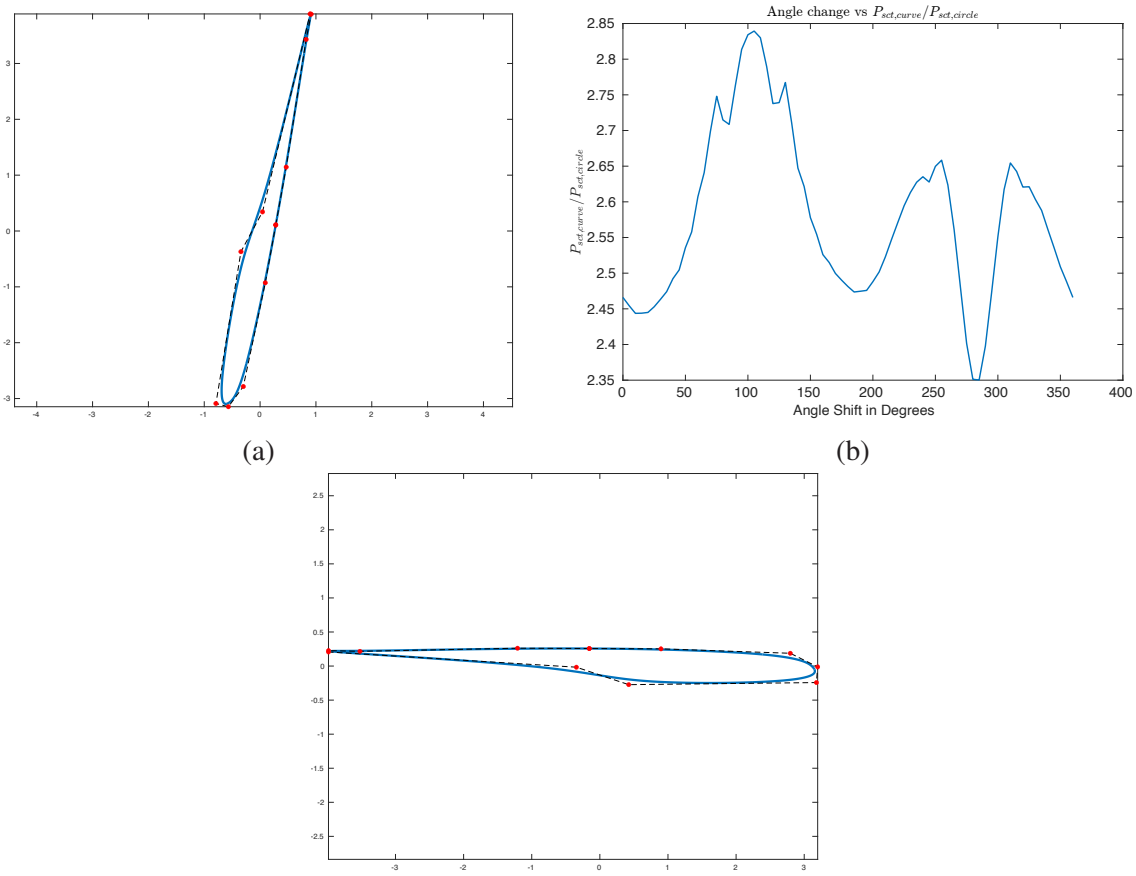
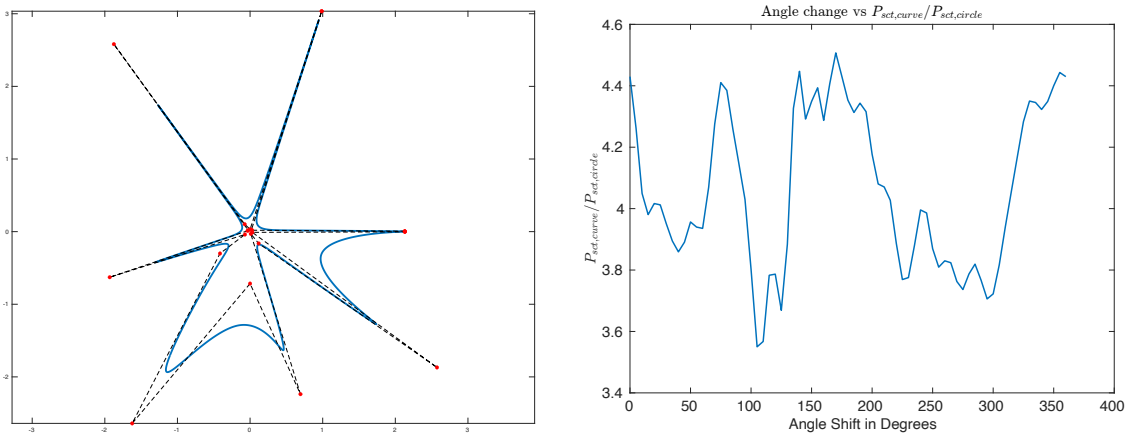


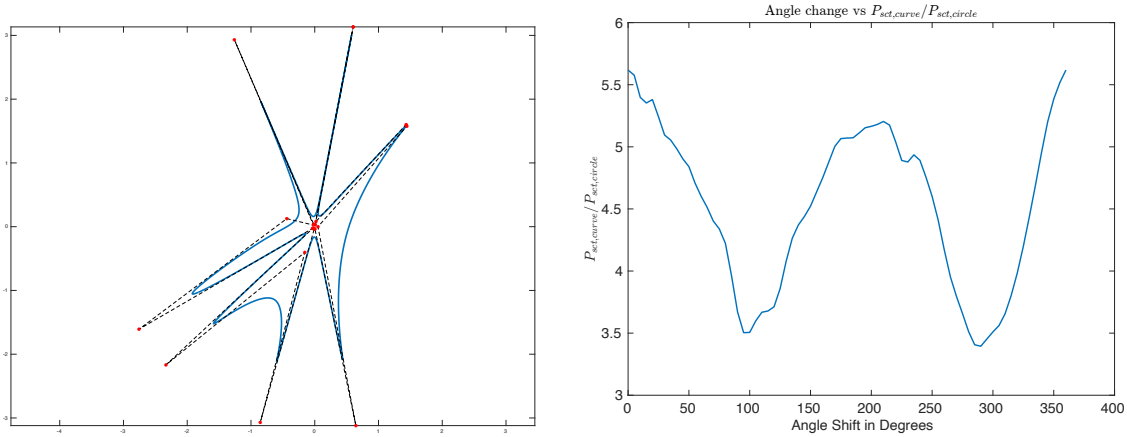
Figure 3.4: Optimized nanotube shape (10 DoFs), Angle-percentage-based parametric model. (a) Initial optimized Nanotube shape with the scattering ratio 2.47 (b) Scattering ratio values change over angle of source incidence to the shape (c) Rotated Nanotube shape with the shape's maximum scattering ratio = 2.84

the optimized parameters from the genetic algorithm are converted to the 2nd parametric model, angle-percentage-based parametric model, and then, used as the starting point for the "patternsearch" or "fmincon" algorithms to be optimized further, which is how the results in Figure 3.6, 3.13 and 3.14 were obtained.



(a) (b)

Figure 3.5: Optimized nanotube shape with GA method (20 DoFs), Equal-sector-based parametric model. (a) Optimized Nanotube shape with the scattering ratio 4.43 (b) Scattering ratio values change over angle of source incidence to the shape



(a) (b)

Figure 3.6: Optimized nanotube shape by "fmincon" with GA (20 DoFs), Equal-sector-based parametric model. (a) Optimized Nanotube shape with the scattering ratio 5.62 (b) Scattering ratio values change over angle of source incidence to the shape

3.1.3 Convergence Study and Computational Time

In order to accurately calculate the electric field on the boundary and the scattering value during shape optimization, it is important to have an idea of what the sufficient number of DoFs should be for the analysis, which is different from the number of DoFs for representing the geometry. Even though it is correct that both the electric field on the boundary and the geometry are represented with and the same splines and basis functions, additional knots are inserted in between the control points of the geometry in order to have more collocation points on the boundary which in turn can be used to compute the electric field more accurately. Up until when the number of DoFs is so high that the numerical errors start to be more noticeable, it is certain that the higher the number of knot insertions (so called h-refinement) is, the more accurate the solution is. Thus, it is the aim of this

study to determine the approximate number of DoFs at which the scattering value starts to converge.

Since, the complexity of a shape greatly influences the number of DoFs needed for correctly computing the electric field, the study was done for three separate families of shapes with varying complexities;

1. Circles with varying arc segments that represent simple shapes,
2. Randomly generated shapes with 20 control points for representing shapes with medium complexity,
3. Optimized shapes from 3.1 that represent a family of complicated shapes.

For simple shapes, the number of DoFs needed for accurately computing the scattering value should be approximated by exploiting the analytical solution for the scattering for the circular nanotube, which can be used as ground truth and be compared to the scattering value calculated with different DoFs. The result of the study can be seen in Figure 3.7 and it can be easily concluded that 50 DoFs is more than sufficient to compute the scattering value correctly.

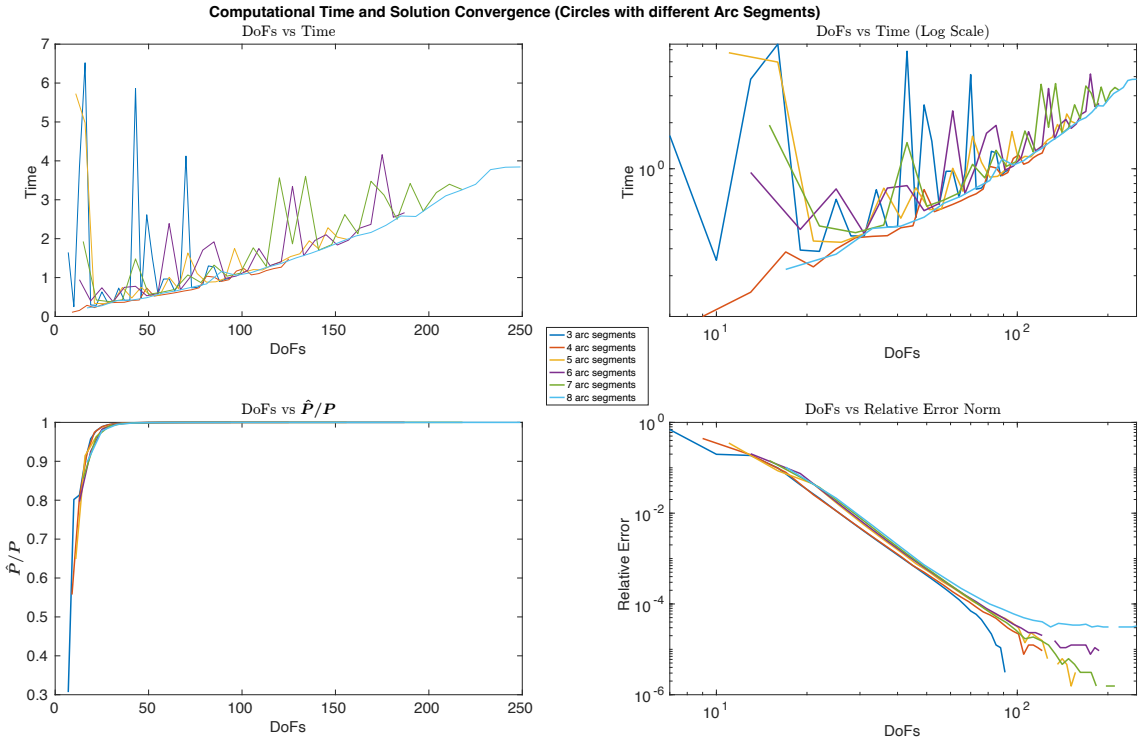


Figure 3.7: Computational Time and Solution Convergence, Circle

For the families of free-form geometries with medium and high complexities, analytical solutions obviously cannot be obtained. Thus, relative error norms were used for the analysis and the average of the scattering values obtained from the highest 10 DoFs was assumed to be the ground truth.

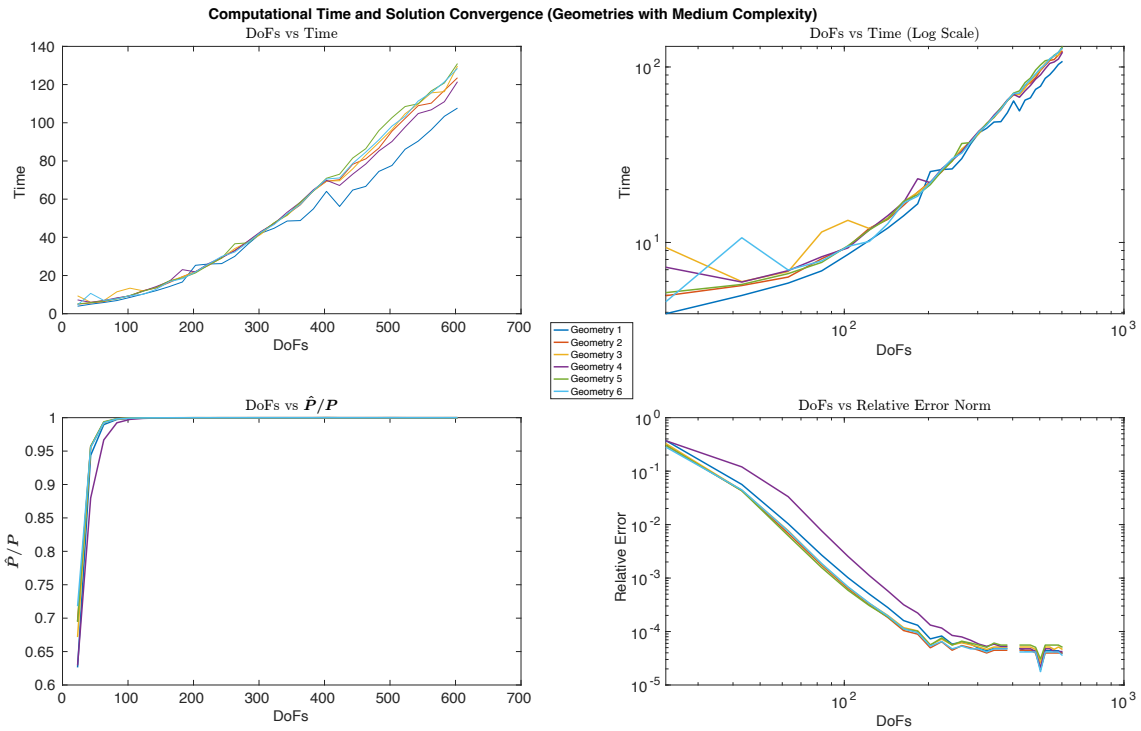


Figure 3.8: Computational Time and Solution Convergence, Geometries with Medium Complexity

When it comes to shapes with medium complexity (see Figure 3.8), at a little more than 100 DoFs the scattering value converges. It is also important to note that the rate at which the solution converges is slower and the computational time that it takes for the solution to converge is longer, which is expected.

Solution convergence for the family of complex geometries is the most crucial one as they are the optimum shapes for maximum scattering and are used in almost all of the optimization runs. From Figure 3.9, it can be seen that five out of six shapes needed about 300 DoFs in order to converge while the outlier converged at about 500 DoFs. However, the rate at which the outlier converged did not differ from the other five. In general, the convergence rate is expectedly higher than the other two families of shapes. The difference in computational time is also apparent when compared to the other two families. To be more specific, the computational time for convergence for a complex shape is about 100 times more than that for a circle and 5 to 7 times more than that for shapes with medium complexity.

The results of the study can be used in the following ways:

1. For simple shapes, with very low DoFs for the geometry (i.e. up to 12 control points) and circles, 50 DoFs for the analysis is sufficient.

2. For shapes with medium complexity (i.e. shapes with about 25 control points), 100 DoFs is acceptable and 120 DoFs is sufficient. To reduce the computational cost, shape optimization can be done with 100 DoFs for the analysis. However, after the

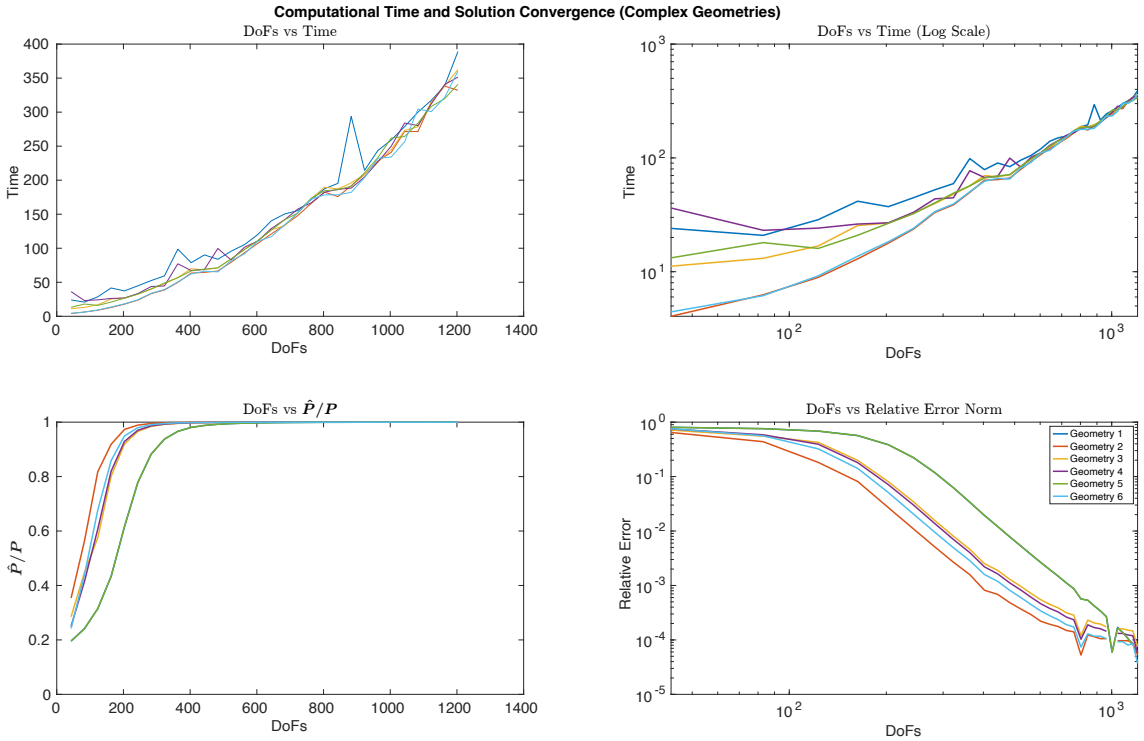


Figure 3.9: Computational Time and Solution Convergence, Geometries with High Complexity

optimized shape is obtained, it is a good idea to recalculate the scattering value for the optimized shape with about 120 DoFs to get a more accurate and final result.

3. For complex shapes (i.e shapes with 40 control points and more), 300 DoFs give accurate results in most cases. Nonetheless, one cannot be sure that it is sufficient to accurately compute the scattering value. Thus, it is a good practice to use 300 DoFs during optimization. However, it is a must to recalculate the electric field on the boundary and the scattering value with about 400 to 450 DoFs for the analysis.

3.1.4 DoFs needed for shape determination

For getting more robust results, it is crucial that the right number of DoFs are determined for both representing the geometry of the shape and for computing the scattering value. When it comes to finding the optimal shape, as we increase the number of DoFs of the shape, we should experience less gains in the value of the scattering. Thus, we should find the number of DoFs at which the change in the value of scattering is negligible. This type of geometry convergence analysis would give an idea of what the sufficient number of DoFs should be in representing the geometry when doing shape optimization. As this type of analysis requires running an optimization algorithm for geometries with multiple degrees of freedom, it is very costly in terms of computational time. Thus, it was decided that the analysis be done starting from 3 DoFs up to 30 DoFs with an increment of 3 and an additional number of DoFs of 40. To further bring down the computational time of this

study, a local optimizer, COBYLA, was used as opposed to the global optimizers such as the Genetic Algorithm or the Improved Harmony Search. This can be justified as the aim of this particular study is not to find the best possible shapes for a given number of DoFs. But, rather it is to show that a certain number of DoFs is sufficient enough to represent the shape and increasing it further would give only negligible improvements in both the complexity of the geometry and the scattering value. Indeed, the COBYLA optimizer was more than good enough for this study and the results are the following.

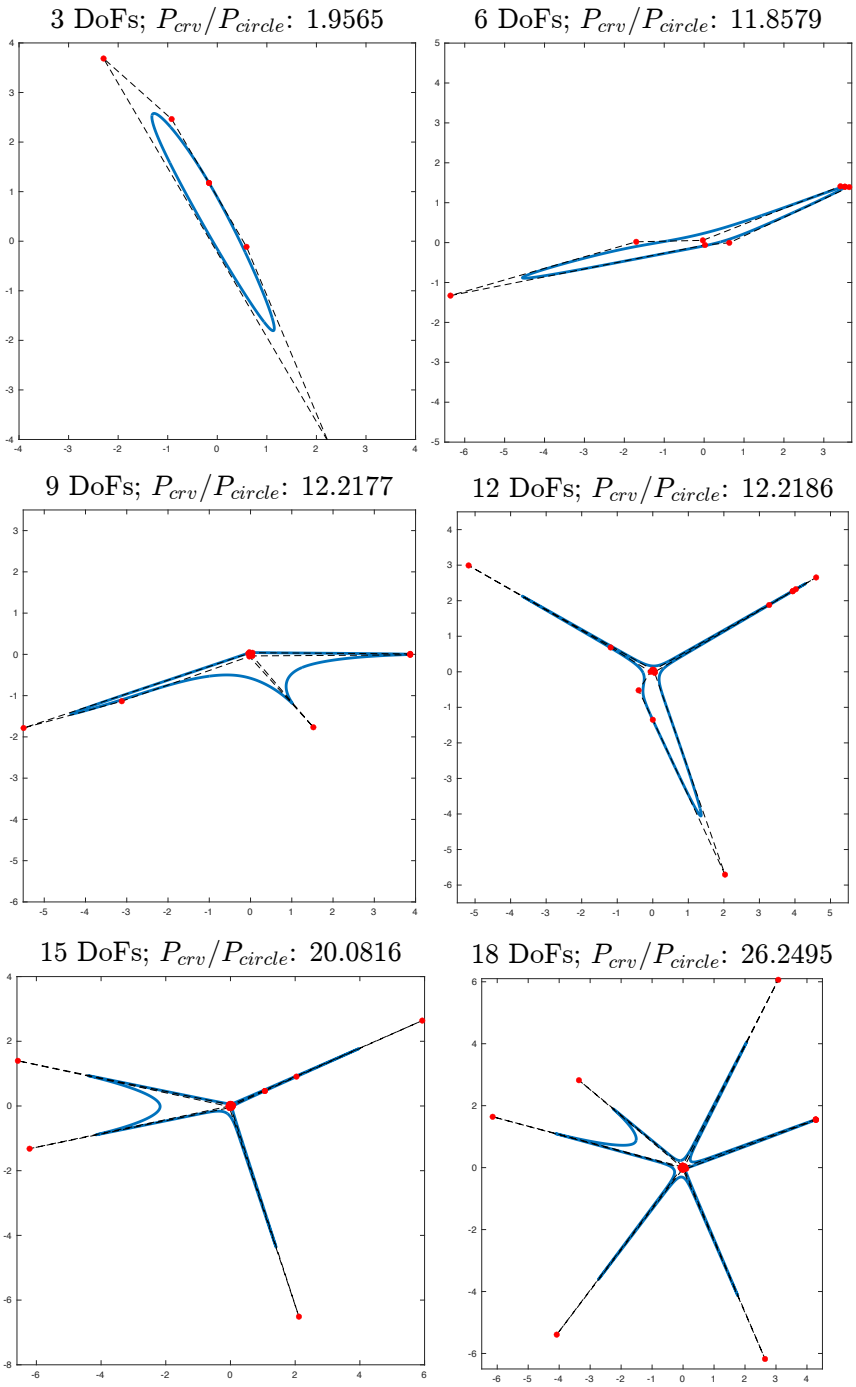


Figure 3.10: Shape Determination, 3-18 DoFs

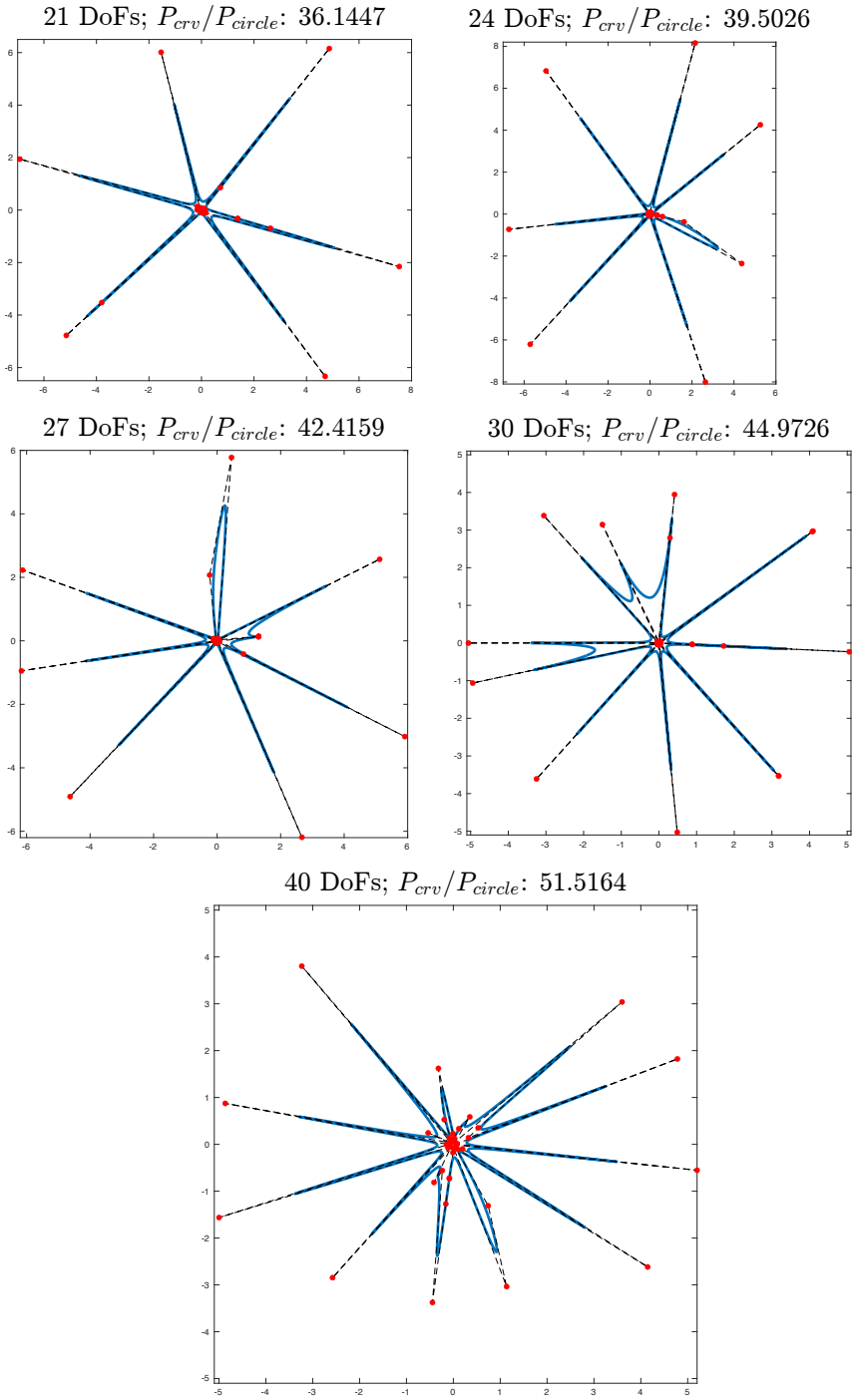


Figure 3.11: Shape Determination, 21-40 DoFs

We can see from Figure 3.10 and Figure 3.11 that as the number of DoFs increase, the shapes are getting more or less similar with repeating patterns. Namely, from 9 DoFs and onwards, the shapes have clear corners or spikes and as the number of DoFs increases the number of corners that a shape has increases as well. Even with 3 and 6 DoFs, the optimizer tries to generate shapes with corners and partially succeeds in doing that as much as the low number of DoFs allows it to. However, it is important to note that the words like 'corner' and 'spike' are used only for convenience and do not imply discontinuity in the geometry. As a matter of fact, since all the shapes in the project have a spline order of

four, the shapes are fully continuous throughout.

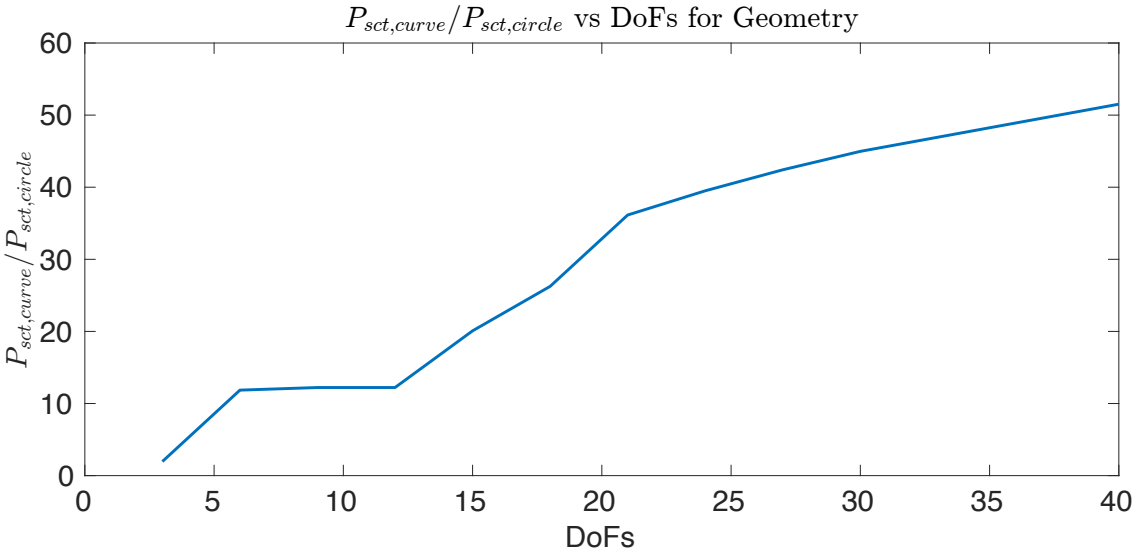


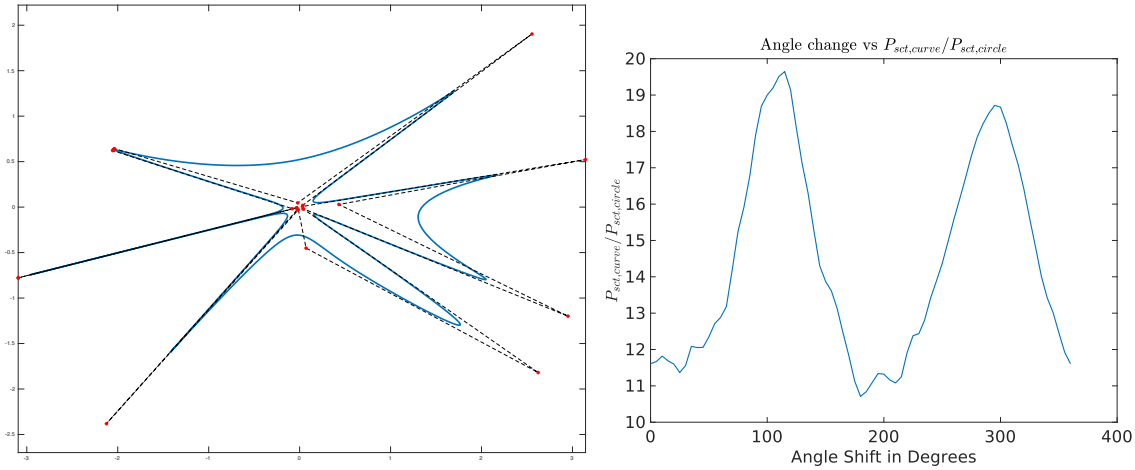
Figure 3.12: Shape Determination, Scattering vs DoFs

Furthermore, this study shows that as DoFs increase the scattering value also increases, which is expected, as can be seen in Figure 3.12. More importantly, the graph shows that as the number of DoFs get to 24 and 27, the improvement in scattering (or the derivative of the scattering) with respect to DoFs is significantly less than the ones with lower DoFs. Thus, we can confidently say that around 30 DoFs would give enough complexity to the geometry in order for it to be optimized for scattering. However, taking into consideration the fact that a local optimizer was used to conduct this study and the fact that a little bit of an increase in computational time could be afforded, mainly 40 DoFs was decided to be used in representing the geometry.

3.1.5 Optimized Shapes Illuminated by a Point Source

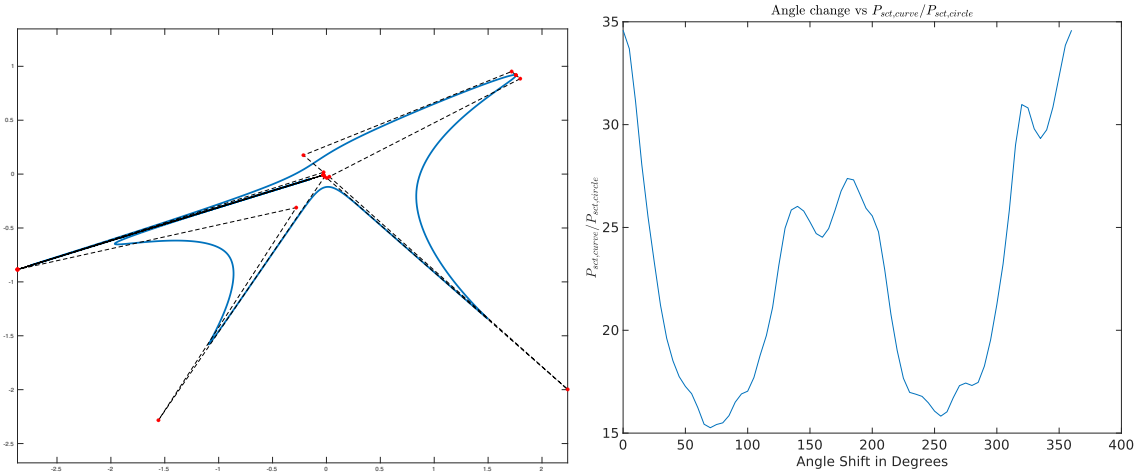
Up to this point, the optimization was done for a relatively high surface complex conductivity, which would not render the tube as transparent. Since one of the main objectives of the project is to get a super scatterer only by changing the shape of an otherwise transparent material, the surface complex conductivity was lowered from $\sigma\eta_0 = 0.12\pi(1+i)$ to $\sigma\eta_0 = 0.0024\pi(1+i)$. The genetic algorithm was applied for global maxima approximation, the result of which was enhanced using the "patternsearch" optimizer and the second, angle-percentage-based, parametric model. Resulting geometries can be seen in Figure 3.13 and 3.14.

Before doing any direct optimization, the geometry that yielded the best result (300-500% scattering increase)for the higher surface conductivity was tested for the lower surface conductivity. This resulted in the scattering being skyrocketed by about 2000%. But, it is obvious that for both the circular nanotube and the shape-optimized nanotube, the overall scattered power goes down as their surface complex conductivity goes down.



(a) (b)

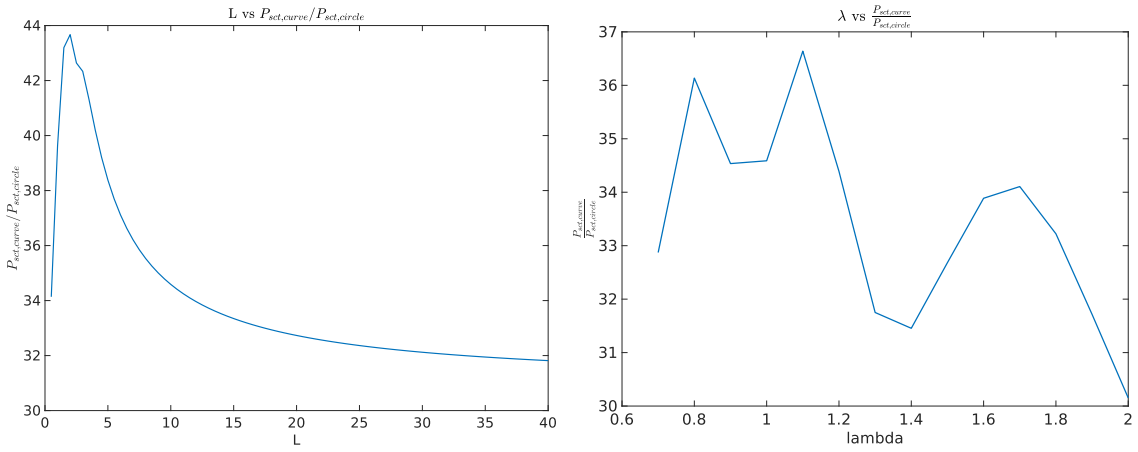
Figure 3.13: Optimized nanotube shape with GA and low conductivity (20 DoFs, Equal-sector-based parametric model). (a) Optimized Nanotube shape with the scattering ratio 19.65 (b) Scattering ratio values change over angle of source incidence to the shape



(a) (b)

Figure 3.14: Optimized by "fmincon" with GA and low conductivity (20 DoFs, Equal-sector-based parametric model). (a) Optimized shape with the scattering ratio 34.57. (b) scattering ratio values change over angle of source incidence to the shape.

Thus, it can be inferred that, with low conductivity, the scattering property of the circle drops down significantly, while the shape optimization's effects become more tangible, which results in the higher scattering ratio between the scattered power by the arbitrarily shaped nanotube and the scatter power by the circular nanotube.



(a) (b)

Figure 3.15: Sensitivity analysis of the nanotube from Figure 3.14. (a) Changing the distance between the point source of EM waves and the nanotube's geometric centroid. (b) Changing the wavelength of the EM waves from the point source.

Next, direct shape optimization was started with $\sigma\eta_0 = 0.0024\pi(1+i)$. The rest of the parameters were kept the same:

- Surface Conductivity, $\sigma\eta_0 = 0.0024\pi(1+i)$
- Wavelength, $\lambda = 1$ unit;
- Area of curve, $A = 2\lambda^2$;
- Distance from point source, $L = 10\lambda$;
- Frequency, $k_0 = 2\pi/\lambda$.

At this point of the project, all the code development and optimization is done in C++. This allowed the process to be much faster and doing optimization using global optimization algorithms can be afforded.

Figure 3.16a shows one of the first results obtained using C++. With 20 DoFs it took around half day and gave a scattering ratio of 9.53. This is probably a local maxima since we could achieve more scattering increase in MATLAB. But still, such value in a short period of time was promising. That is why we decided to increase DoFs 2 times and run the optimization. As a result, we got scattering ratios of 26 and 36 for two simulations with nanotubes that had 40 DoFs on Figures 3.16b and 3.16c. Last result shown on Figures 3.16d gave 85 times increase in scattering value compared to circular nanotube. This for now, the best optimization we could achieve. Overall, from these shapes we can observe a trend of appearing spikes on optimized geometry. We assume that spikes, along with their quantity and length, contribute to the increase of scattering. To be precise, as the number of spikes increases, as well as their length, the scattering value rises significantly.

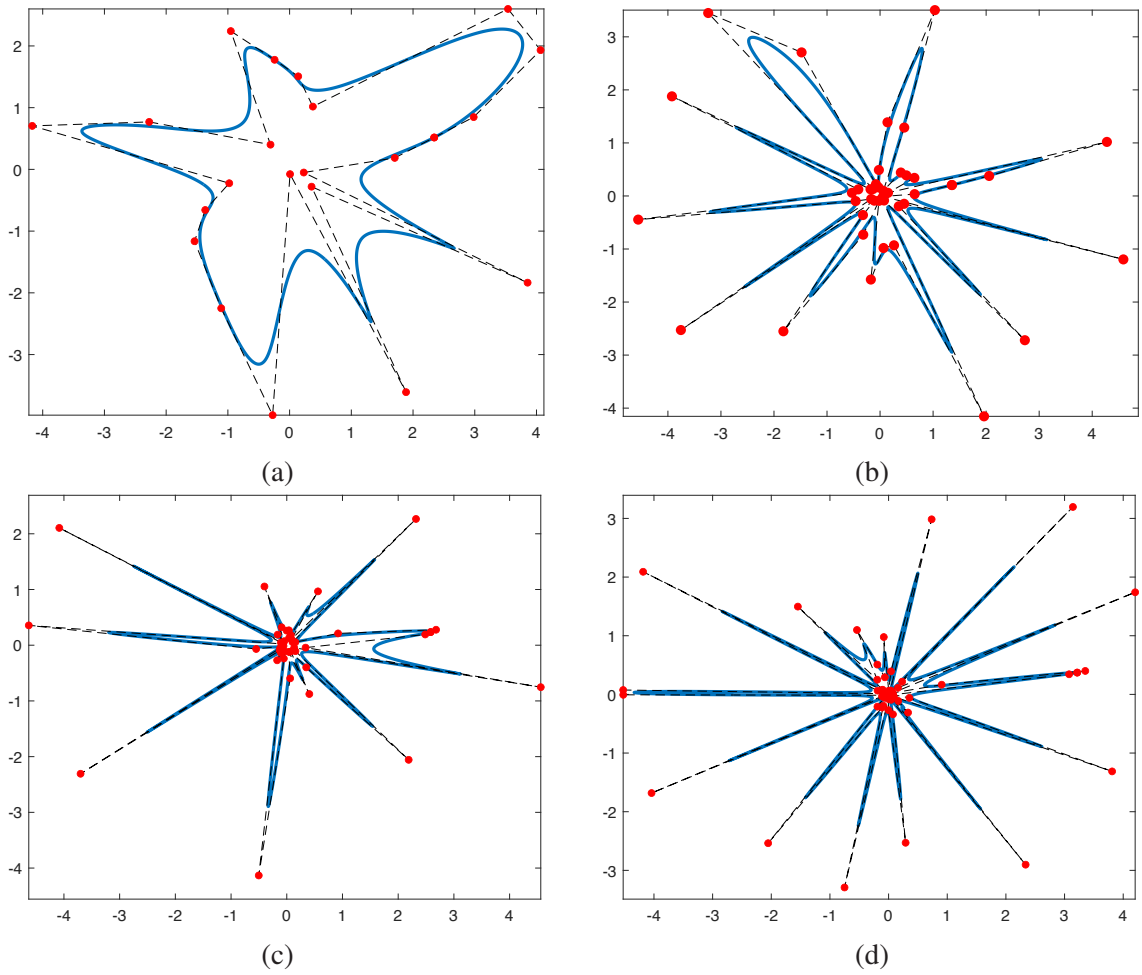


Figure 3.16: Optimized nanotube shapes obtained using "IHS" with different DoFs(a) 20 DoFs, with the scattering ratio 9.53. (b) 40 DoFs, with the scattering ratio 26. (c) 40 DoFs, with the scattering ratio 37. (d) 50 DoFs, with the scattering ratio 85.

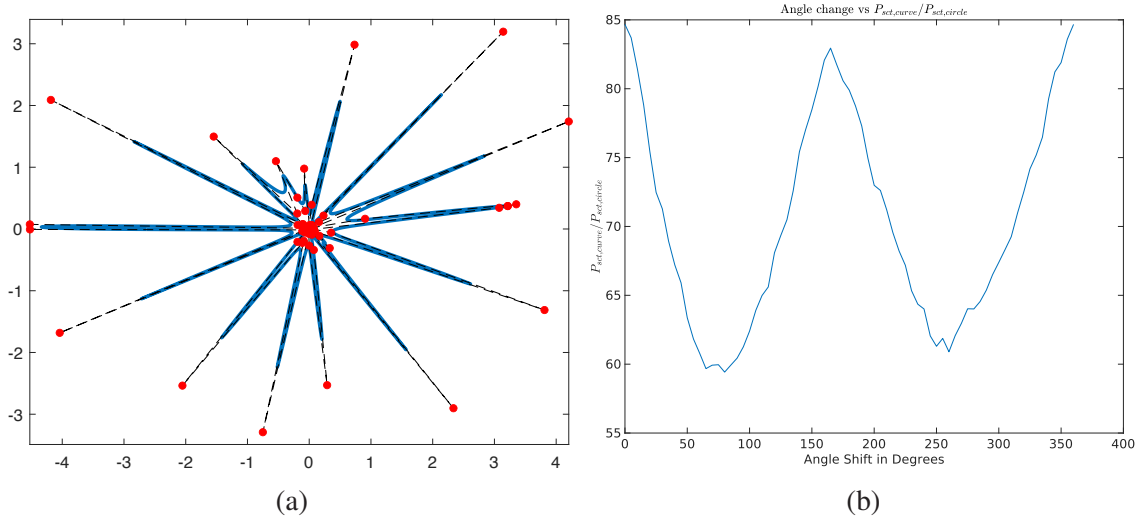


Figure 3.17: Optimized shape with 50 DoFs and its agnle sensetivity (a) Optimized shape with scattering ratio of 85. (b) Angle sensitivity analysis done with step of 5 degrees.

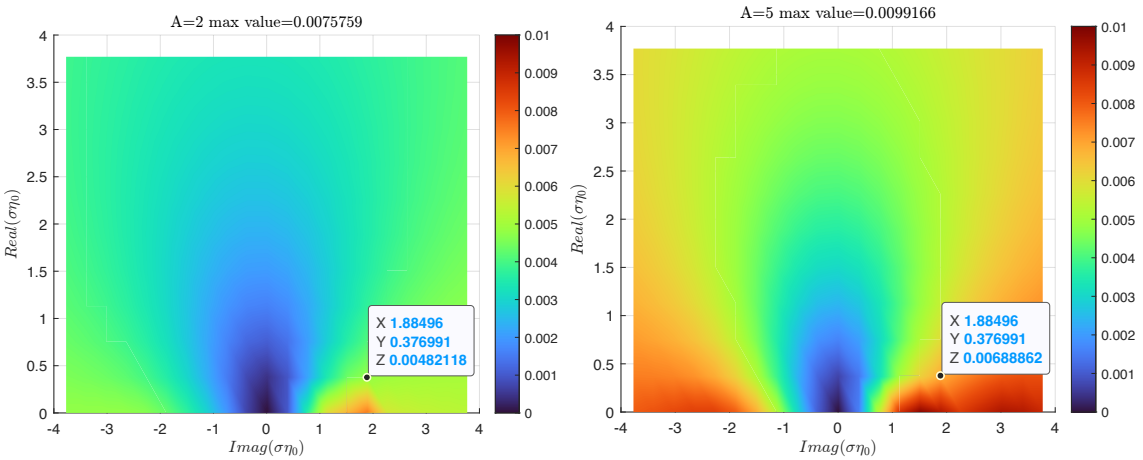
From sensitivity analysis in Figure 3.17b done on incident angle of EM waves, it can

be seen that maximum scattering ratio is achieved at 0 degrees angle and the minimum achieved increase is 59 times higher than the scattering value of the circular nanotube. When the longest spike is directed towards the point source which is illustrated in Figure 3.17a, the scattering value is the highest for this shape. As it was discussed above, spikes' appearance significantly increases the scattering behavior of the nanotube. These angle sensitivity analysis results follow the assumption that the scattering value is positively correlated with the number and length of these spikes. In Figure 3.16c, we can see 8 long spikes and in Figure 3.17a, 12 long spikes appeared.

3.1.6 Optimized Shapes Illuminated by a Planar Wave

Scattering values for the circular nanotubes with small and high surface conductivities $\sigma\eta_0$ and area A varying from $A = 0.5\lambda$ to $A = 10\lambda$ illuminated by a transverse planar wave were determined. The results are presented in Figure 3.18 and Figure 3.20, where X axis is the imaginary part of surface conductivity, Y axis is the real part of surface conductivity and Z axis is the reference scattering value. The conductivity values and the areas displayed on these figures are further used for optimization in C++. Overall, from these results it can be observed that the imaginary part of the surface conductivity plays a huge role and affects resulting scattering significantly and more scattering is produced when the imaginary part of σ is positive. The real part of σ is better to be at its highest value when the imaginary part is minimum.

In Figure 3.18, the search space for large conductivity values is represented. Bottom side of the maps, where the red region appears, caught our attention and we tried to use that conductivity value of $\sigma\eta_0 = 1.885 + 0.377i$ in optimization of the nanotubes with $A = 2\lambda$ and $A = 5\lambda$. In result, 2.54 fold improvement was achieved when $A = 2\lambda$, the shape is illustrated by the curve on the right side of Figure 3.19 and was obtained



(a) (b)

Figure 3.18: Map of scattering values for circular nanotube with area A illuminated by planar wave. (a) $A = 2\lambda$. (b) $A = 5\lambda$.

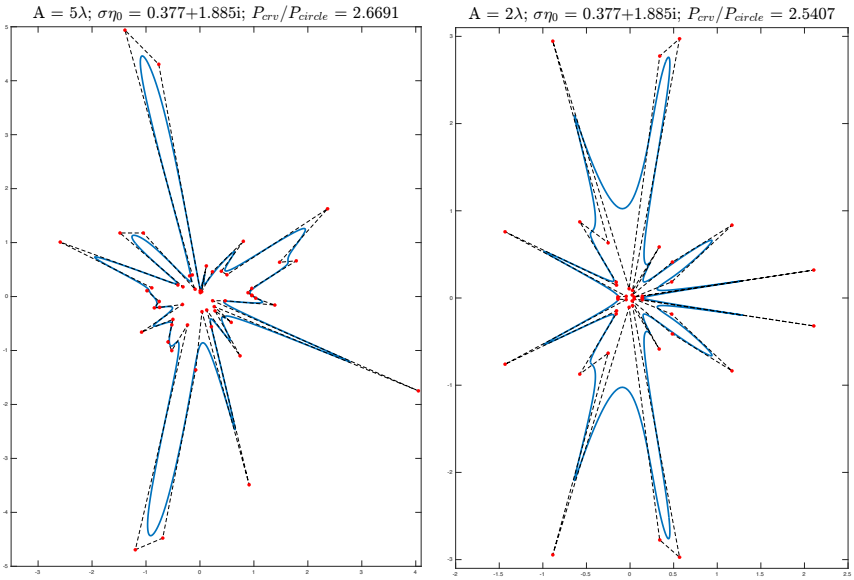


Figure 3.19: Results obtained by taking the parameters from Figure 3.18

.

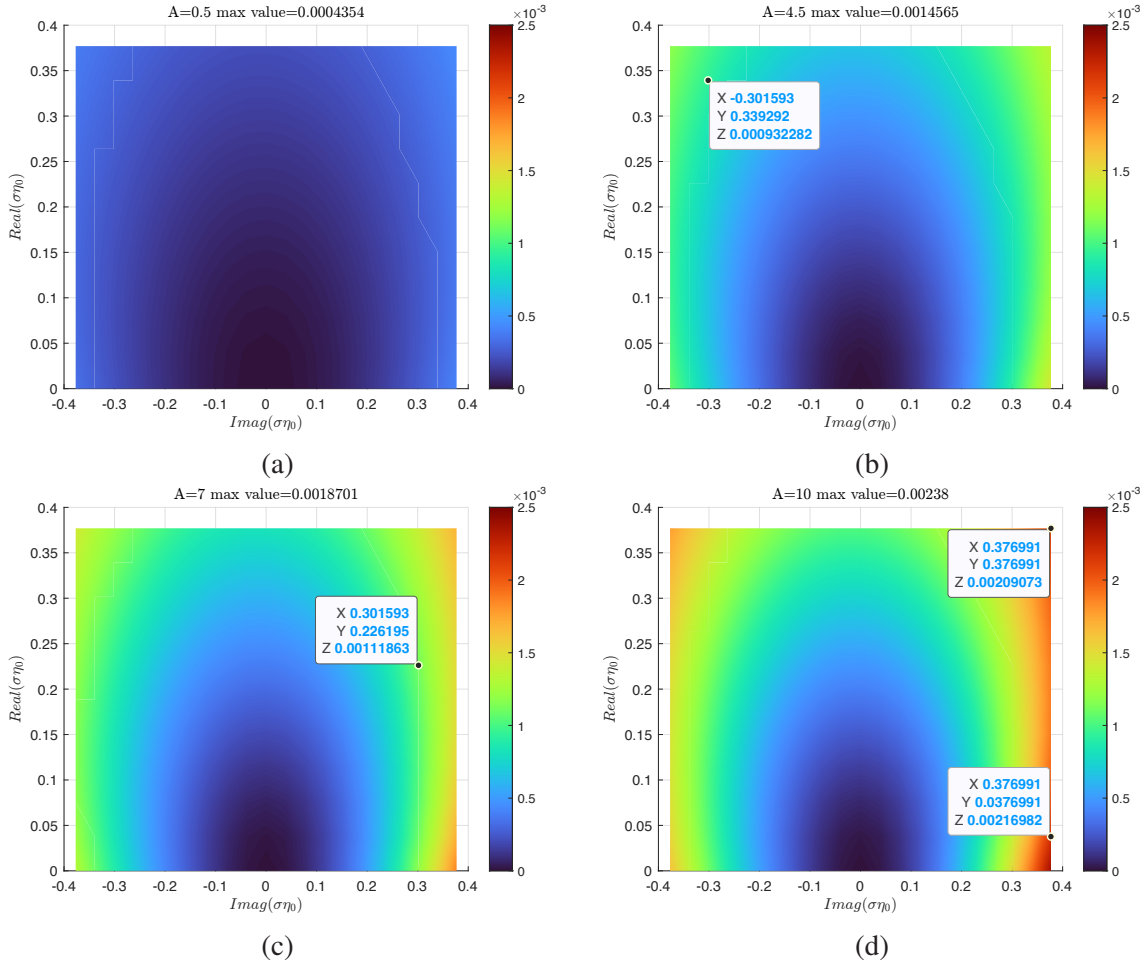


Figure 3.20: Scattering values for different combinations of small complex conductivity values for circular nanotube with area A that is illuminated by a planar wave. (a) $A = 0.5\lambda$. (b) $A = 4.5\lambda$. (c) $A = 7\lambda$. (d) $A = 10\lambda$.

using the symmetric parametric model, while left the curve with $A = 5\lambda$ gave 2.67 times improvement of scattering, which means that increasing the area of the nanotube, increases its scattering property. This is connected to the fact that the bigger the nanotube, the more EM waves it can scatter. And as it was noticed before, decreasing σ increases the scattering, that is why we focused on a search space of low surface conductivity values.

Using Figure 3.20, we set the nanotubes' conductivity values to the ones that are highlighted on the maps. From optimization results in Figure 3.21 it can be seen that the decrease or absence of the real part of surface conductivity indeed increases the scattering of the nanotube. However, it is only a case when the imaginary part of σ is high enough. First three results (obtained using 40 DoFs) in Figure 3.21 show that keeping every other parameter the same and decreasing the real part of surface conductivity by 90% or removing it at all, leads to a 3 times increase in scattering ratio, from 4.8 to 14.15 and 14.85 respectively. This phenomenon is related to the fact that the real part of σ actually conducts the incident EM wave and as it decreases, there is more energy available for the scattering but not the conduction. The results, with small changes in surface conductivity values, illustrated in Figures 3.21d and 3.21e, again show the connection between the area and the scattering property of the nanotube. Here, we observe that as the area decreases, the scattering ratio decreases too. The last result, displayed in Figure 3.21f, obtained using the symmetric parametric model, demonstrates that a nanotube with really small σ is highly likely to behave as a good scatterer after optimization. As a result, we achieved 68 times improvement in scattering with $\sigma\eta_0 = 0.0024\pi(1 + i)$ and small area of $A = 0.5\lambda$.

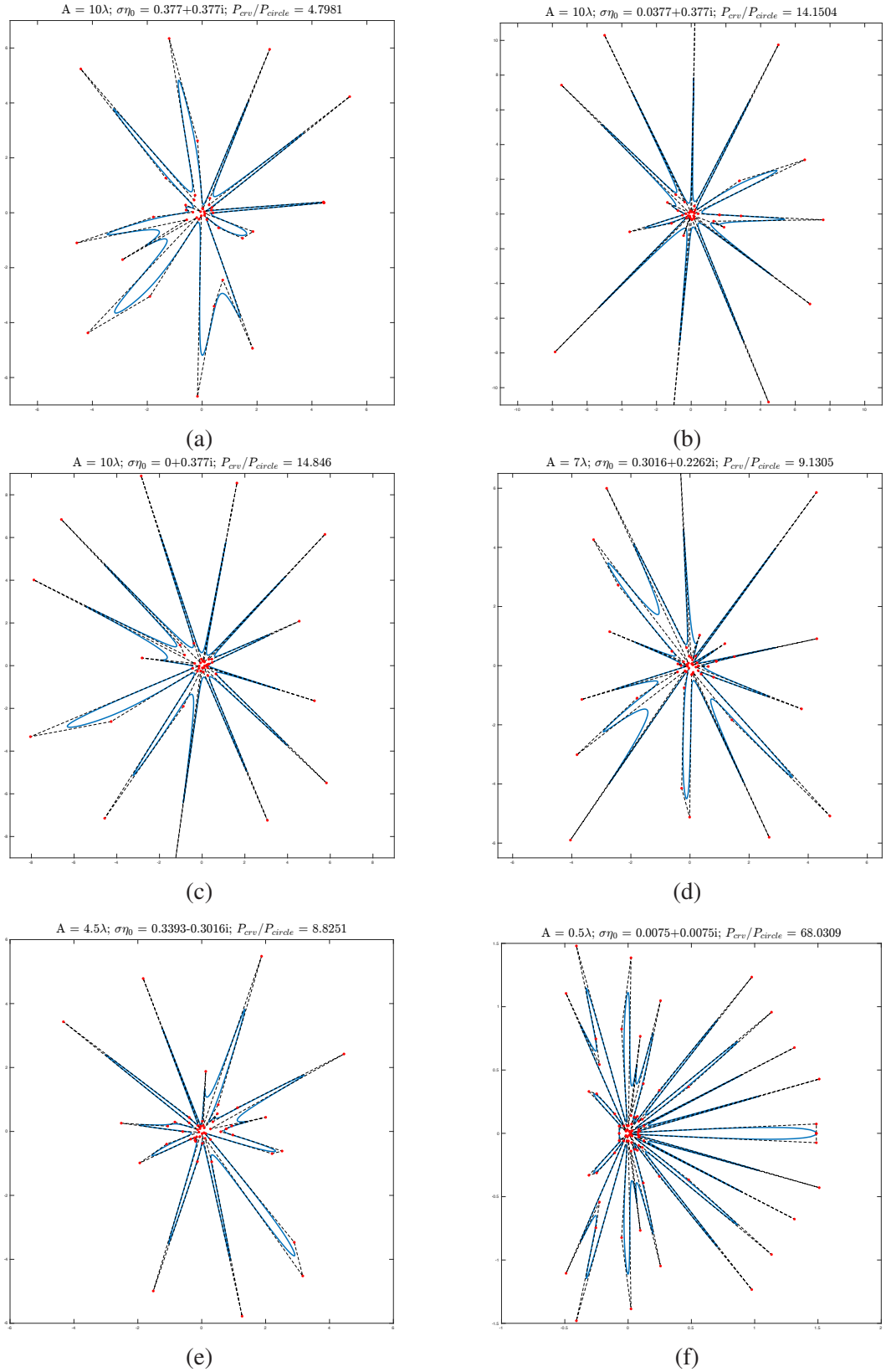


Figure 3.21: Optimization results for nanotubes with medium and small surface conductivity values and other parameters from Figure 3.20

3.1.7 Sensitivity Analysis

Angle Sensitivity

From Equation 2.10 and the results above, we can clearly see the dependence of the scattering on the wave angle, relative to the horizontal axis. For example, in Figure 3.17b, it can be seen that the optimized shape of the nanotube has almost 60 times higher scattering in the worst case scenario compared to the circular nanotube's scattering, while the best scattering ratio achieved for this free-form shape is 85. Furthermore, a better scattering value is obtained when the longest spike is directed towards the source of EM waves.

Distance Sensitivity

Figure 3.15a shows that as the nanotube gets farther from the source, we do not see any change in scattering. This is due to the fact the the circular waves emanating from the point source gradually start to behave as planar waves as the distance from the point source increases.

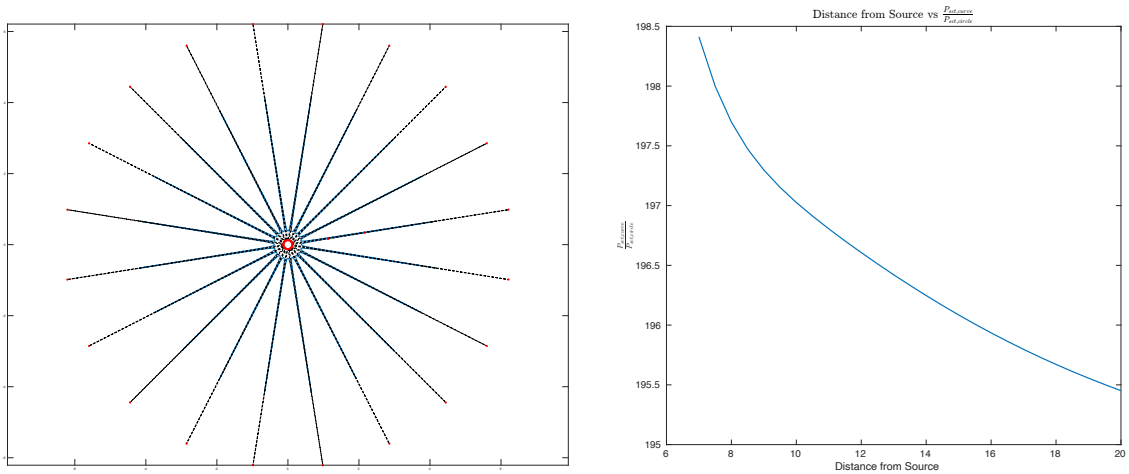
Wavelength Sensitivity

Since we formulated that the ratio between the Area and the square of the wavelength is fixed (namely, $A/\lambda^2 = 2$), only small changes in wavelength are considered. As it can be seen from Figure 3.15b, the scattering does not deviate that much if the wavelength changes are very minor.

3.1.8 Optimum Shapes

Whether the source of the electromagnetic waves is a point source or a planar wave, whether the surface conductivity of the nanotube is small, moderate or high, and no matter what the area of the nanotube is with respect to the wavelength of the wave, the geometries obtained from shape optimization suggest that nanotubes with sharp and long spikes maximize scattering. In order to further confirm that such a shape does, in fact, maximize scattering, a couple of tests were conducted with the shape that can be seen in Figure 3.22a. For a point source that is located at a distance, $L = 10\lambda$, a nanotube with the shown shape with $A = 2\lambda$ and $\sigma\eta_0 = 0.0024\pi(1 + i)$ scatters EM waves about 197 times more than a circular nanotube with the same area. Furthermore, Figure 3.22b shows that as the nanotube get farther from the point source, the scattering ratio, P_{crv}/P_{circle} does not change significantly. Moreover, the same nanotube with the same properties but illuminated by a planar wave achieves a scattering ratio, P_{crv}/P_{circle}, of about 194. As discussed in section 3.1.7, as the nanotube moves farther away from a point source, it reaches a point at which it experiences the incoming waves as planar waves. So, it can be deduced that under a point source, the shape in Figure 3.22a scatters EM waves at least 194

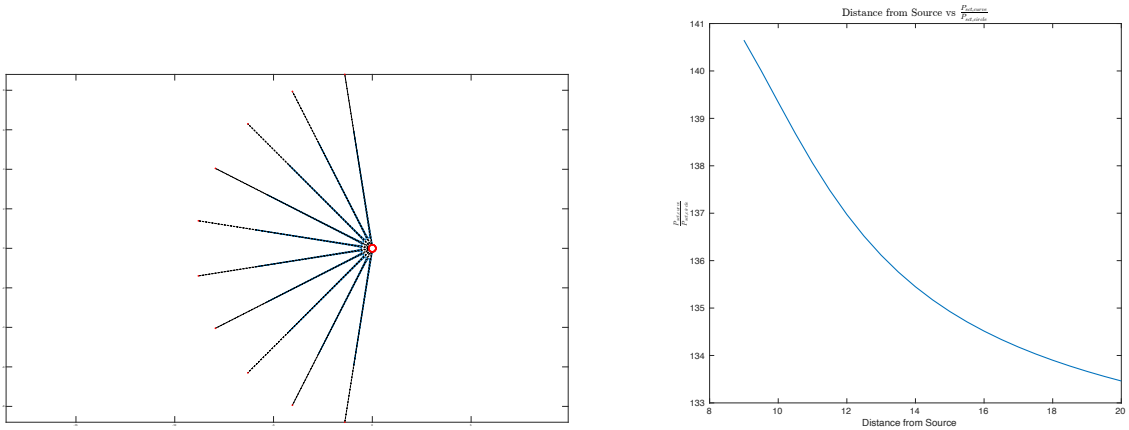
times better (it can go up to 200 times if it is close enough to the source, see Figure 3.22b) than a circular nanotube with the same area.



(a) (b)

Figure 3.22: Shape with 20 spikes. (a) Shape Geometry. (b) Distance Sensitivity.

Next, the same analysis was done for the shape that can be seen in Figure 3.23a. In order for this shape to be more or less comparable to the one in Figure 3.22a, it was generated in a way that it had 10 spikes on one side while the other side was flat. Recall that the shape in Figure 3.22a had 20 spikes all around. Thus, the shape with one flat side can be thought of as the shape in Figure 3.22a cut in half and then scaled up in order for their areas to be the same. First of all, with the same parameters, $A = 2\lambda$, $\sigma\eta_0 = 0.0024\pi(1+i)$ and $L = 10\lambda$, the nanotube with one of its sides being flat achieves a scattering ratio, P_{crv}/P_{circle}, of about 140, which is about 70% of the shape with spikes on both sides (Figure 3.22a). This analysis gets more interesting when we test the shape's angle sensitivity as it shows how the placement of spikes affects the scattering. Surprisingly, an angle sensitivity analysis (see Figure 3.24) shows that if the waves are propagated from left to right (whether it is a point source or a planar wave), the highest scattering is achieved when the spikes of the shape are directed upwards or downwards and not when the spikes are directed right at the wave. The lowest scattering, however, is achieved when the flat side of the shape is directed at the wave, which is expected. In other words, in Figure 3.25, the images on the left and in the middle show the placements of the shape at which it scatters EM waves the best, while the placement that can be seen on the right scatters the worst.



(a) (b)

Figure 3.23: Shape with one side flat, the other with spikes. (a) Shape Geometry. (b) Distance Sensitivity.

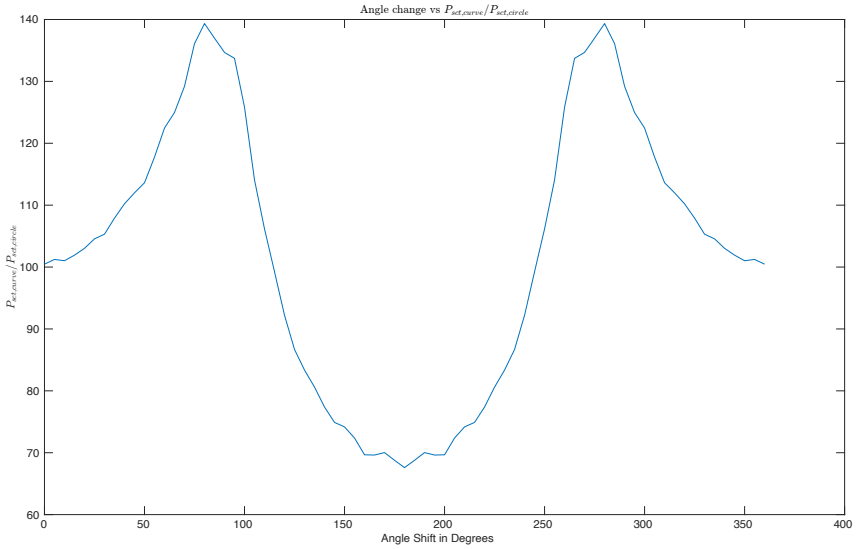


Figure 3.24: Angle sensitivity of shape with one side flat, the other with spikes

.

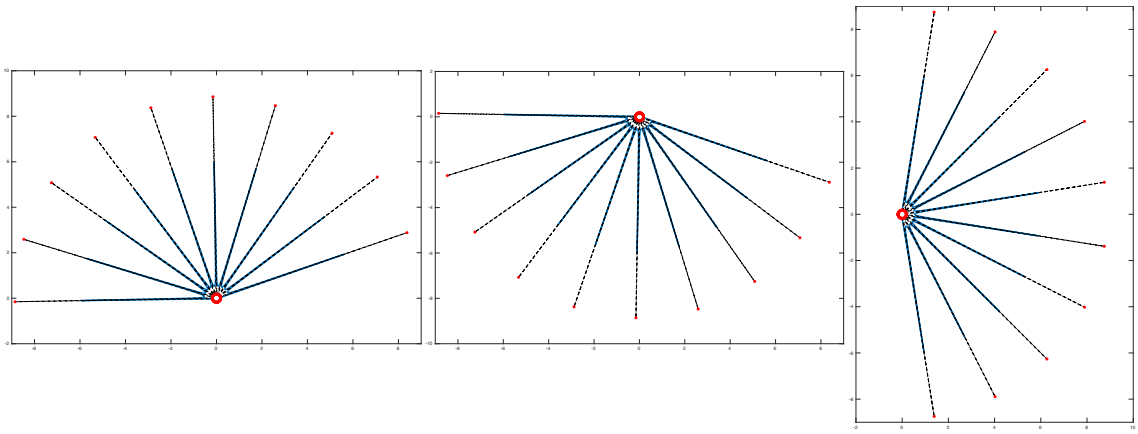


Figure 3.25: Shape with one side flat at different angles.

3.1.8.1 Influence of Number of Spikes on Scattering

As discussed in shape determination analysis (3.1.4), the number of spikes that a nanotube has influences its scattering ability. As stated before, the shape in Figure 3.22a has 20 spikes, but, the number 20 was picked arbitrarily and it is interesting to see what the scattering ratio would be if the number of spikes that a nanotube has is fewer or more than 20 (see Figure 3.26). Thus, a study with nanotubes with different number of spikes was conducted. The area and the surface conductivity of the nanotubes were kept the same (i.e., $A = 2\lambda$, $\sigma\eta_0 = 0.0024\pi(1+i)$) and only the number of spikes that each nanotube has was changed.

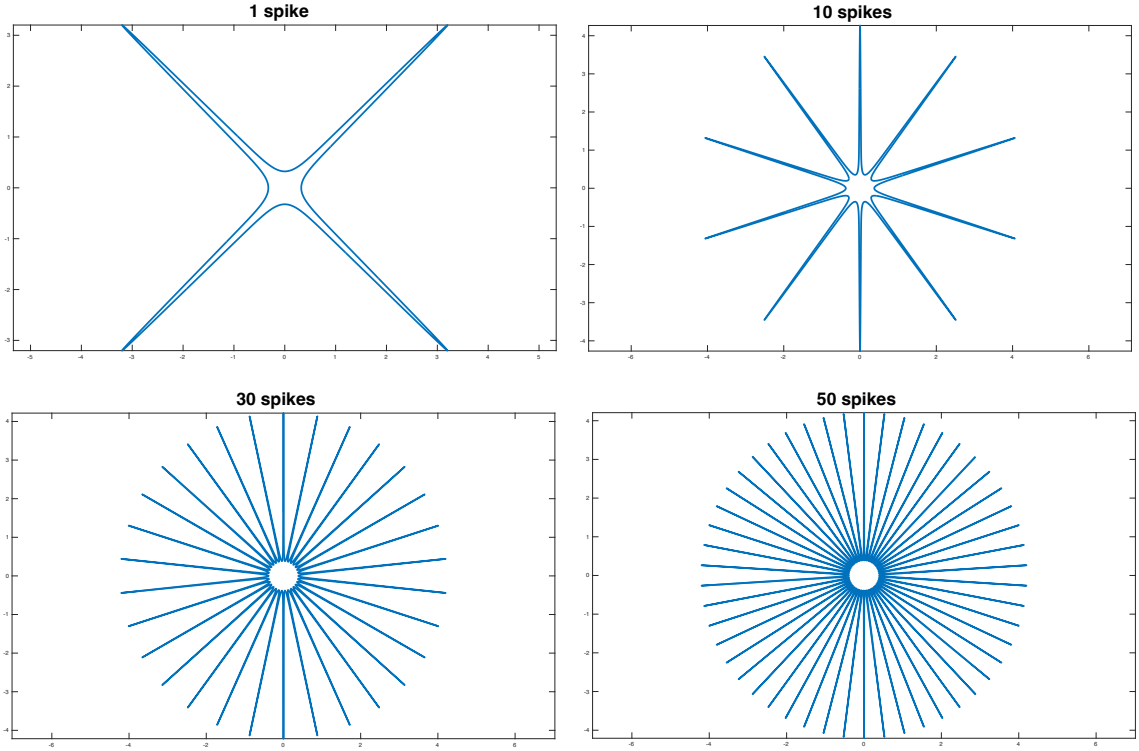


Figure 3.26: Shapes with different number of spikes.

This study is also computationally costly and hence, it was done only up to 50 spikes for a nanotube starting from 4 spikes with an increment of 2. The results are astonishing as can be seen in Figure 3.27. As the number of spikes gets up to 50, the nanotube scatters EM waves about 1000 times better than a circular nanotube of the same area. While a nanotube has to have about 35 spikes all around in order to get an improvement of 500 relative to a circular nanotube. However, even though these are extraordinary results, it is important to note the difficulty in manufacturing any of these shapes as a single tube. However, the improvements that were obtained are so great that they can easily compensate the differences between the CAD models and the final manufactured tubes, assuming metamaterials or other techniques can be used. Furthermore, since the optimization parameters were normalized with respect to the wavelength, the scale of the nanotubes doesn't necessarily have to be in the nanoscale. Which means that with

the current manufacturing processes, there is a high possibility that such tubes can be manufactured given that they are far bigger than the nanoscale.

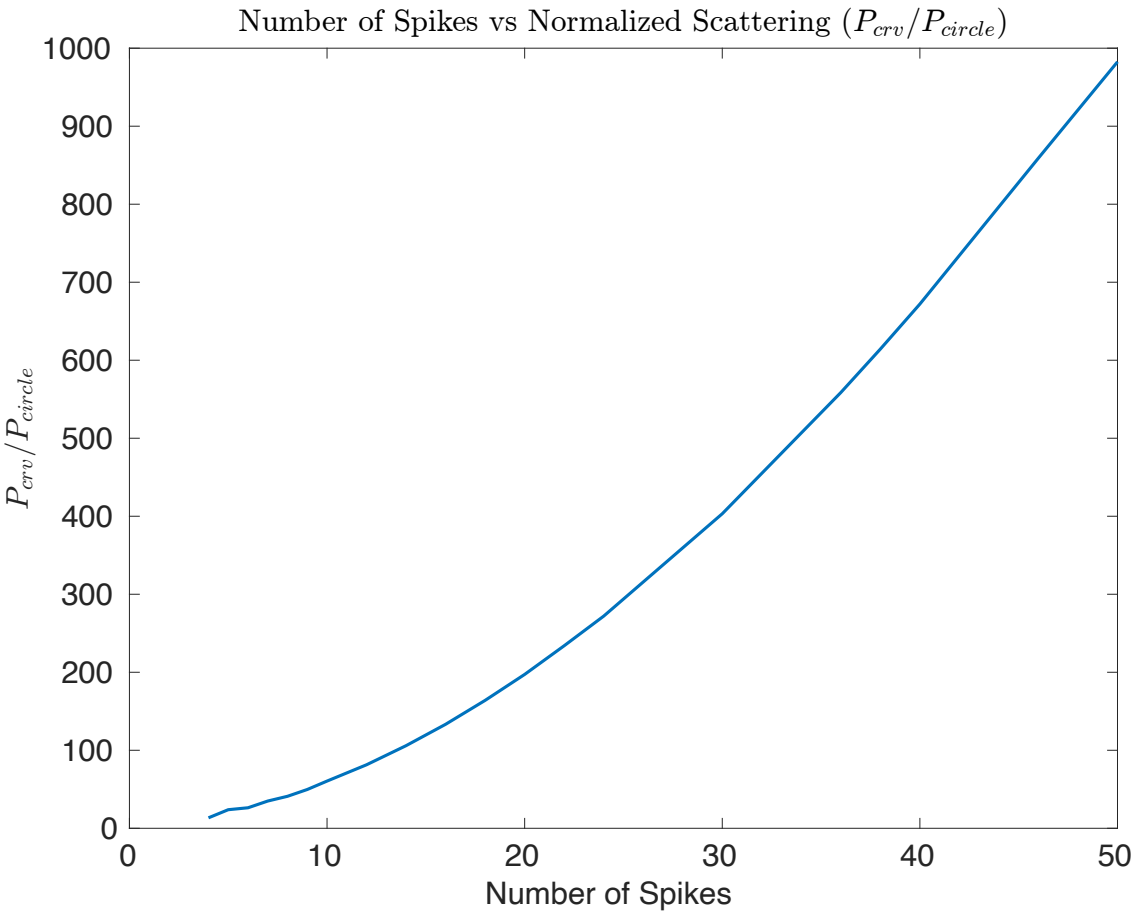


Figure 3.27: Number of scattering vs Normalized scattering, P_{crv}/P_{circle}

3.2 Applications

By boosting a nanotube's scattering property, we propose shielding as a possible application. Namely, the nanotube will act as a superscatterer that creates a region of "weak" EM waves on the opposite side of the superscatterer from a point source of EM waves [32]. Consequently, anything placed behind the superscatterer will be affected by much weaker radiation pressure.

Superscatterers can find their application for optical nanoantennas [33]. Commonly, antennas work as signal receivers, but they can also operate as emitters. Optical nanoantennas are used for an enhancement of EM scattering power [34]. This means that our superscatterers can increase the strength of this optical nanoantenna's property.

Biomedical imaging is another possible field in which superscatters can be used [35]. For example, scattering in coherent X-ray scatter (CXRS) imaging is very crucial [36]. X-ray scattering gives a possibility to display an image of internal organs, or detect explosives during security screenings [36, 37]. Nanoshells with high scattering properties

are valuable for optical coherence tomography and reflectance confocal microscopy [38]. So, superscattering nanotubes can improve and enhance significantly the efficiency of existing biomedical imaging technologies that require scattering of EM waves.

Superscattering nanotubes also bring new possibilities for spectroscopy and nanoplasmonic sensing. Superscattering is vital for surface enhanced Raman spectroscopy (SERS) [39] and helps in studying atoms and molecules, including their structures and electron configurations [40]. Nanotubes that can decently scatter the EM waves will increase the accuracy of nanoplasmonic sensing as well as its quality in general.

3.3 Limitations

In this work, several limitations were encountered, which are the following:

- When using MATLAB, computational time of a single optimization process could take about a week or more. In order to get over this limitation, code development and optimization processes were transferred to C++, which is faster being a low-level language. Moreover, the ability to perform computation processes in parallel added another level of improvement in terms of the computational time. This transition allowed to run optimization processes for only half a day, which is way faster than what MATLAB could offer;
- Due to their constraints, geometric parametric models cannot always construct the best possible geometry for a nanotube that leads to the highest scattering. In other words, the shape optimization results cannot be considered as the most wanted shape. Instead, they can only be considered to be similar to the best shape possible for scattering;
- Even with the transition to the C++ environment, the computational time can reach several days. This limits the possibility to fully explore the search space and do more sensitivity analyses.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

A systematic analysis of scattering as a function of nanotubes' shapes was performed. As we were aiming in scattering maximization, a shape optimization problem was formulated to determine the shape that improves scattering when compared to the scattering values of the corresponding circular tubes. Scattering estimation was performed using a novel isogeometric analysis-based boundary element method (IGABEM) which was tested and verified against analytic solutions. Furthermore, a family of parametric models were developed to secure the generation of valid (non self-intersecting) and accurate NURBS representations of the nanotube boundaries. These parametric modelers enabled, along with the IGABEM solver enabled the automated execution of design optimization with both local and global optimizers. Optimized nanotubes present a noticeable improvement of the electromagnetic scattering when compared to the corresponding circular nanotubes. Considering the results from sensitivity analysis, for a high complex surface conductivity, scattering increases at least 3.5 times (3.6b) when placing the point source at any angle relative to the optimized nanotube.

The results become more impressive with scattering enhancements exceeding 85 times, when low electrical conductivity values are considered. This is a very promising result as one of the primary aims is to increase interaction of weakly interacting nanotubes. Optimization is clearly opting for several design possibilities over other design choices.

After studying effects of spikes on the scattering, it was found that the number of spikes has a linear relation with the resultant scattering ratio as it was shown in Figure 3.27, which was done for low surface conductivity, with its value reaching 1000 for 50 equally distributed spikes and a scattering ratio of 500 for 35 spikes in the geometry. Moreover, as the distance from the source increases, the geometries with a high number of spikes have lowest possible drop in scattering which signifies its effectiveness even with planar waves, with a shape with 20 spikes yielding a scattering ratio of around 195 for a planar wave and 200 near a point source (3.22). All of these results highlight that the best possible shapes for scattering can be achieved by introducing spikes into the geometry.

Applications of such superscatterrers would be crucial in current technologies such as EM-based biomedical imaging [35], like CXRS [36], or surface enhanced Raman Spectroscopy (SERS) [39], since these applications depend on EM scattering significantly and a better scatter would lead to a better product quality.

4.2 Future Work

In future prospects, this work can be further expanded by introducing several additional symmetric parametric models. The symmetric parametric model that was used in this work was symmetric only along the x-axis. However, it can also be modified so that the geometry is symmetric along both the x-axis and the y-axis. In this way, DoFs of the geometry will be quartered in total and the computational time for the optimization would be quartered.

Furthermore, validating the results of this work by conducting experimental studies in the laboratory would be a reasonable continuation of this work.

More importantly, the work can be carried on by introducing an array of self-interacting nanotubes and by doing shape optimization for this array of nanotubes in order to achieve higher values of scattering. In such a way, a nanotube-based metamaterial can be constructed and its application as EM scatterrer can be researched.

4.3 Distribution of Tasks

Tasks were distributed among the team as follows:

Student Name	Tasks
Madeniyet Bespayev	Angle sensitivity. Implementation of fmincon and patternsearch. Optimization for low surface conductivity. Enhancing GA results with patternsearch. Transition to C++. Exploring the search spaces. Implementation of the symmetric parametric model.
Yerassyl Turarov	Optimization for low DoFs. Implementation of equal-sector and angle-percentage-based parametric models. Transition to C++. Distance and Wavelength Sensitivity. Optimization for Planar Waves.
Nurkeldi Iznat	Mathematical Formulation. Verification of the numerical solution. Convergence study. Computational Time study. Transition to C++. Integration of PAGMO (COBYLA, Improved Harmony Search) optimizers into the optimization process. Optimization with GA, IHS for high DoFs. Investigation of the influence of the number of spikes on Scattering.

Bibliography

[1] A. Aqel, K. Abou El-Nour, R. Ammar, and A. Al-Warthan, "Carbon nanotubes, science and technology part (i) structure, synthesis and characterisation," *Arabian Journal of Chemistry*, 2012.

[2] J. Ouyang, "Applications of carbon nanotubes and graphene for third-generation solar cells and fuel cells," *Nano Materials Science*, 2019.

[3] J. Chen, S. Wei, and H. Xie, "A Brief Introduction of Carbon Nanotubes: History, Synthesis, and Properties," *Journal of Physics: Conference Series*, 2021.

[4] J. Kim, T. Pham, and J. Hwang, "Boron nitride nanotubes: synthesis and applications," *Nano Convergence*, 2018.

[5] Y. Dahman, *In Micro and Nano Technologies, Nanotechnology and Functional Materials for Engineers*, ch. 11 - Nanomedicine, pp. 229–249. Elsevier Science, 2017.

[6] R. K. Mishra, J. Cherusseri, E. Allahyari, S. Thomas, and N. Kalarikkal, "Chapter 10 - Small-Angle Light and X-ray Scattering in Nanosciences and Nanotechnology," in *Thermal and Rheological Measurement Techniques for Nanomaterials Characterization* (S. Thomas, R. Thomas, A. K. Zachariah, and R. K. Mishra, eds.), Micro and Nano Technologies, pp. 233–269, Elsevier, 2017.

[7] E. Alozie, A. Musa, N. Faruk, A. L. Imoize, A. Abdulkarim, A. D. Usman, Y. O. Imam-Fulani, K. S. Adewole, A. A. Oloyede, O. A. Sowande, S. Garba, B. A. Baba, Y. A. Adediran, and L. S. Taura, "A review of dust-induced electromagnetic waves scattering theories and models for 5g and beyond wireless communication systems," *Scientific African*, vol. 21, p. e01816, 2023.

[8] Y. Gregory, A. Nikolai, A. Sergey, L. Akhlesh, and M. Oleg, "Electromagnetic Wave Scattering by the Edge of a Carbon Nanotube," *AEU - International Journal of Electronics and Communications*, 2001.

[9] A. V. Melnikov, P. P. Kuzhir, S. A. Maksimenko, G. Y. Slepyan, A. Boag, O. Pulci, I. A. Shelykh, and M. V. Shuba, "Scattering of electromagnetic waves by two crossing metallic single-walled carbon nanotubes of finite length," *Phys. Rev. B*, vol. 103, p. 075438, Feb 2021.

[10] G. Y. Slepyan, M. V. Shuba, S. A. Maksimenko, and A. Lakhtakia, "Theory of optical scattering by achiral carbon nanotubes and their potential as optical nanoantennas," *Phys. Rev. B*, vol. 73, p. 195416, May 2006.

[11] K. Kostas and V. Costas, "Optimally Shaped Nanotubes for Field Concentration," 2023.

[12] M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions*. Cambridge University Press, 1964.

[13] D. Wang, Q. Dongliang, and X. Li, "Superconvergent isogeometric collocation method with greville points," *Computer Methods in Applied Mechanics and Engineering*, 2021.

[14] T. F. Coleman and Y. Li, "On the Convergence of Reflective Newton Methods for Large-scale Nonlinear Minimization Subject to Bounds," tech. rep., Cornell University, 1992.

[15] M. A. Abramson, *Pattern Search Algorithms for Mixed Variable General Constrained Optimization Problems*. PhD thesis, RICE UNIVERSITY, 2002.

[16] D. E. Goldberg, *Genetic Algorithms in Search, Optimization Machine Learning*. Addison-Wesley, 1989.

[17] Kitware, "CMake Reference Documentation." https://cmake.org/cmake/help/latest/. Accessed: 2023-12-20.

[18] GNU org., "GSL - GNU scientific library." https://www.gnu.org/software/gsl/. Accessed: 2023-12-20.

[19] Sintef, "Geometry Toolkits / GoTools." https://www.sintef.no/projectweb/geometry-toolkits/gotools/. Accessed: 2023-12-20.

[20] Intel, "Intel® oneAPI Threading Building Blocks." https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html. Accessed: 2024-02-3.

[21] OpenMPI, "A High Performance Message Passing Library." https://www.open-mpi.org/. Accessed: 2023-12-20.

[22] European Space Agency, "Pagmo 2.19.0 documentation." https://esa.github.io/pagmo2/. Accessed: 2024-02-3.

[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] R. M. L. Tamara G. Kolda and V. Torczon, "Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods," *SIAM Journal Vol. 45, No. 3, pp. 385-482*, 2003.

[25] M. J. D. Powell, "Direct search algorithms for optimization calculations," *Cambridge University Press*, 2008.

[26] 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.

[27] "An overview of the watson transformation presented through a simple example," *Progress In Electromagnetics Research*, vol. 75, pp. 137–152, 2007.

[28] C. Politis, A. I. Ginnis, P. D. Kaklis, K. Belibassakis, and C. Feurer, "An isogeometric BEM for exterior potential-flow problems in the plane," in *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, (New York, NY, USA), p. 349–354, Association for Computing Machinery, 2009.

[29] A. Ginnis, K. Kostas, C. Politis, P. Kaklis, K. Belibassakis, T. Gerostathis, M. Scott, and T. Hughes, "Isogeometric boundary-element analysis for the wave-resistance problem using t-splines," *Computer Methods in Applied Mechanics and Engineering*, vol. 279, pp. 425–439, 2014.

[30] K. Kostas, M. Fyrillas, C. Politis, A. Ginnis, and P. Kaklis, "Shape optimization of conductive-media interfaces using an IGA-BEM solver," *Computer Methods in Applied Mechanics and Engineering*, vol. 340, pp. 600–614, 2018.

[31] K. Kostas, C. Politis, I. Zhanabay, and P. Kaklis, "Geometry parametrization effects in Igabem Analysis," *Available at SSRN: https://ssrn.com/abstract=4239433 or http://dx.doi.org/10.2139/ssrn.4239433*, 2022.

[32] A. Canós Valero, H. K. Shamkhi, A. Kupriianov, T. Weiss, A. Pavlov, D. Redka, V. Bobrovs, Y. Kivshar, and A. Shalin, "Superscattering emerging from the physics of bound states in the continuum," *Nature Communications*, vol. 14, 08 2023.

[33] Z. Ruan and S. Fan, "Superscattering of light from subwavelength nanostructures," *Phys. Rev. Lett.*, vol. 105, p. 013901, Jun 2010.

[34] J. Schuller, T. Taubner, and M. Brongersma, "Optical antenna thermal emitters," *Nature Photonics - NAT PHOTONICS*, vol. 3, 10 2009.

[35] C. Qian, X. Lin, Y. Yang, F. Gao, Y. Shen, J. Lopez, I. Kaminer, B. Zhang, E. Li, M. Soljačić, and H. Chen, "Multifrequency superscattering from subwavelength hyperbolic structures," *ACS Photonics*, vol. 5, 02 2018.

[36] G. Harding and B. Schreiber, "Coherent x-ray scatter imaging and its applications in biomedical science and industry," *Radiation Physics and Chemistry*, vol. 56, no. 1, pp. 229–245, 1999.

[37] M. Bech, A. Tapfer, A. Velroyen, A. Yaroshenko, B. Pauwels, J. Hostens, P. Bruyndonckx, A. Sasov, and F. Pfeiffer, "In-vivo dark-field and phase-contrast x-ray imaging," *Scientific reports*, vol. 3, p. 3209, 11 2013.

[38] C. Loo, A. Lowery, N. Halas, J. West, and R. Drezek, "Immunotargeted nanoshells for integrated cancer imaging and therapy," *Nano Letters*, vol. 5, no. 4, pp. 709–711, 2005.

[39] J. Ye, F. Wen, H. Sobhani, J. B. Lassiter, P. Van Dorpe, P. Nordlander, and N. J. Halas, "Plasmonic Nanoclusters: Near Field Properties of the Fano Resonance Interrogated with SERS," *Nano Letters*, vol. 12, no. 3, pp. 1660–1667, 2012.

[40] K. Lim and S. Lewis, "Spectroscopic techniques," in *Encyclopedia of Forensic Sciences (Second Edition)* (J. A. Siegel, P. J. Saukko, and M. M. Houck, eds.), pp. 627–634, Waltham: Academic Press, second edition ed., 2013.

Appendices

Appendix A

Source Code - C++ Code

WICAN_scat

Generated by Doxygen 1.10.0

1 Class Documentation **49**

1.1 analysis_curves Class Reference 49

1.1.1 Constructor & Destructor Documentation 51

1.1.2 Member Function Documentation 52

1.1.3 Member Data Documentation 60

1.2 BIE_gsl_f_params Struct Reference 61

1.2.1 Member Data Documentation 62

1.3 gsl_function_pp< F > Class Template Reference 62

1.3.1 Constructor & Destructor Documentation 63

1.3.2 Member Function Documentation 63

1.3.3 Member Data Documentation 63

1.4 IntegrationWorkspace Class Reference 64

1.4.1 Constructor & Destructor Documentation 64

1.4.2 Member Function Documentation 64

1.4.3 Member Data Documentation 64

1.5 opt_max_scat_area_const Class Reference 65

1.5.1 Constructor & Destructor Documentation 66

1.5.2 Member Function Documentation 66

1.5.3 Member Data Documentation 67

1.6 opt_max_scat_wo_area Class Reference 67

1.6.1 Constructor & Destructor Documentation 68

1.6.2 Member Function Documentation 69

1.6.3 Member Data Documentation 69

1.7 opt_problem Class Reference 69

1.7.1 Constructor & Destructor Documentation 70

1.7.2 Member Function Documentation 70

1.7.3 Member Data Documentation 71

1.8 opt_problem_data Struct Reference 71

1.8.1 Member Data Documentation 72

2 File Documentation **73**

2.1 angle_sens.cpp File Reference 73

2.1.1 Function Documentation 73

2.2 comp_time.cpp File Reference 73

2.2.1 Function Documentation 73

2.3 distSens.cpp File Reference 74

2.3.1 Function Documentation 74

2.4 numSpikes.cpp File Reference 74

2.4.1 Function Documentation 74

2.5 openMPI_par_dist.cpp File Reference 75

2.5.1 Function Documentation 75

2.6 scratch.cpp File Reference 75

2.6.1 Function Documentation . . . 76
2.7 test_mesh.cpp File Reference . . . 76
2.7.1 Function Documentation . . . 76
2.8 test_opt.cpp File Reference . . . 76
2.8.1 Function Documentation . . . 77
2.9 test_opt_old.cpp File Reference . . . 77
2.9.1 Function Documentation . . . 77
2.10 test_pagmo.cpp File Reference . . . 77
2.10.1 Function Documentation . . . 78
2.11 test_splineCurve.cpp File Reference . . . 78
2.11.1 Function Documentation . . . 78
2.12 analysis_NURBSCurve.h File Reference . . . 79
2.12.1 Macro Definition Documentation . . . 79
2.12.2 Function Documentation . . . 79
2.13 analysis_NURBSCurve.h . . . 80
2.14 integration.h File Reference . . . 81
2.14.1 Macro Definition Documentation . . . 81
2.14.2 Function Documentation . . . 82
2.15 integration.h . . . 82
2.16 opt_problems.h File Reference . . . 82
2.16.1 Macro Definition Documentation . . . 83
2.16.2 Function Documentation . . . 83
2.17 opt_problems.h . . . 83
2.18 special_functions.h File Reference . . . 84
2.18.1 Macro Definition Documentation . . . 85
2.18.2 Function Documentation . . . 85
2.19 special_functions.h . . . 87
2.20 analysis_NURBSCurve.cpp File Reference . . . 87
2.21 opt_problems.cpp File Reference . . . 87
2.22 special_functions.cpp File Reference . . . 88
2.22.1 Function Documentation . . . 88

Index **91**

1 Class Documentation

1.1 analysis_curves Class Reference

`#include <analysis_NURBSCurve.h>`

Public Member Functions

- analysis_curves ()
  default constructor
- ~analysis_curves ()
  default destructor
- analysis_curves (double wavelength, complex< double > sigma, double area, double dist)
  constructer with initializers
- 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 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_index
- 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, unsigned int curve_index=0)
  generate a closed curve of arbitrary shape using the 1st parametric model at position curve_index

- void CurveFromParameters_pm2 (vector< double > parameters, double r_max, double area0, unsigned int curve_index=0)

  generate a closed curve of arbitrary shape using the 2nd parametric model (with a given enclosed area) at position curve_index
- void CurveFromParameters_dynamic_pm1 (vector< double > parameters, double r_max, unsigned int curve_index=0)

  generate a closed curve of arbitrary shape using the dynamic parametric model at position curve_index
- void CurveFromParameters_dynamic_pm2 (vector< double > parameters, double r_max, double area0, unsigned int curve_index=0)

  generate a closed curve of arbitrary shape using the dynamic parametric model (with a given enclosed area) at position curve_index
- double CurveFromParameters_pm3 (vector< double > parameters, double r_max, double area0, int &error_flag, unsigned int curve_index=0)
- 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, complex< double > sigma, double L=-1., double angle=0, 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, complex< double > sigma, double L=-1., double angle=0, 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 ComputeFieldOnSingleCurve_GSL (double wavelength, complex< double > sigma, double L=-1., double angle=0, unsigned int curve_index=0)

  compute electric field on single curve from a given background field using GSL integration
- void ComputeFieldOnSingleCurve_MPI (double wavelength, complex< double > sigma, double L=-1., double angle=0, 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 quantity on mesh

- double CalculateQuantityOnMesh_GSL (double wavelength, double angle, complex< double > sigma, unsigned int curve_index=0, double tolerance=1e-4, string filename="")
- double Scattering_analytic (double r, complex< double > sigma, double lambda, double L)

  calculate the EM Scattering Analytically given a Point Source at a distance L
- double Scattering_numeric (complex< double > sigma, double wavelength, unsigned int curve_index=0)

  calculate the EM Scattering Numerically given the solution of the electric field on the boundary
- double ComputeScattering (double wavelength, double angle, complex< double > sigma, unsigned int curve_index=0)

  calculate the EM Scattering Numerically given the solution of the electric field on the boundary

Private Member Functions

- SplineCurve ComputeSplineCurve (vector< double > params, double rv, vector< double > knots, unsigned int k, unsigned int n)
- SplineCurve ComputeSplineCurve_dynamic (vector< double > params, double rv, vector< double > knots, unsigned int k, unsigned int n)
- SplineCurve ComputeSymSplineCurve (vector< double > params, double rv, unsigned int k)

  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)
- void updateAnalysisModel ()
- void shiftGreville (valarray< double > t, unsigned int curve_index)

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.
- double lambda_class

  Wavelength.
- complex< double > sigma_class

  Conductivity.
- double area_class

  Area of Nanotube.
- double dist_class

  Distance from Point Source (-1 if the background field is plane wave)

1.1.1 Constructor & Destructor Documentation

analysis_curves() [1/2]

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

default constructor

~analysis_curves()

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

default destructor

analysis_curves() [2/2]

```
analysis_curves::analysis_curves (
            double wavelength,
            complex< double > sigma,
            double area,
            double dist )
```

constructer with initializers

1.1.2 Member Function Documentation

addAnalysisCurve()

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

add analysis curve in the analysis model

addCurve()

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

add a curve in the CAD model

addGrevilleAbscissae()

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

add greville Abscissae

CalculateGreville()

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

calculate Greville Abscissae and Points

CalculateQuantityOnMesh()

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

calculate quantity on mesh

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 = "" )
```

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

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)

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)

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.

ComputeFieldOnSingleCurve()

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

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

ComputeFieldOnSingleCurve_GSL()

```
void analysis_curves::ComputeFieldOnSingleCurve_GSL (
            double wavelength,
            complex< double > sigma,
            double L = -1.,
            double angle = 0,
            unsigned int curve_index = 0 )
```

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

ComputeFieldOnSingleCurve_MPI()

```
void analysis_curves::ComputeFieldOnSingleCurve_MPI (
            double wavelength,
            complex< double > sigma,
            double L = -1.,
            double angle = 0,
            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

ComputeFieldOnSingleCurve_TBB()

```
void analysis_curves::ComputeFieldOnSingleCurve_TBB (
            double wavelength,
            complex< double > sigma,
            double L = -1.,
            double angle = 0,
            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

ComputeScattering()

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

calculate the EM Scattering Numerically given the solution of the electric field on the boundary

ComputeSolutionValues()

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

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

ComputeSplineCurve()

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

ComputeSplineCurve_dynamic()

```
SplineCurve analysis_curves::ComputeSplineCurve_dynamic (
            vector< double > params,
            double rv,
            vector< double > knots,
            unsigned int k,
            unsigned int n )  [private]
```

ComputeSymSplineCurve()

```
SplineCurve analysis_curves::ComputeSymSplineCurve (
            vector< double > params,
            double rv,
            unsigned int k )  [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)

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

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)

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)

CurveCentroid()

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

get closed curve centroid

CurveFromParameters_dynamic_pm1()

```
void analysis_curves::CurveFromParameters_dynamic_pm1 (
            vector< double > parameters,
            double r_max,
            unsigned int curve_index = 0 )
```

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

CurveFromParameters_dynamic_pm2()

```
void analysis_curves::CurveFromParameters_dynamic_pm2 (
            vector< double > parameters,
            double r_max,
            double area0,
            unsigned int curve_index = 0 )
```

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

CurveFromParameters_pm1()

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

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

CurveFromParameters_pm2()

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

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

CurveFromParameters_pm3()

```
double analysis_curves::CurveFromParameters_pm3 (
            vector< double > parameters,
            double r_max,
            double area0,
            int & error_flag,
            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; error_flag must return 0 for normal operation.

CurveLength()

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

get length of curve at curve_index

DoFs()

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

get number of DoFs

getAnalysisCurve()

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

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

GetGreville()

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

get i greville Abscissa from analysis model j

GetGrevilleAbscissa()

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

get greville Abscissa

getSolution()

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

get the solution for the ith curve

isCurveClosed()

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

check if curve at curve_index is closed

memory_size()

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

compute memory used for curves storage

nCurves()

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

get number of surfaces in model

operator[]()

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

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

PrintCurveInfo()

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

print curve info

PrintSolution()

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

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

read_iges_file()

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

read curve from IGES file

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.

save_analysis_model()

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

save analysis curves in IGES file

Scattering_analytic()

```
double analysis_curves::Scattering_analytic (
            double r,
            complex< double > sigma,
            double lambda,
            double L )
```

calculate the EM Scattering Analytically given a Point Source at a distance L

Scattering_numeric()

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

calculate the EM Scattering Numerically given the solution of the electric field on the boundary

shiftGreville()

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

updateAnalysisModel()

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

1.1.3 Member Data Documentation

area_class

```
double analysis_curves::area_class  [private]
```

Area of Nanotube.

dist_class

```
double analysis_curves::dist_class  [private]
```

Distance from Point Source (-1 if the background field is plane wave)

lambda_class

```
double analysis_curves::lambda_class  [private]
```

Wavelength.

m_analysis_model

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

curve model after refinement used for analysis

m_cad_model

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

initial NURBS CAD model

m_e

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

solution vector (electric field on analysis_model)

m_greville

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

Greville abscissae.

m_grevillePoints

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

Greville points.

sigma_class

```
complex<double> analysis_curves::sigma_class [private]
```

Conductivity.

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

- analysis_NURBSCurve.h
- analysis_NURBSCurve.cpp

1.2 BIE_gsl_f_params Struct Reference

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

Public Attributes

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

1.2.1 Member Data Documentation

c

```
SplineCurve BIE_gsl_f_params::c
```

e

```
SplineCurve BIE_gsl_f_params::e
```

isReal

```
bool BIE_gsl_f_params::isReal
```

q

```
Point BIE_gsl_f_params::q
```

w

```
double BIE_gsl_f_params::w
```

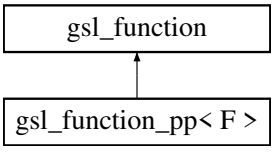
The documentation for this struct was generated from the following file:

- analysis_NURBSCurve.h

1.3 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

1.3.1 Constructor & Destructor Documentation

gsl_function_pp()

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

1.3.2 Member Function Documentation

invoke()

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

operator gsl_function *()

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

1.3.3 Member Data Documentation

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

1.4 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

1.4.1 Constructor & Destructor Documentation

IntegrationWorkspace()

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

~IntegrationWorkspace()

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

1.4.2 Member Function Documentation

operator gsl_integration_workspace *()

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

1.4.3 Member Data Documentation

wsp

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

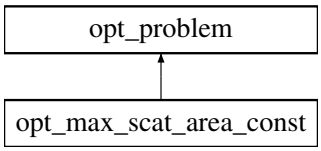
The documentation for this class was generated from the following file:

- integration.h

1.5 opt_max_scat_area_const Class Reference

`#include <opt_problems.h>`

Inheritance diagram for opt_max_scat_area_const:



Public Member Functions

- opt_max_scat_area_const (unsigned int problem_size=0)
- ~opt_max_scat_area_const ()
- opt_max_scat_area_const (unsigned int problem_size, double area)
- opt_max_scat_area_const (unsigned int problem_size, double area, opt_problem_data d)
- opt_max_scat_area_const (const opt_max_scat_area_const &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

1.5.1 Constructor & Destructor Documentation

opt_max_scat_area_const() [1/4]

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

~opt_max_scat_area_const()

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

opt_max_scat_area_const() [2/4]

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

opt_max_scat_area_const() [3/4]

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

opt_max_scat_area_const() [4/4]

```
opt_max_scat_area_const::opt_max_scat_area_const (
            const opt_max_scat_area_const & o )
```

1.5.2 Member Function Documentation

batch_fitness()

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

Reimplemented from opt_problem.

fitness()

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

Reimplemented from opt_problem.

get_area_constraint()

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

get_nic()

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

Reimplemented from opt_problem.

set_area_constraint()

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

set_problem_data()

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

1.5.3 Member Data Documentation

m_area

```
double opt_max_scat_area_const::m_area  [private]
```

m_data

```
opt_problem_data opt_max_scat_area_const::m_data  [private]
```

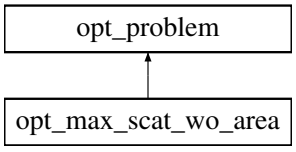
The documentation for this class was generated from the following files:

- opt_problems.h
- opt_problems.cpp

1.6 opt_max_scat_wo_area Class Reference

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

Inheritance diagram for opt_max_scat_wo_area:



Public Member Functions

- opt_max_scat_wo_area (unsigned int problem_size=0)
- ~opt_max_scat_wo_area ()
- opt_max_scat_wo_area (unsigned int problem_size, opt_problem_data d)
- opt_max_scat_wo_area (const opt_max_scat_wo_area &o)
- void set_problem_data (opt_problem_data d)
- vector_double fitness (const vector_double &v) const
- vector_double batch_fitness (const vector_double &v) 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
- 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)

Private Attributes

- 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

1.6.1 Constructor & Destructor Documentation

opt_max_scat_wo_area() [1/3]

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

~opt_max_scat_wo_area()

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

opt_max_scat_wo_area() [2/3]

```
opt_max_scat_wo_area::opt_max_scat_wo_area (
            unsigned int problem_size,
            opt_problem_data d )  [inline]
```

opt_max_scat_wo_area() [3/3]

```
opt_max_scat_wo_area::opt_max_scat_wo_area (
            const opt_max_scat_wo_area & o )
```

1.6.2 Member Function Documentation

batch_fitness()

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

Reimplemented from opt_problem.

fitness()

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

Reimplemented from opt_problem.

set_problem_data()

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

1.6.3 Member Data Documentation

m_data

```
opt_problem_data opt_max_scat_wo_area::m_data  [private]
```

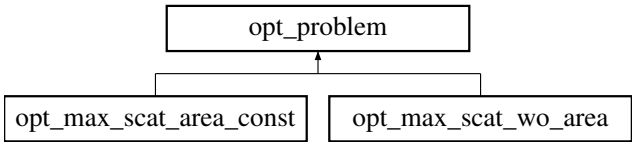
The documentation for this class was generated from the following files:

- opt_problems.h
- opt_problems.cpp

1.7 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

1.7.1 Constructor & Destructor Documentation

opt_problem()

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

~opt_problem()

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

1.7.2 Member Function Documentation

batch_fitness()

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

Reimplemented in opt_max_scat_area_const, and opt_max_scat_wo_area.

fitness()

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

Reimplemented in opt_max_scat_area_const, and opt_max_scat_wo_area.

get_bounds()

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

get_nec()

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

get_nic()

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

Reimplemented in opt_max_scat_area_const.

set_bounds()

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

set_problem_size()

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

1.7.3 Member Data Documentation

m_lb

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

m_problem_size

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

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

1.8 opt_problem_data Struct Reference

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

Public Attributes

- double wavelength
- double area0
- double waveangle
- complex< double > sigma
- double dist
- unsigned int refinement_level

1.8.1 Member Data Documentation

area0

```
double opt_problem_data::area0
```

dist

```
double opt_problem_data::dist
```

refinement_level

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

sigma

```
complex<double> opt_problem_data::sigma
```

waveangle

```
double opt_problem_data::waveangle
```

wavelength

```
double opt_problem_data::wavelength
```

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

- opt_problems.h

2 File Documentation

2.1 angle_sens.cpp File Reference

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

Functions

- int main (int argc, char ∗argv[ ])

2.1.1 Function Documentation

main()

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

2.2 comp_time.cpp File Reference

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

Functions

- int main (int argc, char ∗argv[ ])

2.2.1 Function Documentation

main()

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

2.3 distSens.cpp File Reference

```
#include <mpi.h>
#include <iostream>
#include "analysis_NURBSCurve.h"
#include <random>
#include <stdlib.h>
#include <tbb/parallel_for.h>
```

Functions

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

2.3.1 Function Documentation

main()

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

2.4 numSpikes.cpp File Reference

```
#include <mpi.h>
#include <iostream>
#include "analysis_NURBSCurve.h"
#include <random>
#include <stdlib.h>
#include <tbb/parallel_for.h>
```

Functions

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

2.4.1 Function Documentation

main()

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

2.5 openMPI_par_dist.cpp File Reference

```
#include <mpi.h>
#include <iostream>
#include "analysis_NURBSCurve.h"
#include "tbb/tick_count.h"
#include <random>
#include <stdlib.h>
#include <tbb/parallel_for.h>
```

Functions

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

2.5.1 Function Documentation

broadcastCurveData()

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

main()

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

2.6 scratch.cpp File Reference

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

Functions

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

2.6.1 Function Documentation

main()

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

2.7 test_mesh.cpp File Reference

```
#include <iostream>
#include <random>
#include "analysis_NURBSCurve.h"
#include <stdlib.h>
#include "tbb/tick_count.h"
```

Functions

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

2.7.1 Function Documentation

main()

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

2.8 test_opt.cpp File Reference

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

Functions

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

2.8.1 Function Documentation

main()

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

2.9 test_opt_old.cpp File Reference

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

Functions

- int main (int argc, char ∗argv[ ])

2.9.1 Function Documentation

main()

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

2.10 test_pagmo.cpp File Reference

```
#include <iostream>
#include <pagmo/algorithm.hpp>
#include <pagmo/algorithms/sade.hpp>
#include <pagmo/archipelago.hpp>
#include <pagmo/problem.hpp>
#include <pagmo/problems/schwefel.hpp>
```

Functions

- string vector2str (vector< double > v)
- int main ()

2.10.1 Function Documentation

main()

```
int main ( )
```

vector2str()

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

2.11 test_splineCurve.cpp File Reference

```
#include <mpi.h>
#include <iostream>
#include "analysis_NURBSCurve.h"
#include "tbb/tick_count.h"
#include <random>
```

Functions

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

2.11.1 Function Documentation

broadcastCurveData()

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

main()

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

2.12 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)

2.12.1 Macro Definition Documentation

ANALYSIS_NURBSCURVE_H

```
#define ANALYSIS_NURBSCURVE_H
```

GOTOOLS_BASIS_VERSION

```
#define GOTOOLS_BASIS_VERSION
```

2.12.2 Function Documentation

BIE_gsl_f()

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

2.13 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 class analysis_curves
00039 {
00040     private:
00041     SplineCurve ComputeSplineCurve(vector<double> params, double rv, vector<double> knots, unsigned
      int k, unsigned int n);
00042     SplineCurve ComputeSplineCurve_dynamic(vector<double> params, double rv, vector<double> knots,
      unsigned int k, unsigned int n);
00044     SplineCurve ComputeSymSplineCurve(vector<double> params, double rv, unsigned int k);
00045     void updateAnalysisModel() { m_analysis_model = m_cad_model; CalculateGreville(); }
00046     void shiftGreville(valarray<double> t, unsigned int curve_index);
00048     vector<SplineCurve> m_cad_model;
00050     vector<SplineCurve> m_analysis_model;
00052     vector<SplineCurve> m_e;
00054     vector<valarray<double> > m_greville;
00056     vector<vector<Point> > m_grevillePoints;
00058     double lambda_class;
00060     complex<double> sigma_class;
00062     double area_class;
00064     double dist_class;
00065
00066 public:
00068     analysis_curves() { }
00070     ~analysis_curves() { }
00072     analysis_curves(double wavelength, complex<double> sigma, double area, double dist);
00074     unsigned int nCurves() const{return m_cad_model.size();}
00076     vector<unsigned int> DoFs(bool CAD = false) const;
00078     const SplineCurve& operator[](unsigned int i) const;
00080     const SplineCurve& getAnalysisCurve(unsigned int i) const;
00082     const SplineCurve& getSolution(unsigned int i) const;
00084     void addCurve(SplineCurve& curve) { m_cad_model.push_back(curve); }
00086     void addAnalysisCurve(SplineCurve& curve) { m_analysis_model.push_back(curve); }
00088     void addGrevilleAbscissae(double* vals, int size) {m_greville.push_back(valarray<double>(vals,
      size));}
00090     double GetGrevilleAbscissa(unsigned int i, unsigned int curve_index = 0);
00092     bool isCurveClosed(double tolerance=1e-6, unsigned int curve_index = 0);
00094     double CurveArea(unsigned int curve_index = 0);
00096     double CurveArea(const SplineCurve& curve);
00098     double CurveLength(unsigned int curve_index = 0);
00100     double CurveArcLength(double start_param = 0., double end_param = 1., unsigned int curve_index =
      0);
00102     Point CurveCentroid(const SplineCurve& curve);
00104     void Circle(unsigned int ns=3, double r = 1, vector<double> v = { 0,0 }, unsigned int curve_index
      = 0);
00106     void CurveFromParameters_pm1(vector<double> parameters, double r_max, unsigned int curve_index=0);
00108     void CurveFromParameters_pm2(vector<double> parameters, double r_max, double area0, unsigned int
      curve_index = 0);
```

```
00110     void CurveFromParameters_dynamic_pm1(vector<double> parameters, double r_max, unsigned int
      curve_index=0);
00112     void CurveFromParameters_dynamic_pm2(vector<double> parameters, double r_max, double area0,
      unsigned int curve_index=0);
00115     double CurveFromParameters_pm3(vector<double> parameters, double r_max, double area0, int&
      error_flag, unsigned int curve_index = 0);
00118     void RefineCurve(unsigned int knots_per_span, unsigned int curve_index = 0);
00120     double ComputeBasisValue(double param, unsigned int b_id, unsigned int curve_index = 0);
00122     valarray<valarray<double> > ComputeBasisValues(valarray<double> params, unsigned int curve_index =
      0);
00124     vector<vector<Point> > ComputeCurvePoints(valarray<double> params, unsigned int derivs = 0,
      unsigned int curve_index = 0);
00126     valarray<complex<double> > ComputeSolutionValues(valarray<double> params, unsigned int curve_index
      = 0);
00128     void ComputeFieldOnSingleCurve(double wavelength, complex<double> sigma, double L = -1., double
      angle = 0, unsigned int curve_index = 0, double step = 1e-4);
00130     void ComputeFieldOnSingleCurve_TBB(double wavelength, complex<double> sigma, double L = -1.,
      double angle = 0, unsigned int curve_index = 0, double step = 1e-4);
00132     void ComputeFieldOnSingleCurve_GSL(double wavelength, complex<double> sigma, double L = -1.,
      double angle = 0, unsigned int curve_index = 0);
00134     void ComputeFieldOnSingleCurve_MPI(double wavelength, complex<double> sigma, double L = -1.,
      double angle = 0, unsigned int curve_index = 0, double step = 1e-4);
00136     bool read_iges_file(const char* filename);
00138     bool save_analysis_model(const char* filename, bool IGES = true);
00140     string PrintSolution(unsigned int curve_index = 0);
00142     int memory_size();
00144     double GetGreville(unsigned int i /*curve*/, unsigned int j /*Greville Abscissa index*/) { return
      m_greville[i][j];}
00146     string PrintCurveInfo(unsigned int i, bool analysis = false);
00148     void CalculateGreville();
00150     double CalculateQuantityOnMesh(double wavelength, double angle, complex<double> sigma, unsigned
      int curve_index=0, double step = 1e-4, string filename="");
00151     double CalculateQuantityOnMesh_GSL(double wavelength, double angle, complex<double> sigma,
      unsigned int curve_index = 0, double tolerance = 1e-4, string filename = "");
00153     double Scattering_analytic(double r, complex<double> sigma, double lambda, double L);
00155     double Scattering_numeric(complex<double> sigma, double wavelength, unsigned int curve_index=0);
00157     double ComputeScattering(double wavelength, double angle, complex<double> sigma, unsigned int
      curve_index = 0);
00158
00159 };
00160 #endif
```

2.14 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)

2.14.1 Macro Definition Documentation

INTEGRATION_H

```
#define INTEGRATION_H
```

2.14.2 Function Documentation

make_gsl_function()

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

2.15 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_cquad {
00007 //  gsl_integration_cquad_workspace* wsp;
00008 //public:
00009 //  IntegrationWorkspace_cquad(const size_t n = 1000) :
00010 //      wsp(gsl_integration_cquad_workspace_alloc(n)) {}
00011 //  ~IntegrationWorkspace_cquad() { gsl_integration_cquad_workspace_free(wsp); }
00012 //
00013 //  operator gsl_integration_cquad_workspace* () { return wsp; }
00014 //};
00015
00016 class IntegrationWorkspace {
00017     gsl_integration_workspace* wsp;
00018
00019 public:
00020     IntegrationWorkspace(const size_t n = 1000) :
00021         wsp(gsl_integration_workspace_alloc(n)) {}
00022     ~IntegrationWorkspace() { gsl_integration_workspace_free(wsp); }
00023
00024     operator gsl_integration_workspace* () { return wsp; }
00025
00026 };
00027 // Build gsl_function from lambda
00028 template <typename F>
00029 class gsl_function_pp : public gsl_function {
00030     const F func;
00031     static double invoke(double x, void* params) {
00032         return static_cast<gsl_function_pp*>(params)->func(x);
00033     }
00034 public:
00035     gsl_function_pp(const F& f) : func(f) {
00036         function = &gsl_function_pp::invoke; //inherited from gsl_function
00037         params = this;                       //inherited from gsl_function
00038     }
00039     operator gsl_function* () { return this; }
00040 };
00041
00042 // Helper function for template construction
00043 template <typename F>
00044 gsl_function_pp<F> make_gsl_function(const F& func) {
00045     return gsl_function_pp<F>(func);
00046 }
00047
00048 #endif
```

2.16 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_scat_area_const
- class opt_max_scat_wo_area

Macros

- #define OPT_PROBLEMS_H

Functions

- string vector2str (vector< double > v)

2.16.1 Macro Definition Documentation

OPT_PROBLEMS_H

```
#define OPT_PROBLEMS_H
```

2.16.2 Function Documentation

vector2str()

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

2.17 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
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 area0;
00023     double waveangle;
00024     complex<double> sigma;
00025     double dist;
00026     unsigned int refinement_level;
00027 };
00028
```

```
00029 //General optimization problem with a dummy objective function and side constraints
00030 class opt_problem
00031 {
00032 public:
00033     opt_problem(unsigned int problem_size = 0) : m_problem_size(problem_size) {}
00034     ~opt_problem() {}
00035     virtual vector_double fitness(const vector_double& v) const { return { 0 }; }
00036     virtual vector_double batch_fitness(const vector_double& v) const { return { 0 }; }
00037     virtual vector_double::size_type get_nec() const { return 0; }
00038     virtual vector_double::size_type get_nic() const { return 0; }
00039     pair<vector_double, vector_double> get_bounds() const;
00040     void set_bounds(vector<double> lb, vector<double> ub);
00041     void set_problem_size(unsigned int size) { m_problem_size = size; m_lb.resize(0); m_ub.resize(0);
      }
00042 protected:
00043     unsigned int m_problem_size;
00044     vector<double> m_lb;
00045     vector<double> m_ub;
00046 };
00047
00048 //Single nanotube optimization problem target maximization of scattering subject to an area constraint
00049 class opt_max_scat_area_const : public opt_problem
00050 {
00051 public:
00052     opt_max_scat_area_const(unsigned int problem_size = 0) : opt_problem(problem_size), m_area(0) {}
00053     ~opt_max_scat_area_const() {}
00054     opt_max_scat_area_const(unsigned int problem_size, double area) : opt_problem(problem_size),
      m_area(fabs(area)) {}
00055     opt_max_scat_area_const(unsigned int problem_size, double area, opt_problem_data d) :
      opt_problem(problem_size), m_area(fabs(area)), m_data(d) {}
00056     opt_max_scat_area_const(const opt_max_scat_area_const& o);
00057     void set_problem_data(opt_problem_data d) { m_data = { d.wavelength, d.area0, d.waveangle,
      d.sigma, d.dist, d.refinement_level }; }
00058     void set_area_constraint(double area) { m_area = fabs(area); }
00059     double get_area_constraint() { return m_area; }
00060     vector_double fitness(const vector_double& v) const;
00061     vector_double batch_fitness(const vector_double& v) const;
00062     vector_double::size_type get_nic() const { return 1; }
00063 private:
00064     double m_area;
00065     opt_problem_data m_data;
00066 };
00067
00068 class opt_max_scat_wo_area : public opt_problem
00069 {
00070 public:
00071     opt_max_scat_wo_area(unsigned int problem_size = 0) : opt_problem(problem_size){}
00072     ~opt_max_scat_wo_area() {}
00073     opt_max_scat_wo_area(unsigned int problem_size, opt_problem_data d) : opt_problem(problem_size),
      m_data(d) {}
00074     opt_max_scat_wo_area(const opt_max_scat_wo_area& o);
00075     void set_problem_data(opt_problem_data d) { m_data = { d.wavelength, d.area0, d.waveangle,
      d.sigma, d.dist, d.refinement_level }; }
00076     vector_double fitness(const vector_double& v) const;
00077     vector_double batch_fitness(const vector_double& v) const;
00078 private:
00079     opt_problem_data m_data;
00080 };
00081
00082 string vector2str(vector<double> v) {
00083     ostringstream ss;
00084     ss « "(";
00085     unsigned int i;
00086     for (i = 0; i < v.size() - 1; i++)
00087         ss « v[i] « ",";
00088     ss « v[i] « ")";
00089     return ss.str();
00090 }
00091 #endif
```

2.18 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 angle=0)
- complex< double > E_pnt_src_back (Point p, double wavelength, double dist, double angle=0)
- string point2str (Point a, bool addCR=false)
- double TriangleArea (Point a, Point b, Point c)
- complex< double > Green (Point p0, Point p, double wavelength)
- valarray< complex< double > > Green (Point p0, vector< Point > p, double wavelength)
- complex< double > EzAtCNT (double phi, complex< double > cn0, double k0a, double k0L, unsigned int bound=20)

2.18.1 Macro Definition Documentation

SPECIAL_FUNCTIONS_H

```
#define SPECIAL_FUNCTIONS_H
```

2.18.2 Function Documentation

E_plane_back()

```
complex< double > E_plane_back (
            Point a,
            double wavelength,
            double angle = 0 )
```

E_pnt_src_back()

```
complex< double > E_pnt_src_back (
            Point p,
            double wavelength,
            double dist,
            double angle = 0 )
```

EzAtCNT()

```
complex< double > EzAtCNT (
            double phi,
            complex< double > cn0,
            double k0a,
            double k0L,
            unsigned int bound = 20 )
```

Green() [1/2]

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

Green() [2/2]

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

i_unit()

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

NormOfPointsInVector()

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

point2str()

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

toComplexArray()

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

toGSLcomplex()

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

transposeArray()

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

TriangleArea()

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

2.19 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 angle = 0);
00020 complex<double> E_pnt_src_back(Point p, double wavelength, double dist, double angle = 0);
00021 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(); }
00022 double TriangleArea(Point a, Point b, Point c);
00023 complex<double>  Green(Point p0, Point p, double wavelength);
00024 valarray<complex<double> > Green(Point p0, vector<Point> p, double wavelength);
00025 complex<double> EzAtCNT(double phi, complex<double> cn0, double k0a, double k0L, unsigned int bound =
      20);
00026
00027 #endif
```

2.20 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>
```

2.21 opt_problems.cpp File Reference

```
#include "opt_problems.h"
#include "analysis_NURBSCurve.h"
#include <tbb/parallel_for.h>
```

2.22 special_functions.cpp File Reference

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

Functions

- complex< double > E_plane_back (Point p, double l, double a)
- complex< double > E_pnt_src_back (Point p, double l, double dist, 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)
- complex< double > EzAtCNT (double phi, complex< double > cn0, double k0a, double k0L, unsigned int bound)

2.22.1 Function Documentation

E_plane_back()

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

E_pnt_src_back()

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

EzAtCNT()

```
complex< double > EzAtCNT (
            double phi,
            complex< double > cn0,
            double k0a,
            double k0L,
            unsigned int bound )
```

Green() [1/2]

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

Green() [2/2]

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

NormOfPointsInVector()

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

toComplexArray()

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

transposeArray()

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

TriangleArea()

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

Index

~IntegrationWorkspace
    IntegrationWorkspace, 64
~analysis_curves
    analysis_curves, 51
~opt_max_scat_area_const
    opt_max_scat_area_const, 66
~opt_max_scat_wo_area
    opt_max_scat_wo_area, 68
~opt_problem
    opt_problem, 70

addAnalysisCurve
    analysis_curves, 52
addCurve
    analysis_curves, 52
addGrevilleAbscissae
    analysis_curves, 52
analysis_curves, 49
    ~analysis_curves, 51
    addAnalysisCurve, 52
    addCurve, 52
    addGrevilleAbscissae, 52
    analysis_curves, 51, 52
    area_class, 60
    CalculateGreville, 52
    CalculateQuantityOnMesh, 52
    CalculateQuantityOnMesh_GSL, 53
    Circle, 53
    ComputeBasisValue, 53
    ComputeBasisValues, 53
    ComputeCurvePoints, 53
    ComputeFieldOnSingleCurve, 54
    ComputeFieldOnSingleCurve_GSL, 54
    ComputeFieldOnSingleCurve_MPI, 54
    ComputeFieldOnSingleCurve_TBB, 54
    ComputeScattering, 54
    ComputeSolutionValues, 55
    ComputeSplineCurve, 55
    ComputeSplineCurve_dynamic, 55
    ComputeSymSplineCurve, 55
    CurveArcLength, 55
    CurveArea, 56
    CurveCentroid, 56
    CurveFromParameters_dynamic_pm1, 56
    CurveFromParameters_dynamic_pm2, 56
    CurveFromParameters_pm1, 56
    CurveFromParameters_pm2, 57
    CurveFromParameters_pm3, 57
    CurveLength, 57
    dist_class, 60
    DoFs, 57
    getAnalysisCurve, 57
    GetGreville, 58
    GetGrevilleAbscissa, 58
    getSolution, 58
    isCurveClosed, 58
    lambda_class, 60
    m_analysis_model, 60
    m_cad_model, 61
    m_e, 61
    m_greville, 61
    m_grevillePoints, 61
    memory_size, 58
    nCurves, 58
    operator[], 58
    PrintCurveInfo, 59
    PrintSolution, 59
    read_iges_file, 59
    RefineCurve, 59
    save_analysis_model, 59
    Scattering_analytic, 59
    Scattering_numeric, 60
    shiftGreville, 60
    sigma_class, 61
    updateAnalysisModel, 60
analysis_NURBSCurve.cpp, 87
analysis_NURBSCurve.h, 79, 80
    ANALYSIS_NURBSCURVE_H, 79
    BIE_gsl_f, 79
    GOTOOLS_BASIS_VERSION, 79
ANALYSIS_NURBSCURVE_H
    analysis_NURBSCurve.h, 79
angle_sens.cpp, 73
    main, 73
area0
    opt_problem_data, 72
area_class
    analysis_curves, 60

batch_fitness
    opt_max_scat_area_const, 66
    opt_max_scat_wo_area, 69
    opt_problem, 70
BIE_gsl_f
    analysis_NURBSCurve.h, 79
BIE_gsl_f_params, 61
    c, 62
    e, 62
    isReal, 62
    q, 62
    w, 62
broadcastCurveData
    openMPI_par_dist.cpp, 75
    test_splineCurve.cpp, 78

c
    BIE_gsl_f_params, 62
CalculateGreville
    analysis_curves, 52
CalculateQuantityOnMesh
    analysis_curves, 52

CalculateQuantityOnMesh_GSL
analysis_curves, 53
Circle
analysis_curves, 53
comp_time.cpp, 73
main, 73
ComputeBasisValue
analysis_curves, 53
ComputeBasisValues
analysis_curves, 53
ComputeCurvePoints
analysis_curves, 53
ComputeFieldOnSingleCurve
analysis_curves, 54
ComputeFieldOnSingleCurve_GSL
analysis_curves, 54
ComputeFieldOnSingleCurve_MPI
analysis_curves, 54
ComputeFieldOnSingleCurve_TBB
analysis_curves, 54
ComputeScattering
analysis_curves, 54
ComputeSolutionValues
analysis_curves, 55
ComputeSplineCurve
analysis_curves, 55
ComputeSplineCurve_dynamic
analysis_curves, 55
ComputeSymSplineCurve
analysis_curves, 55
CurveArcLength
analysis_curves, 55
CurveArea
analysis_curves, 56
CurveCentroid
analysis_curves, 56
CurveFromParameters_dynamic_pm1
analysis_curves, 56
CurveFromParameters_dynamic_pm2
analysis_curves, 56
CurveFromParameters_pm1
analysis_curves, 56
CurveFromParameters_pm2
analysis_curves, 57
CurveFromParameters_pm3
analysis_curves, 57
CurveLength
analysis_curves, 57

dist
opt_problem_data, 72
dist_class
analysis_curves, 60
distSens.cpp, 74
main, 74
DoFs
analysis_curves, 57

e
BIE_gsl_f_params, 62
E_plane_back
special_functions.cpp, 88
special_functions.h, 85
E_pnt_src_back
special_functions.cpp, 88
special_functions.h, 85
EzAtCNT
special_functions.cpp, 88
special_functions.h, 85

fitness
opt_max_scat_area_const, 66
opt_max_scat_wo_area, 69
opt_problem, 70
func
gsl_function_pp< F >, 63

get_area_constraint
opt_max_scat_area_const, 66
get_bounds
opt_problem, 70
get_nec
opt_problem, 70
get_nic
opt_max_scat_area_const, 67
opt_problem, 71
getAnalysisCurve
analysis_curves, 57
GetGreville
analysis_curves, 58
GetGrevilleAbscissa
analysis_curves, 58
getSolution
analysis_curves, 58
GOTOOLS_BASIS_VERSION
analysis_NURBSCurve.h, 79
Green
special_functions.cpp, 88
special_functions.h, 85, 86
gsl_function_pp
gsl_function_pp< F >, 63
gsl_function_pp< F >, 62
func, 63
gsl_function_pp, 63
invoke, 63
operator gsl_function *, 63

i_unit
special_functions.h, 86
integration.h, 81, 82
INTEGRATION_H, 81
make_gsl_function, 82
INTEGRATION_H
integration.h, 81
IntegrationWorkspace, 64
~IntegrationWorkspace, 64
IntegrationWorkspace, 64
operator gsl_integration_workspace *, 64

wsp, 64
invoke
gsl_function_pp< F >, 63
isCurveClosed
analysis_curves, 58
isReal
BIE_gsl_f_params, 62

lambda_class
analysis_curves, 60

m_analysis_model
analysis_curves, 60
m_area
opt_max_scat_area_const, 67
m_cad_model
analysis_curves, 61
m_data
opt_max_scat_area_const, 67
opt_max_scat_wo_area, 69
m_e
analysis_curves, 61
m_greville
analysis_curves, 61
m_grevillePoints
analysis_curves, 61
m_lb
opt_problem, 71
m_problem_size
opt_problem, 71
m_ub
opt_problem, 71
main
angle_sens.cpp, 73
comp_time.cpp, 73
distSens.cpp, 74
numSpikes.cpp, 74
openMPI_par_dist.cpp, 75
scratch.cpp, 76
test_mesh.cpp, 76
test_opt.cpp, 77
test_opt_old.cpp, 77
test_pagmo.cpp, 78
test_splineCurve.cpp, 78
make_gsl_function
integration.h, 82
memory_size
analysis_curves, 58

nCurves
analysis_curves, 58
NormOfPointsInVector
special_functions.cpp, 89
special_functions.h, 86
numSpikes.cpp, 74
main, 74

openMPI_par_dist.cpp, 75
broadcastCurveData, 75
main, 75
operator gsl_function *
gsl_function_pp< F >, 63
operator gsl_integration_workspace *
IntegrationWorkspace, 64
operator[]
analysis_curves, 58
opt_max_scat_area_const, 65
~opt_max_scat_area_const, 66
batch_fitness, 66
fitness, 66
get_area_constraint, 66
get_nic, 67
m_area, 67
m_data, 67
opt_max_scat_area_const, 66
set_area_constraint, 67
set_problem_data, 67
opt_max_scat_wo_area, 67
~opt_max_scat_wo_area, 68
batch_fitness, 69
fitness, 69
m_data, 69
opt_max_scat_wo_area, 68
set_problem_data, 69
opt_problem, 69
~opt_problem, 70
batch_fitness, 70
fitness, 70
get_bounds, 70
get_nec, 70
get_nic, 71
m_lb, 71
m_problem_size, 71
m_ub, 71
opt_problem, 70
set_bounds, 71
set_problem_size, 71
opt_problem_data, 71
area0, 72
dist, 72
refinement_level, 72
sigma, 72
waveangle, 72
wavelength, 72
opt_problems.cpp, 87
opt_problems.h, 82, 83
OPT_PROBLEMS_H, 83
vector2str, 83
OPT_PROBLEMS_H
opt_problems.h, 83

point2str
special_functions.h, 86
PrintCurveInfo
analysis_curves, 59
PrintSolution
analysis_curves, 59

q
BIE_gsl_f_params, 62

read_iges_file
analysis_curves, 59
RefineCurve
analysis_curves, 59
refinement_level
opt_problem_data, 72

save_analysis_model
analysis_curves, 59
Scattering_analytic
analysis_curves, 59
Scattering_numeric
analysis_curves, 60
scratch.cpp, 75
main, 76
set_area_constraint
opt_max_scat_area_const, 67
set_bounds
opt_problem, 71
set_problem_data
opt_max_scat_area_const, 67
opt_max_scat_wo_area, 69
set_problem_size
opt_problem, 71
shiftGreville
analysis_curves, 60
sigma
opt_problem_data, 72
sigma_class
analysis_curves, 61
special_functions.cpp, 88
E_plane_back, 88
E_pnt_src_back, 88
EzAtCNT, 88
Green, 88
NormOfPointsInVector, 89
toComplexArray, 89
transposeArray, 89
TriangleArea, 89
special_functions.h, 84, 87
E_plane_back, 85
E_pnt_src_back, 85
EzAtCNT, 85
Green, 85, 86
i_unit, 86
NormOfPointsInVector, 86
point2str, 86
SPECIAL_FUNCTIONS_H, 85
toComplexArray, 86
toGSLcomplex, 86
transposeArray, 86
TriangleArea, 86
SPECIAL_FUNCTIONS_H
special_functions.h, 85

test_mesh.cpp, 76
main, 76
test_opt.cpp, 76
main, 77
test_opt_old.cpp, 77
main, 77
test_pagmo.cpp, 77
main, 78
vector2str, 78
test_splineCurve.cpp, 78
broadcastCurveData, 78
main, 78
toComplexArray
special_functions.cpp, 89
special_functions.h, 86
toGSLcomplex
special_functions.h, 86
transposeArray
special_functions.cpp, 89
special_functions.h, 86
TriangleArea
special_functions.cpp, 89
special_functions.h, 86

updateAnalysisModel
analysis_curves, 60

vector2str
opt_problems.h, 83
test_pagmo.cpp, 78

w
BIE_gsl_f_params, 62
waveangle
opt_problem_data, 72
wavelength
opt_problem_data, 72
wsp
IntegrationWorkspace, 64

Appendix B

Source Code MatLAB Code

GaOptimization_MatLAB

Generated by Doxygen 1.10.0

1 File Documentation **100**

1.1 areaconstraint.m File Reference . . . 100
1.1.1 Variable Documentation . . . 100
1.2 areagreen.m File Reference . . . 101
1.2.1 Variable Documentation . . . 101
1.3 changeAngle.m File Reference . . . 101
1.3.1 Function Documentation . . . 102
1.3.2 Variable Documentation . . . 103
1.4 changeAngleGeom.m File Reference . . . 105
1.4.1 Function Documentation . . . 105
1.4.2 Variable Documentation . . . 105
1.5 changeAngleNewParam.m File Reference . . . 106
1.5.1 Function Documentation . . . 106
1.5.2 Variable Documentation . . . 106
1.6 changeDistance.m File Reference . . . 107
1.6.1 Variable Documentation . . . 107
1.7 convertNRB.m File Reference . . . 109
1.7.1 Variable Documentation . . . 109
1.8 countSteps.m File Reference . . . 109
1.8.1 Variable Documentation . . . 109
1.9 create_circle.m File Reference . . . 110
1.10 curve_area.m File Reference . . . 110
1.10.1 Variable Documentation . . . 110
1.11 curve_centroid.m File Reference . . . 111
1.11.1 Variable Documentation . . . 111
1.12 dataFunc.m File Reference . . . 111
1.12.1 Variable Documentation . . . 112
1.13 dataKernelNumeric.m File Reference . . . 112
1.13.1 Variable Documentation . . . 113
1.14 E_back.m File Reference . . . 113
1.14.1 Variable Documentation . . . 113
1.15 EzAtCNT.m File Reference . . . 114
1.15.1 Variable Documentation . . . 114
1.16 freeform_geom.m File Reference . . . 115
1.16.1 Function Documentation . . . 115
1.16.2 Variable Documentation . . . 115
1.17 freeform_geom_withArea.m File Reference . . . 116
1.17.1 Function Documentation . . . 117
1.17.2 Variable Documentation . . . 117
1.18 freeform_geom_wo_pers.m File Reference . . . 118
1.18.1 Function Documentation . . . 119
1.18.2 Variable Documentation . . . 119

1.19 freeform_x_sym1_withArea.m File Reference . . . 120
1.19.1 Function Documentation . . . 120
1.19.2 Variable Documentation . . . 121
1.20 freeform_x_sym_withArea.m File Reference . . . 123
1.20.1 Function Documentation . . . 124
1.20.2 Variable Documentation . . . 124
1.21 Green.m File Reference . . . 126
1.21.1 Variable Documentation . . . 126
1.22 halfSpike.m File Reference . . . 127
1.22.1 Function Documentation . . . 127
1.22.2 Variable Documentation . . . 128
1.23 igabem_circle_refinement.m File Reference . . . 128
1.23.1 Function Documentation . . . 129
1.23.2 Variable Documentation . . . 130
1.24 igabem_single_cnt.m File Reference . . . 131
1.24.1 Function Documentation . . . 132
1.24.2 Variable Documentation . . . 133
1.25 IncreaseDoFsInModel.m File Reference . . . 136
1.25.1 Function Documentation . . . 137
1.25.2 Variable Documentation . . . 137
1.26 info_curve.m File Reference . . . 139
1.26.1 Function Documentation . . . 139
1.26.2 Variable Documentation . . . 140
1.27 insertKnots.m File Reference . . . 142
1.27.1 Variable Documentation . . . 142
1.28 insertOneKnot.m File Reference . . . 143
1.28.1 Function Documentation . . . 143
1.28.2 Variable Documentation . . . 144
1.29 kernel_analytic.m File Reference . . . 145
1.29.1 Variable Documentation . . . 145
1.30 kernel_numeric.m File Reference . . . 146
1.30.1 Function Documentation . . . 146
1.30.2 Variable Documentation . . . 146
1.31 new_freeform.m File Reference . . . 147
1.31.1 Function Documentation . . . 147
1.31.2 Variable Documentation . . . 148
1.32 new_freeform_unclamped.m File Reference . . . 149
1.32.1 Function Documentation . . . 149
1.32.2 Variable Documentation . . . 150
1.33 newIncrDoF.m File Reference . . . 151
1.33.1 Function Documentation . . . 151
1.33.2 Variable Documentation . . . 152

1.34 oldToNew.m File Reference 153
1.34.1 Function Documentation 154
1.34.2 Variable Documentation 154
1.35 p_sctr.m File Reference 155
1.35.1 Function Documentation 155
1.35.2 Variable Documentation 156
1.36 test_calc.m File Reference 158
1.36.1 Function Documentation 158
1.36.2 Variable Documentation 159
1.37 test_calc_multiple.m File Reference 161
1.37.1 Function Documentation 162
1.37.2 Variable Documentation 164
1.38 test_freeform.m File Reference 166
1.38.1 Function Documentation 166
1.38.2 Variable Documentation 166
1.39 test_newIncrDoF.m File Reference 167
1.39.1 Function Documentation 167
1.39.2 Variable Documentation 168
1.40 test_optimize.m File Reference 169
1.40.1 Function Documentation 170
1.40.2 Variable Documentation 171
1.41 test_optimize2.m File Reference 173
1.41.1 Function Documentation 174
1.41.2 Variable Documentation 175

Index **179**

1 File Documentation

1.1 areaconstraint.m File Reference

Variables

- function [c, ceq]
- if nargin
- end x = reshape(x, [2,length(x)/2])
- curve = convertNRB(freeform_geom(x))
- in_area = areagreen(curve)
- c = abs(in_area-area0) - 1e-2
- ceq = [ ]

1.1.1 Variable Documentation

c

```
end c = abs(in_area-area0) - 1e-2
```

ceq

```
ceq = [ ]
```

curve

```
curve = convertNRB(freeform_geom(x))
```

function

```
end function
```

Initial value:

```
= areaconstraint(x, a0)
persistent area0
```

in_area

```
in_area = areagreen(curve)
```

nargin

```
elseif nargin
```

Initial value:

```
== 2
    area0 = a0
```

x

```
end x = reshape(x, [2,length(x)/2])
```

1.2 areagreen.m File Reference

Variables

- function [area]
- area = 0.5∗integral(@func,0,1)
- function y

1.2.1 Variable Documentation

area

```
area = 0.5*integral(@func,0,1)
```

function

```
function[area]
```

Initial value:

```
= areagreen(geom)
geom = convertNRB(geom)
```

y

```
General y
```

Initial value:

```
= func(x)
        p = fntlr(geom,2,x)
```

1.3 changeAngle.m File Reference

Functions

- id dataFunc ()
- freeform_geom ([ ], 8.9445)
- curve (i)
- kernel_numeric ([ ], lambda, e_numerical(i), curve(i))
- scat_numerical (i)
- plot (delAng, ratio)
- title ("Angle change vs $\frac{P_{sct, curve}}{P_{sct, circle}}$", Interpreter="latex") xlabel("Angle Shift in Degrees") ylabel("$\frac

Variables

- clear
- k0 = 2∗pi/lambda
- h0 = 120∗pi
- multiplier = k0/(16∗pi∗h0)∗abs(sigma∗h0)^2
- x0
- x1 = [0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01]
- x2 = 0.5 ∗ ones(1,30)
- geom = freeform_x_sym_withArea(x0, r_max, areaconstr)
- circle = create_circle(r,10,true)
- x11 = 0.01 ∗ ones(1,15)
- curve = freeform_geom_withArea(x0, r_max, areaconstr)r
- delAng = 0:5:360
- n = length(delAng)
- parfor i
- scat_analytic = multiplier ∗ integral(@kernel_analytic,0,2∗pi)
- ratio = scat_numerical/scat_analytic
- figure

1.3.1 Function Documentation

curve()

```
curve (
            i  )
```

dataFunc()

```
id dataFunc ( )  [virtual]
```

freeform_geom()

```
freeform_geom (
            8.  9445 )
```

kernel_numeric()

```
kernel_numeric (
            lambda ,
            e_numerical(i) ,
            curve(i)  )
```

plot()

```
plot (
            delAng ,
            ratio  )
```

scat_numerical()

```
scat_numerical (
             i  )
```

title()

```
title (
             "Angle change vs $\frac{P_{sct, curve}}{P_{sct, circle}}$" ,
             Interpreter  = "latex" )
```

1.3.2 Variable Documentation

circle

```
circle with arc segments and a radius of r circle = create_circle(r,10,true)
```

clear

```
clear
```

curve

```
curve = freeform_geom_withArea(x0, r_max, areaconstr)r
```

delAng

```
delAng = 0:5:360
```

figure

```
figure
```

geom

```
assume cubics geom = freeform_x_sym_withArea(x0, r_max, areaconstr)
```

h0

```
h0 = 120*pi
```

i

for i

Initial value:

```
= 1:n
    curve(i) = changeAngleGeom(geom, delAng(i))
```

k0

```
k0 = 2*pi/lambda
```

multiplier

```
multiplier = k0/(16*pi*h0)*abs(sigma*h0)^2
```

n

```
n = length(delAng)
```

ratio

```
ratio = scat_numerical/scat_analytic
```

scat_analytic

```
scat_analytic = multiplier * integral(@kernel_analytic,0,2*pi)
```

x0

```
x0
```

Initial value:

```
=
```

x1

```
x1 = [0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.↩
01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01, 0.01, 0.99, 0.01]
```

x11

```
x11 = 0.01 * ones(1,15)
```

x2

```
x2 = 0.5 * ones(1,30)
```

1.4 changeAngleGeom.m File Reference

Functions

- new_geom coefs (1,:)
- new_geom coefs (2,:)

Variables

- function new_geom
- th = atan2(geom.coefs(2,:),geom.coefs(1,:))

1.4.1 Function Documentation

coefs() [1/2]

```
new_geom coefs (
              1 ,
              :    )
```

coefs() [2/2]

```
new_geom coefs (
              2 ,
              :    )
```

1.4.2 Variable Documentation

new_geom

```
new_geom
```

Initial value:

```
= changeAngleGeom(geom, phi)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
phi = deg2rad(phi)
```

th

```
th = atan2(geom.coefs(2,:),geom.coefs(1,:))
```

1.5 changeAngleNewParam.m File Reference

Functions

- if isempty (params(params >1|params< 0)) new _params(1
- More free but wilder phi (1)
- new_params (2,:)
- new_params (2, 1)

Variables

- function new_params
- if n = length(params)
- for i
- end phi = phi + del

1.5.1 Function Documentation

isempty()

```
if isempty (
             params(params >1|params< 0)  )  [new]
```

new_params() [1/2]

```
new_params (
             2 ,
             1  )
```

new_params() [2/2]

```
new_params (
             2 ,
             :    )
```

phi()

```
More free but wilder phi (
             1  )
```

1.5.2 Variable Documentation

i

```
for i
```

Initial value:

```
= 2:n
        phi(i) = params(2,i)*(2*pi - phi(i-1)) + phi(i-1)
```

n

```
if n = length(params)
```

new_params

```
end new_params
```

Initial value:

```
= changeAngleNewParam(params, del)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

del = deg2rad(del)
```

phi

```
end For the more conservative model phi = phi + del
```

1.6 changeDistance.m File Reference

Variables

- clear addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox areaconstr = 2
- r =sqrt(areaconstr/pi)
- sigma = 0.001 + 0.001i
- lambda = 1
- L = 8:0.1:12
- n = length(L)
- r_max = 3∗r
- step = 1e-4
- k0 = 2∗pi/lambda
- h0 = 120∗pi
- multiplier = k0/(16∗pi∗h0)∗abs(sigma∗h0)^2
- circle = create_circle(r,10,true)
- parfor i

1.6.1 Variable Documentation

areaconstr

```
clear addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox
areaconstr = 2
```

circle

```
circle with arc segments and a radius of r circle = create_circle(r,10,true)
```

h0

```
h0 = 120*pi
```

i

```
parfor i
```

Initial value:

```
=1:n
    kernel_analytic([],r,sigma,lambda,L(i))
```

k0

```
k0 = 2*pi/lambda
```

L

```
L = 8:0.1:12
```

lambda

```
lambda = 1
```

multiplier

```
multiplier = k0/(16*pi*h0)*abs(sigma*h0)^2
```

n

```
n = length(L)
```

r

```
r =sqrt(areaconstr/pi)
```

r_max

```
r_max = 3*r
```

sigma

```
sigma = 0.001 + 0.001i
```

step

```
step = 1e-4
```

1.7 convertNRB.m File Reference

Variables

- function [curve_rep2]
- if curve_rep1 form

1.7.1 Variable Documentation

form

```
elseif curve_rep1 form
```

Initial value:

```
== "rB"
    curve_rep2 =
      nrbmak([curve_rep1.coefs(1:2,:);zeros(1,curve_rep1.number);curve_rep1.coefs(3,:)],curve_rep1.knots)
```

function

```
function[curve_rep2]
```

Initial value:

```
= convertNRB(curve_rep1)
%convert from MATLAB's rational spline curve format to NURBS toolbox format
%and vice versa
curve_rep2 = []
```

1.8 countSteps.m File Reference

Variables

- function steps
- uniqueKnots =unique(curve.knots(k:end-k+1))
- l = length(uniqueKnots)-1
- d0 = curve.number

1.8.1 Variable Documentation

d0

```
d0 = curve.number
```

l

```
l = length(uniqueKnots)-1
```

steps

```
steps
```

Initial value:

```
= countSteps(curve, DoFsLimit)
k = curve.order
```

uniqueKnots

```
uniqueKnots =unique(curve.knots(k:end-k+1))
```

1.9 create_circle.m File Reference

1.10 curve_area.m File Reference

Variables

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

1.10.1 Variable Documentation

function

```
function[val]
```

Initial value:

```
= curve_area(crv)
if crv.form == "B-NURBS"
    crv = rsmak(crv.knots,crv.coefs([1:2,4],:))
```

val

```
end val = integral(@func,crv.knots(crv.order),crv.knots(end-crv.order+1))
```

y

```
y
```

Initial value:

```
=func(x)
    [p] = fntlr(crv,2,x)
```

1.11 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

1.11.1 Variable Documentation

c

```
end c = [Mx,My]./area
```

function

```
function[c]
```

Initial value:

```
= curve_centroid(crv, area, only_x)
if nargin==1
    area = curve_area(crv)
```

Mx

```
end Mx = 0.5*integral(@funcx,crv.knots(crv.order),crv.knots(end-crv.order+1))
```

My

```
if ∼only_x My = -0.5*integral(@funcy,crv.knots(crv.order),crv.knots(end-crv.order+1))
```

y

```
y
```

Initial value:

```
=funcx(x)
    [p] = fntlr(crv,2,x)
```

1.12 dataFunc.m File Reference

Variables

- function [areaconstr, r, r_max, L, lambda, sigma, step]
- r =sqrt(areaconstr/pi)
- r_max = 4∗r
- L = 10
- lambda = 1
- sigma = 2e-5 + 2e-5∗1i
- step = 1e-4

1.12.1 Variable Documentation

function

```
function[areaconstr, r, r_max, L, lambda, sigma, step]
```

Initial value:
```
= dataFunc()

areaconstr = 2
```

L

```
L = 10
```

lambda

```
lambda = 1
```

r

```
r =sqrt(areaconstr/pi)
```

r_max

```
r_max = 4*r
```

sigma

```
sigma = 2e-5 + 2e-5*1i
```

step

```
step = 1e-4
```

1.13 dataKernelNumeric.m File Reference

Variables

- function [lambda, e, g]
- persistent e = e_numeric
- persistent g = geom
- UNTITLED4 Summary of this function goes here Detailed explanation goes here lambda = wavelength

1.13.1 Variable Documentation

e

```
elseif e = e_numeric
```

function

```
function[lambda, e, g]
```

Initial value:

```
= dataKernelNumeric(wavelength, e_numeric, geom)
persistent lambda
```

g

```
elseif g = geom
```

lambda

```
UNTITLED4 Summary of this function goes here Detailed explanation goes here lambda = wavelength
```

1.14 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

1.14.1 Variable Documentation

current

```
current = -4/(k0*120*pi)
```

function

```
function[res]
```

Initial value:

```
= E_back(p,lambda,L_plus_r)
k0 = 2*pi/lambda
```

H0

```
order H0 = besselh(nu,k,k0*sqrt((p(1,:)+L_plus_r).^2+p(2,:).^2))
```

k

```
switch nargin case point source k = 2
```

nu

```
kind nu = 0
```

res

```
end res = -k0*120*pi/4*current*H0
```

1.15 EzAtCNT.m File Reference

Variables

- function [out]
- CTM1 = (2.*(-1).^n.*H(n,k0a+k0L))./(2+cn0.*k0a.*pi.*H(n,k0a).*J(n,k0a))
- for j

1.15.1 Variable Documentation

CTM1

```
CTM1 = (2.*(-1).^n.*H(n,k0a+k0L))./(2+cn0.*k0a.*pi.*H(n,k0a).*J(n,k0a))
```

function

```
function[out]
```

Initial value:

```
= EzAtCNT(phi, cn0, k0a, k0L, N)
n    = -N:N
```

j

```
for j
```

Initial value:

```
=length(phi):-1:1
    out(j,:)  = sum(CTM1.*J(n, k0a).*exp(1i.*n.*phi(j)))
```

1.16 freeform_geom.m File Reference

Functions

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

Variables

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

1.16.1 Function Documentation

isempty()

```
if isempty (
             params(params >1|params< 0)  )  [new]
```

sin()

```
r * sin (
             phi  )
```

1.16.2 Variable Documentation

ctrl_pts

```
ctrl_pts = [r.*cos(phi)
```

function

```
function[geom]
```

Initial value:

```
= freeform_geom(params, max_r)

persistent max_r_per
```

geom

```
geom = [ ]
```

knots

```
normalize knot vector geom knots = (geom.knots-geom.knots(1))./(geom.knots(end)-geom.knots(1))
```

nargin

```
elseif nargin
```

Initial value:

```
==2
    max_r_per = max_r
```

phi

```
phi = (params(2,:)+(0:n-1))*2*pi/n
```

r

```
r = max_r_per.*(0.99.*params(1,:)+0.01)
```

1.17 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)]

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 = areagreen(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 = (areagreen(g2)-areagreen(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)

1.17.1 Function Documentation

abs()

```
while abs (
             area- area0 )
```

isempty()

```
if isempty (
             params(params >1|params< 0)  ) &&
```

sin()

```
r * sin (
             phi  )
```

1.17.2 Variable Documentation

area

```
area = areagreen(tmpgeom)
```

area_tol

```
area_tol =1e-4
```

ctrl_pts

```
ctrl_pts = compute_ctrl_pts(params,max_r)
```

darea

```
darea = (areagreen(g2)-areagreen(g1))/diff_tol
```

diff_tol

```
diff_tol =1e-6
```

function

```
end function[res]
```

Initial value:

```
= freeform_geom_withArea(params,max_r,area0)
geom = []
```

g1

```
g1 = nrbmak(pts1,0:1/(n+6):1)
```

g2

```
g2 = nrbmak(pts2,0:1/(n+6):1)
```

geom

```
geom = tmpgeom
```

knots

```
normalize knot vector geom knots = (geom.knots-geom.knots(1))./(geom.knots(end)-geom.knots(1))
```

max_r

```
max_r = max_r - (area-area0)/darea
```

phi

```
phi = (params(2,:)+(0:n-1))*2*pi/n
```

pts2

```
pts2 = compute_ctrl_pts(params,max_r+diff_tol/2)
```

res

```
res = [r.*cos(phi)
```

tmpgeom

```
end tmpgeom = nrbmak(ctrl_pts,0:1/(n+6):1)
```

1.18 freeform_geom_wo_pers.m File Reference

Functions

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

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))

1.18.1 Function Documentation

isempty()

```
if isempty (
             params(params >1|params< 0)  ) &&
```

sin()

```
r * sin (
             phi  )
```

1.18.2 Variable Documentation

ctrl_pts

```
ctrl_pts = [r.*cos(phi)
```

function

```
function[geom]
```

Initial value:

```
= freeform_geom_wo_pers(params,max_r)
geom = []
```

geom

```
geom = nrbmak(ctrl_pts,0:1/(n+6):1)
```

knots

```
normalize knot vector geom knots = (geom.knots-geom.knots(1))./(geom.knots(end)-geom.knots(1))
```

phi

```
phi = (params(2,:)+(0:n-1))*2*pi/n
```

r

```
r = max_r*(0.99*params(1,:)+0.01)
```

1.19 freeform_x_sym1_withArea.m File Reference

Functions

- if isempty (params(params >1|params< 0)) &&max_r >0 n
- while abs (area-area0)>area_tol &&nIter< 100[pts1
- if any (isnan(ctrl_pts), 'all')||any(isinf(ctrl_pts)
- r ∗ sin (phi)]
- res (2, n0+3:end)

Variables

- function [geom, cx]
- cx = 0
- x coord of area centroid area_tol =1e-4
- diff_tol =1e-6
- nIter =0
- 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
- if all return
- end geom = tmpgeom
- n0 = length(params)
- phi = (params(2,:)+(0:n-1))∗(pi/n - pi/180)+pi/180
- res = [r.∗cos(phi)
- a = res(1,1)
- b = res(1,end)
- knots = [zeros(1,3) 0:1/(n0-1):1 ones(1,3)]

1.19.1 Function Documentation

abs()

```
while abs (
             area- area0 ) &&
```

any()

```
if any (
             isnan(ctrl_pts) ,
             'all'  )
```

isempty()

```
if isempty (
             params(params >1|params< 0)  ) &&
```

res()

```
res (
             2 ,
             n0+3:end  )
```

sin()

```
r * sin (
             phi  )
```

1.19.2 Variable Documentation

a

```
a = res(1,1)
```

area

```
area = curve_area(tmpgeom)
```

area_tol

```
x coord of area centroid area_tol =1e-4
```

b

```
b = res(1,end)
```

cx

```
cx = 0
```

darea

```
darea = (curve_area(g2)-curve_area(g1))/diff_tol
```

diff_tol

```
diff_tol =1e-6
```

function

```
end function[res, knots]
```

Initial value:

```
= freeform_x_sym1_withArea(params,max_r,area0)
geom = []
```

g1

```
g1 = nrbmak(pts1,knots1)
```

g2

```
g2 = nrbmak(pts2,knots2)
```

geom

```
geom = tmpgeom
```

knots

```
knots = [zeros(1,3) 0:1/(n0-1):1 ones(1,3)]
```

knots1

```
while knots1 = compute_ctrl_pts(params,max_r-diff_tol/2)
```

max_r

```
max_r = max_r - (area-area0)/darea
```

n0

```
n0 = length(params)
```

nIter

```
nIter =0
```

phi

```
phi = (params(2,:)+(0:n-1))*(pi/n - pi/180)+pi/180
```

res

```
res = [r.*cos(phi)
```

return

```
if all return
```

tmpgeom

```
end tmpgeom = nrbmak(ctrl_pts,knots)
```

1.20 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)

1.20.1 Function Documentation

abs()

```
while abs (
             area- area0 )
```

isempty()

```
if isempty (
             params(params >1|params< 0)  ) &&
```

params()

```
if params (
             2 ,
             1  )
```

r()

```
r (
             2:end- 1 )
```

res() [1/2]

```
if && res (
             1 ,
             1  )
```

res() [2/2]

```
if res (
             2 ,
             2  )
```

1.20.2 Variable Documentation

area

```
area = curve_area(tmpgeom)
```

area_tol

```
x coord of area centroid area_tol =1e-4
```

cx

```
cx = 0
```

darea

```
darea = (curve_area(g2)-curve_area(g1))/diff_tol
```

diff_tol

```
diff_tol =1e-6
```

function

```
end function[res, knots]
```

Initial value:

```
= freeform_x_sym_withArea(params,max_r,area0)
geom = []
```

g1

```
g1 = nrbmak(pts1,knots1)
```

g2

```
g2 = nrbmak(pts2,knots2)
```

geom

```
geom = tmpgeom
```

knots1

```
while knots1 = compute_ctrl_pts(params,max_r-diff_tol/2)
```

max_r

```
max_r = max_r - (area-area0)/darea
```

n0

```
n0 = length(params)
```

phi

```
phi = (params(2,2:end-1)+(0:n-3))*(pi/(n-2) - pi/180)+pi/180
```

res

```
res = [r(2:end-1).*cos(phi)
```

tmpgeom

```
tmpgeom = nrbmak(ctrl_pts,knots)
```

1.21 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

1.21.1 Variable Documentation

function

```
function[res]
```

Initial value:

```
= Green(p0,p,lambda)
k0 = 2*pi/lambda
```

H0

```
order H0 = besselh(nu,k,k0*sqrt((p(1,:)-p0(1)).^2+(p(2,:)-p0(2)).^2))
```

k

```
k = 2
```

nu

```
kind nu = 0
```

res

```
res = -1i/4*H0
```

1.22 halfSpike.m File Reference

Functions

- x0 (length(x0)/2+1:end)]
- freeform_geom ([ ], 7.505) curve
- nrbctrlplot (curve)

Variables

- addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox [areaconstr, r, r_max, L, lambda, sigma, step] = dataFunc()
- x0 = [0.001,0.976868,0.0316843,0.00100835,0.944784,0.944305,0.00950486,0.0144524,0.913437,0.000847884,0.00949567,0.0495045,0.678524,0.042625,0.0347004,0.702235,0.00646753,0.0466691,0.527094,0.0632831,0.00374493,0.0114064,0.776126,0.0019033,0.0857474,0.031821,0.0107389,0.840329,0.065796,0.00627833,0.895396,0.436762,0.698289,0.124496,0.89597,0.994974,0.316554,0.0633606,0.161891,0.154962,0.581284,0.692315,0.29053,0.164839,0.217272,0.368261,0.132399,0.729571,0.881846,0.715367,0.298244,0.313699,0.372292,0.185516,0.868891,0.0127458,0.249449,0.235312,0.138519,0.72611,0.38182,0.625508,0.0848148,0.87296,0.0847924,0.714883,0.908252,0.473116,0.409339,0.216367,0.155927,0.0140034,0.892533,0.699062,0.0275017,0.264952,0.715356,0.360065,0.684838,0.48222]
- n = 11
- figure
- area

1.22.1 Function Documentation

freeform_geom()

```
freeform_geom (
             7.  505 )
```

nrbctrlplot()

```
nrbctrlplot (
             curve  )
```

x0()

```
x0 (
             length(x0)/2+1:end  )
```

1.22.2 Variable Documentation

area

```
area
```

Initial value:

```
= areagreen(curve)

opt_param = x0
```

figure

```
figure
```

n

```
n = 11
```

NURBStoolbox

```
addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox[areaconstr,
r, r_max, L, lambda, sigma, step] = dataFunc()
```

x0

```
x0 = [0.001,0.001,0.001,0.001,0.001,0.001,0.001,0.001,0.001,0.001,0.001,0.001,0.001,0.001,0.↩
001,0.001,0.001,0.001,0.001,0.001,0.001,0.976868,0.0316843,0.00100835,0.944784,0.944305,0.↩
00950486,0.0144524,0.913437,0.000847884,0.00949567,0.0495045,0.678524,0.042625,0.0347004,0.↩
702235,0.00646753,0.0466691,0.527094,0.0632831,0.00374493,0.0114064,0.776126,0.0019033,0.↩
0857474,0.031821,0.0107389,0.840329,0.065796,0.00627833,0.895396,0.436762,0.698289,0.124496,0.↩
89597,0.994974,0.316554,0.0633606,0.161891,0.154962,0.581284,0.692315,0.29053,0.164839,0.↩
217272,0.368261,0.132399,0.729571,0.881846,0.715367,0.298244,0.313699,0.372292,0.185516,0.↩
868891,0.0127458,0.249449,0.235312,0.138519,0.72611,0.38182,0.625508,0.0848148,0.87296,0.↩
0847924,0.714883,0.908252,0.473116,0.409339,0.216367,0.155927,0.0140034,0.892533,0.699062,0.↩
0275017,0.264952,0.715356,0.360065,0.684838,0.48222]
```

1.23 igabem_circle_refinement.m File Reference

Functions

- disp (phi0) n
- analytic_solution (i)
- size (solution) DoFs(i)
- Linf (:, i)
- max (abs(real(analytic_solution) -real(solution)))
- max (abs(imag(analytic_solution) -imag(solution)))]
- L2 (:, i)
- trapz ((real(analytic_solution) -real(solution)).^2)
- trapz ((imag(analytic_solution) -imag(solution)).^2)]

Variables

- function [L2, Linf, DoFs]
- end nSteps = countSteps(geometry,max_dofs)
- t0 =0:step:1
- vals = fnval(geometry,t0)
- phi0 =atan2(vals(2,:),vals(1,:))
- L2 =zeros(3,nSteps)
- Linf =zeros(3,nSteps)
- DoFs =zeros(1,nSteps)
- analytic_solution = zeros(n,1)
- r
- s =sigma(omega)
- parfor i

1.23.1 Function Documentation

analytic_solution()

```
analytic_solution (
             i  )
```

disp()

```
disp (
             phi0  )
```

L2()

```
L2 (
             :  ,
             i  )
```

Linf()

```
Linf (
             :  ,
             i  )
```

max() [1/2]

```
max (
             abs(imag(analytic_solution) -imag(solution))  )
```

max() [2/2]

```
max (
             abs(real(analytic_solution) -real(solution))  )
```

size()

```
size (
              solution  )
```

trapz() [1/2]

```
trapz (
              (imag(analytic_solution) -imag(solution)).^ 2 )
```

trapz() [2/2]

```
trapz (
              (real(analytic_solution) -real(solution)).^ 2 )
```

1.23.2 Variable Documentation

analytic_solution

```
analytic_solution = zeros(n,1)
```

DoFs

```
DoFs =zeros(1,nSteps)
```

function

```
function[L2, Linf, DoFs]
```

Initial value:

```
= igabem_circle_refinement(geometry,lambda,L,max_dofs,step,mode)
if strcmp(geometry.form,'B-NURBS')
    geometry = convertNRB(geometry)
```

i

```
end parfor i
```

Initial value:

```
=1:n
    analytic_solution(i) = EzAtCNT(phi0(i),120*pi*s,2*r*pi/lambda,2*L*pi/lambda,20)
```

L2

```
L2 =zeros(3,nSteps)
```

Linf

```
Linf =zeros(3,nSteps)
```

nSteps

```
end nSteps = countSteps(geometry,max_dofs)
```

phi0

```
phi0 =atan2(vals(2,:),vals(1,:))
```

r

```
r
```

Initial value:

```
=vals(1,1)
omega = 3e8 * 2*pi/lambda / 1e-9
```

s

```
s =sigma(omega)
```

t0

```
t0 =0:step:1
```

vals

```
vals = fnval(geometry,t0)
```

1.24 igabem_single_cnt.m File Reference

Functions

- Green (g, fnvals(1:2,:), lambda) if mode vals
- LHS (i,:)
- if greville (i) -0 >0 %quad1
- @integ_fun, greville(i true ()
- end LHS (i, k)
- end end b (i)
- 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(circle.knots,circle.order)
- DoFs = circle.number
- N = spcol(circle.knots,circle.order,greville)
- RBases = (circle.coefs(3,:).∗N)./(N∗circle.coefs(3,:)')
- Rational bases greville LHS = zeros(DoFs,DoFs)
- b = zeros(DoFs,1)
- p1 = fnval(circle,0)
- L_plus_r = L+r
- if mode t0 =0:step:1
- nPoints = length(t0)
- Rational bases greville Nt = spcol(circle.knots,circle.order,t0)
- Rt = (circle.coefs(3,:).∗Nt)./(Nt∗circle.coefs(3,:)')
- fnvals = fntlr(circle,2,t0)
- coefs = [1, repmat([4,2],1,floor((nPoints-2)/2)), 4, 1]
- end fngvals = fntlr(circle,1,greville)
- for i
- else for k
- quad2 = 0
- quad1 =quadgk(@integ_fun,0,greville(i))
- e_in = rsmak(circle.knots,[transpose(e_in_b).∗circle.coefs(3,:);circle.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)

1.24.1 Function Documentation

b()

```
end end b (
             i  )
```

cond()

```
end cond (
             RBases+ LHS )
```

g()

```
else g (
             i  )
```

Green()

```
Green (
            g ,
            fnvals(1:2,:)  ,
            lambda  )
```

greville()

```
if greville (
            i  )
```

LHS() [1/2]

```
end LHS (
            i ,
            k  )
```

LHS() [2/2]

```
LHS (
            i ,
            :    )
```

true()

```
@integ_fun, greville(i true ( )  [virtual]
```

∼isempty()

```
if ∼isempty (
            tv  )
```

1.24.2 Variable Documentation

b

```
b = zeros(DoFs,1)
```

coefs

```
coefs = [1, repmat([4,2],1,floor((nPoints-2)/2)), 4, 1]
```

DoFs

```
DoFs = circle.number
```

e_in

```
e_in = rsmak(circle.knots,[transpose(e_in_b).*circle.coefs(3,:);circle.coefs(3,:)])
```

fngvals

```
end fngvals = fntlr(circle,1,greville)
```

fnvals

```
fnvals = fntlr(circle,2,t0)
```

function

```
end end function[g]
```

Initial value:

```
= igabem_single_cnt(circle,L,r,lambda,sigma_val,step,mode)
if strcmp(circle.form,'B-NURBS')
    circle = convertNRB(circle)
```

greville

```
greville = aveknt(circle.knots,circle.order)
```

h0

```
h0 = 120*pi
```

i

```
for i
```

Initial value:

```
=1:DoFs
    g = fngvals(1:2,i)
```

k

```
else for k
```

Initial value:

```
=1:DoFs
            quad1 = 0
```

k0

```
end k0 = 2*pi/lambda
```

L_plus_r

```
L_plus_r = L+r
```

LHS

```
Rational bases greville LHS = zeros(DoFs,DoFs)
```

N

```
N = spcol(circle.knots,circle.order,greville)
```

nPoints

```
nPoints = length(t0)
```

Nt

```
Nt = spcol(circle.knots,circle.order,t0)
```

omega

```
omega = 3e8 * 2*pi/lambda / 1e-9
```

p1

```
p1 = fnval(circle,0)
```

quad1

```
quad1 =quadgk(@integ_fun,0,greville(i))
```

quad2

```
quad2 = 0
```

RBases

```
RBases = (circle.coefs(3,:).*N)./(N*circle.coefs(3,:)')
```

res

```
res = Green(g,vals(1:2,:),lambda).*Rt(:,k)'.*sqrt(vals(3,:).^2+vals(4,:).^2)
```

Rt

```
Rt = (circle.coefs(3,:).*Nt)./(Nt*circle.coefs(3,:)')
```

sigma_val

```
sigma_val = sigma(omega)
```

solution

```
else solution = Rt*e_in_b
```

t0

```
if mode t0 =0:step:1
```

1.25 IncreaseDoFsInModel.m File Reference

Functions

- if isempty (params(params >1|params< 0)) &&max_r > 0 g1
- areaconstraint ([ ], areaconstr)
- params2 (n+2:end)]

Variables

- function [geom2, params2, fval]
- geom2 = [ ]
- n = length(params)
- areaconstr = areagreen([ ],g1)
- opts = optimoptions('fmincon','Display','none')
- params2 = [params2(1:n+1)
- end function val
- end function dev
- p2 = fnval(g2,0:.01:1)
- if nargin
- in_area = areagreen(curve)
- c = abs(in_area-area0) - 1e-2
- ceq = [ ]

1.25.1 Function Documentation

areaconstraint()

```
areaconstraint (
            areaconstr  )
```

isempty()

```
if isempty (
            params(params >1|params< 0)  ) &&
```

params2()

```
params2 (
            n+2:end  )
```

1.25.2 Variable Documentation

areaconstr

```
areaconstr = areagreen([],g1)
```

c

```
c = abs(in_area-area0) - 1e-2
```

ceq

```
ceq = []
```

dev

```
dev
```

Initial value:

```
= compareCurves(g1,g2)
        p1 = fnval(g1,0:.01:1)
```

function

```
end function[c, ceq]
```

Initial value:

```
= IncreaseDoFsInModel(params,max_r)
params2 = []
```

geom2

```
geom2 = []
```

in_area

```
in_area = areagreen(curve)
```

n

```
n = length(params)
```

nargin

```
elseif nargin
```

Initial value:

```
== 2
           area0 = a0
```

opts

```
opts = optimoptions('fmincon','Display','none')
```

p2

```
p2 = fnval(g2,0:.01:1)
```

params2

```
params2 = [params2(1:n+1)
```

val

```
val
```

Initial value:

```
= objective(v)
        g2 = freeform_geom_wo_pers([v(1:n+1);v(n+2:end)],max_r)
```

1.26 info_curve.m File Reference

Functions

- id dataFunc ()
- kernel_analytic ([ ], r, sigma, lambda, L)
- disp ("Circle: Area = "+areaconstr+" | P_sct = "+scat_analytic)
- freeform_geom ([ ], r_max) %curve
- freeform_geom ([ ], 8.9445)
- nrbctrlplot (curve)
- kernel_numeric ([ ], lambda, e_numerical, curve)
- disp ("Ratio: "+scat_numerical/scat_analytic)

Variables

- P_sct for circle with same area k0 = 2∗pi/lambda
- h0 = 120∗pi
- multiplier = k0/(16∗pi∗h0)∗abs(sigma∗h0)^2
- scat_analytic = multiplier ∗ integral(@kernel_analytic,0,2∗pi)
- x0
- x1 = [0.743655,0.18929,0.0245574,0.01456,0.67417,0.0189316,0.225592,0.0310905,0.0188206,0.↩
0468859,0.0449436,0.0605055,0.22784,0.00779339,0.0427269,0.00578469,0.959214,0.0142547,0.↩
0105672,0.969899,0.106701,0.00260714,0.0173,0.910195,0.0230313,0.026505,0.0576546,0.014068,0.↩
0416613,0.868372,0.11606,0.0164118,0.193552,0.0182761,0.101555,0.623538,0.0296796,0.0196488,0.↩
964949,0.0617436,0.554608,0.438401,0.167019,0.181445,0.926196,0.763703,0.657349,0.587516,0.↩
0249876,0.0991114,0.0262314,0.78419,0.312979,0.570388,0.512629,0.960871,0.969149,0.163126,0.↩
458313,0.512084,0.779242,0.517895,0.623669,0.548462,0.0871972,0.920977,0.388815,0.916498,0.↩
621588,0.228947,0.651873,0.929406,0.755951,0.617836,0.585856,0.196035,0.181298,0.875225,0.↩
957696,0.188842]
- rArr = x1(1:40)
- phiArr = x1(41:end)
- x11 = 0.01 ∗ ones(1,15)
- x2 = 0.5 ∗ ones(1,60)
- curve = freeform_geom_withArea(x0, r_max, areaconstr)r
- area = areagreen(curve)
- e_numerical = igabem_single_cnt(curve,L,r,lambda,sigma,step,1)
- scat_numerical = multiplier ∗ integral(@kernel_numeric,0,2∗pi)

1.26.1 Function Documentation

dataFunc()

```
id dataFunc ( )  [virtual]
```

disp() [1/2]

```
disp ( )
```

disp() [2/2]

```
disp (
            "Ratio:  "+scat_numerical/ scat_analytic )
```

freeform_geom() [1/2]

```
freeform_geom (
            8.  9445 )
```

freeform_geom() [2/2]

```
freeform_geom (
            r_max  )
```

kernel_analytic()

```
kernel_analytic (
            r ,
            sigma ,
            lambda ,
            L  )
```

kernel_numeric()

```
kernel_numeric (
            lambda ,
            e_numerical ,
            curve  )
```

nrbctrlplot()

```
nrbctrlplot (
            curve  )
```

1.26.2 Variable Documentation

area

```
area = areagreen(curve)
```

curve

```
curve = freeform_geom_withArea(x0, r_max, areaconstr)r
```

e_numerical

```
e_numerical = igabem_single_cnt(curve,L,r,lambda,sigma,step,1)
```

h0

```
h0 = 120*pi
```

k0

```
P_sct for circle with same area k0 = 2*pi/lambda
```

multiplier

```
multiplier = k0/(16*pi*h0)*abs(sigma*h0)^2
```

phiArr

```
phiArr = x1(41:end)
```

rArr

```
rArr = x1(1:40)
```

scat_analytic

```
scat_analytic = multiplier * integral(@kernel_analytic,0,2*pi)
```

scat_numerical

```
scat_numerical = multiplier * integral(@kernel_numeric,0,2*pi)
```

x0

```
x0
```

Initial value:

```
=
```

x1

```
x1 = [0.743655,0.18929,0.0245574,0.01456,0.67417,0.0189316,0.225592,0.0310905,0.0188206,0.↩
0468859,0.0449436,0.0605055,0.22784,0.00779339,0.0427269,0.00578469,0.959214,0.0142547,0.↩
0105672,0.969899,0.106701,0.00260714,0.0173,0.910195,0.0230313,0.026505,0.0576546,0.014068,0.↩
0416613,0.868372,0.11606,0.0164118,0.193552,0.0182761,0.101555,0.623538,0.0296796,0.0196488,0.↩
964949,0.0617436,0.554608,0.438401,0.167019,0.181445,0.926196,0.763703,0.657349,0.587516,0.↩
0249876,0.0991114,0.0262314,0.78419,0.312979,0.570388,0.512629,0.960871,0.969149,0.163126,0.↩
458313,0.512084,0.779242,0.517895,0.623669,0.548462,0.0871972,0.920977,0.388815,0.916498,0.↩
621588,0.228947,0.651873,0.929406,0.755951,0.617836,0.585856,0.196035,0.181298,0.875225,0.↩
957696,0.188842]
```

x11

```
x11 = 0.01 * ones(1,15)
```

x2

```
x2 = 0.5 * ones(1,60)
```

1.27 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

1.27.1 Variable Documentation

i

```
for i
```

Initial value:

```
=1:l
    step = (uniqueKnots(i+1)-uniqueKnots(i))/(n+1)
```

j

```
for j
```

Initial value:

```
=1:n
        knotsIn((i-1)*n+j) = uniqueKnots(i)+j*step
```

knotsIn

```
knotsIn = zeros(1,l*n)
```

l

```
l = length(uniqueKnots)-1
```

new_curve

```
end end new_curve
```

Initial value:

```
= insertKnots(curve, n)
%add n knots for each non-zero interval
k = curve.order
```

uniqueKnots

```
uniqueKnots =unique(curve.knots(k:end-k+1))
```

1.28 insertOneKnot.m File Reference

Functions

- id order ()

Variables

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

1.28.1 Function Documentation

order()

```
id order ( )  [virtual]
```

1.28.2 Variable Documentation

i

```
for i
```

Initial value:

```
=place
    step = (uniqueKnots(i+1)-uniqueKnots(i))/(n+1)
```

j

```
for j
```

Initial value:

```
=1:n
        knotsIn((i-1)*n+j) = uniqueKnots(i)+j*step
```

knotsIn

```
knotsIn = zeros(1,l*n)
```

l

```
l = length(uniqueKnots)-1
```

new_curve

```
end end new_curve
```

Initial value:

```
= insertOneKnot(curve, place)
n = 1
```

place

```
place = floor(place)
```

uniqueKnots

```
uniqueKnots =unique(curve.knots(k:end-k+1))
```

1.29 kernel_analytic.m File Reference

Variables

- function [res]
- persistent s = sigma
- persistent lambda = wavelength
- persistent L = distance_from_source
- res = 0
- if nargin
- end function y

1.29.1 Variable Documentation

function

```
function[res]
```

Initial value:

```
= kernel_analytic(phi,radius,sigma,wavelength,distance_from_source)
persistent r
```

L

```
L = distance_from_source
```

lambda

```
lambda = wavelength
```

nargin

```
elseif nargin
```

Initial value:

```
== 5
    r  = radius
```

res

```
end res = 0
```

s

```
s = sigma
```

y

```
end function y
```

Initial value:
```
= func(x)
        y = exp(-1i*2*pi/lambda*r*cos(phi(j)-x)).*EzAtCNT(x,120*pi*s,2*r*pi/lambda,2*L*pi/lambda,14).'*r
```

1.30 kernel_numeric.m File Reference

Functions

- elseif isempty (lambda)[lambda

Variables

- function [res]
- persistent e = e_numeric
- persistent g = geom
- res = 0
- if nargin
- end function y
- p = fntlr(g,2,x)

1.30.1 Function Documentation

isempty()

```
elseif isempty (
             lambda  )
```

1.30.2 Variable Documentation

e

```
elseif e = e_numeric
```

function

```
function[res]
```

Initial value:
```
= kernel_numeric(phi,wavelength,e_numeric,geom)
persistent lambda
```

g

```
elseif g = geom
```

nargin

```
elseif nargin
```

Initial value:

```
== 4
    lambda = wavelength
```

p

```
p = fntlr(g,2,x)
```

res

```
end res = 0
```

y

```
General y
```

Initial value:

```
= func(x)
        e_vals = fnval(e,x)
```

1.31 new_freeform.m File Reference

Functions

- if isempty (params(params >1|params< 0)) n
- More free but wilder phi (1)
- r ∗ sin (phi)]

Variables

- function geom
- if nargin
- r = max_r_per.∗(0.99.∗params(1,:)+0.01)
- for i
- end phi = (params(2,:)+(0:n-1))∗2∗pi/n
- phi ctrl_pts = [r.∗cos(phi)
- normalize knot vector geom knots = (geom.knots-geom.knots(1))./(geom.knots(end)-geom.knots(1))

1.31.1 Function Documentation

isempty()

```
if isempty (
             params(params >1|params< 0)  )  [new]
```

phi()

```
More free but wilder phi (
             1  )
```

sin()

```
r * sin (
             phi  )
```

1.31.2 Variable Documentation

ctrl_pts

```
ctrl_pts = [r.*cos(phi)
```

geom

```
geom
```

Initial value:

```
= new_freeform(params, max_r)

persistent max_r_per
```

i

```
for i
```

Initial value:

```
= 2:n
            phi(i) = params(2,i)*(2*pi - phi(i-1)) + phi(i-1)
```

knots

```
normalize knot vector geom knots = (geom.knots-geom.knots(1))./(geom.knots(end)-geom.knots(1))
```

nargin

```
elseif nargin
```

Initial value:

```
==2
    max_r_per = max_r
```

phi

```
end phi = (params(2,:)+(0:n-1))*2*pi/n
```

r

```
r = max_r_per.*(0.99.*params(1,:)+0.01)
```

1.32 new_freeform_unclamped.m File Reference

Functions

- elseif isempty (max_r_per)[∼
- if isempty (params(params >1|params< 0)) n
- More free but wilder phi (1)
- r ∗ sin (phi)]

Variables

- function unclgeom
- if nargin
- elseif max_r_per
- r = max_r_per.∗(0.99.∗params(1,:)+0.01)
- for i
- end phi = (params(2,:)+(0:n-1))∗2∗pi/n
- phi ctrl_pts = [r.∗cos(phi)

1.32.1 Function Documentation

isempty() [1/2]

```
elseif isempty (
             max_r_per  )
```

isempty() [2/2]

```
if isempty (
             params(params >1|params< 0)  )  [new]
```

phi()

```
More free but wilder phi (
             1  )
```

sin()

```
r * sin (
             phi  )
```

1.32.2 Variable Documentation

ctrl_pts

```
ctrl_pts = [r.*cos(phi)
```

i

```
for i
```

Initial value:

```
= 2:n
            phi(i) = params(2,i)*(2*pi - phi(i-1)) + phi(i-1)
```

max_r_per

```
elseif max_r_per
```

nargin

```
elseif elseif nargin
```

Initial value:

```
==2
    max_r_per = max_r
```

phi

```
end phi = (params(2,:)+(0:n-1))*2*pi/n
```

r

```
r = max_r_per.*(0.99.*params(1,:)+0.01)
```

unclgeom

```
unclgeom
```

Initial value:

```
= new_freeform_unclamped(params, max_r)
%unclamped geometry for Increasing DoF
```

```
persistent max_r_per
```

1.33 newIncrDoF.m File Reference

Functions

- if isempty (params(params >1|params< 0)) n
- For the more free model phi (1)
- r ∗ sin (phi)]
- end end if phi (end)
- new_params (1,:)
- new_params (2, 1)

Variables

- function new_params
- r = max_r.∗(0.99.∗params(1,:)+0.01)
- for i
- end ctrl_pts = [r.∗cos(phi)
- geom = nrbmak(ctrl_pts,0:1/(n+6):1)
- assume cubics curve = convertNRB(geom)
- new_curve = insertOneKnot(curve,n/2)
- new_geom = convertNRB(new_curve)
- n = n+1
- ctrl_points = new_geom.coefs(1:2, 5:end-2)
- phi = atan2(ctrl_points(2,:),ctrl_points(1,:))
- end phi radii = sqrt(ctrl_points(1,:).^2+ctrl_points(2,:).^2)

1.33.1 Function Documentation

isempty()

```
if isempty (
             params(params >1|params< 0)  )  [new]
```

new_params() [1/2]

```
new_params (
             1 ,
             :    )
```

new_params() [2/2]

```
new_params (
             2 ,
             1  )
```

phi() [1/2]

```
For the more free model phi (
             1  )
```

phi() [2/2]

```
end end if phi  (
             end  )
```

sin()

```
r * sin (
             phi  )
```

1.33.2 Variable Documentation

ctrl_points

```
ctrl_points = new_geom.coefs(1:2, 5:end-2)
```

ctrl_pts

```
ctrl_pts = [r.*cos(phi)
```

curve

```
assume cubics curve = convertNRB(geom)
```

geom

```
geom = nrbmak(ctrl_pts,0:1/(n+6):1)
```

i

```
for i
```

Initial value:

```
= 2:n
        phi(i) = params(2,i)*(2*pi - phi(i-1)) + phi(i-1)
```

n

```
n = n+1
```

new_curve

```
new_curve = insertOneKnot(curve,n/2)
```

new_geom

```
new_geom = convertNRB(new_curve)
```

new_params

```
function new_params
```

Initial value:

```
= newIncrDoF(params, max_r)
% Increases DoF of the new parametric model

geom = []
```

phi

```
phi = atan2(ctrl_points(2,:),ctrl_points(1,:))
```

r

```
r = max_r.*(0.99.*params(1,:)+0.01)
```

radii

```
end phi radii = sqrt(ctrl_points(1,:).^2+ctrl_points(2,:).^2)
```

1.34 oldToNew.m File Reference

Functions

- r ∗ sin (phi)]
- new_params (1,:)
- new_params (2, 1)

Variables

- function new_params
- r = max_r.∗(0.99.∗params(1,:)+0.01)
- phi = (params(2,:)+(0:n-1))∗2∗pi/n
- ctrl_pts = [r.∗cos(phi)
- for i
- end end radii = sqrt(ctrl_pts(1,:).^2+ctrl_pts(2,:).^2)

1.34.1 Function Documentation

new_params() [1/2]

```
new_params (
              1 ,
              :    )
```

new_params() [2/2]

```
new_params (
              2 ,
              1  )
```

sin()

```
r * sin (
              phi  )
```

1.34.2 Variable Documentation

ctrl_pts

```
ctrl_pts = [r.*cos(phi)
```

i

```
for i
```

Initial value:

```
= 1:length(phi)
        if phi(i)<0
            phi(i) = phi(i) + 2*pi
```

new_params

```
function new_params
```

Initial value:

```
= oldToNew(params, max_r)
% Transforms the old parametric model to the new one

if isempty(params(params>1|params<0))
    n = length(params)
```

phi

```
phi = (params(2,:)+(0:n-1))*2*pi/n
```

r

```
r = max_r.*(0.99.*params(1,:)+0.01)
```

radii

```
end end radii = sqrt(ctrl_pts(1,:).^2+ctrl_pts(2,:).^2)
```

1.35 p_sctr.m File Reference

Functions

- disp ("iteration start") k0
- nrbctrlplot (curve)
- kernel_numeric ([ ], lambda, e_numerical, curve)
- title ("A="+area+" | P_{sct}="+scat_numerical)
- disp ("area="+area+" | P_sct="+scat_numerical)

Variables

- function scat_numerical
- persistent lambda = wavelength
- persistent L = distance_from_source
- persistent r = radius
- persistent step = stp
- if nargin
- end x = reshape(x, [2,length(x)/2])
- h0 = 120∗pi
- multiplier = k0/(16∗pi∗h0)∗abs(s∗h0)^2
- curve = freeform_geom(x)
- figure
- area = areagreen(curve)
- e_numerical = igabem_single_cnt(curve,L,r,lambda,s,step,1)
- t =0:.005:1
- p = fnval(curve,t)
- phi0 =atan2(p(2,:),p(1,:))

1.35.1 Function Documentation

disp() [1/2]

```
disp ( )
```

disp() [2/2]

```
disp (
             "iteration start"  )
```

kernel_numeric()

```
kernel_numeric (
             lambda ,
             e_numerical ,
             curve  )
```

nrbctrlplot()

```
nrbctrlplot (
             curve  )
```

title()

```
title ( )
```

1.35.2 Variable Documentation

area

```
area = areagreen(curve)
```

curve

```
curve = freeform_geom(x)
```

e_numerical

```
e_numerical = igabem_single_cnt(curve,L,r,lambda,s,step,1)
```

figure

```
figure
```

h0

```
h0 = 120*pi
```

L

```
L = distance_from_source
```

lambda

```
lambda = wavelength
```

multiplier

```
multiplier = k0/(16*pi*h0)*abs(s*h0)^2
```

nargin

```
elseif nargin
```

Initial value:

```
==6
    s = sigma
```

p

```
p = fnval(curve,t)
```

phi0

```
phi0 =atan2(p(2,:),p(1,:))
```

r

```
r = radius
```

scat_numerical

```
scat_numerical
```

Initial value:

```
= p_sctr(x,sigma,wavelength,distance_from_source, radius, stp)

persistent s
```

step

```
step = stp
```

t

```
t =0:.005:1
```

x

```
end x = reshape(x, [2,length(x)/2])
```

1.36 test_calc.m File Reference

Functions

- calculation with analytic solution kernel_analytic ([ ], r, sigma, lambda, L)
- kernel_numeric ([ ], lambda, e_numerical, circle)
- hold on plot (t, abs(p), 'k-')
- plot (t, real(e_analytic), 'r--') plot(t
- imag (e_analytic)
- r real (p)
- plot (t, imag(p), 'k:')

Variables

- clear addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox figure
- example with circle and point source lambda = 1
- step =1e-4
- L =10
- h0 = 120∗pi
- sigma = 0.001+0.001i
- r =sqrt(2/pi)
- circle with arc segments and a radius of r circle = convertNRB(circle)
- multiplier = k0/(16∗pi∗h0)∗abs(sigma∗h0)^2
- tic scat_analytic
- e_numerical = igabem_single_cnt(circle,L,r,lambda,sigma,step,1)
- t =0:.005:1
- p = fnval(circle,t)
- phi0 =atan2(p(2,:),p(1,:))
- e_analytic = EzAtCNT(phi0,120∗pi∗sigma,2∗r∗pi/lambda,2∗L∗pi/lambda,14)
- tic scat_numerical
- r __pad0__

1.36.1 Function Documentation

imag()

```
imag (
              e_analytic  )
```

kernel_analytic()

```
calculation with analytic solution kernel_analytic (
              r ,
              sigma ,
              lambda ,
              L  )
```

kernel_numeric()

```
kernel_numeric (
             lambda ,
             e_numerical ,
             circle  )
```

plot() [1/3]

```
hold on plot (
             t ,
             abs(p) ,
             'k-'  )
```

plot() [2/3]

```
plot (
             t ,
             imag(p) ,
             'k:'    )
```

plot() [3/3]

```
plot (
             t ,
             real(e_analytic) ,
             'r--'  )
```

real()

```
r real (
             p  )
```

1.36.2 Variable Documentation

__pad0__

```
r __pad0__
```

circle

```
circle with arc segments and a radius of r circle = convertNRB(circle)
```

e_analytic

```
e_analytic = EzAtCNT(phi0,120*pi*sigma,2*r*pi/lambda,2*L*pi/lambda,14)
```

e_numerical

```
e_numerical = igabem_single_cnt(circle,L,r,lambda,sigma,step,1)
```

figure

```
clear addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox
figure
```

h0

```
h0 = 120*pi
```

L

```
L =10
```

lambda

```
lambda = 1
```

multiplier

```
multiplier = k0/(16*pi*h0)*abs(sigma*h0)^2
```

p

```
p = fnval(circle,t)
```

phi0

```
phi0 =atan2(p(2,:),p(1,:))
```

r

```
r =sqrt(2/pi)
```

scat_analytic

```
tic scat_analytic
```

Initial value:

```
=  multiplier * integral(@kernel_analytic,0,2*pi)
toc
%calculation with numerical solution
circle = insertKnots(circle, 7)
```

scat_numerical

```
tic scat_numerical
```

Initial value:

```
= multiplier * integral(@kernel_numeric,0,2*pi)
toc
hold off
plot(t,abs(e_analytic),'r-')
```

sigma

```
sigma = 0.001+0.001i
```

step

```
step =1e-4
```

t

```
t =0:.005:1
```

1.37 test_calc_multiple.m File Reference

Functions

- L (i)
- sigma (i)
- r (i)
- calculation with analytic solution kernel_analytic ([ ], r(i), sigma(i), lambda(i), L(i))
- scat_analytic (i)
- kernel_numeric ([ ], lambda(i), e_numerical, circle)
- scat_numerical (i)
- fprintf (' Analytical:%f\n', scat_analytic(i))
- fprintf (' Numeric:%f\n\n', scat_numerical(i))
- end hold off plot (t, abs(e_analytic), 'r-')
- hold on plot (t, abs(p), 'k-')
- plot (t, real(e_analytic), 'r--') % plot(t
- imag (e_analytic)
- r real (p)
- plot (t, imag(p), 'k:')

Variables

- clear addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox example with circle and point source lambda = 100
- step =1e-4
- L =500
- h0 = 120∗pi
- sigma = 0.001+0.001i
- r =30
- circle with arc segments and a radius of r circle = convertNRB(circle)
- parfor i
- re = -5 +(5+5)∗rand(1,1)
- im = -5 +(5+5)∗rand(1,1)
- multiplier = k0/(16∗pi∗h0)∗abs(sigma(i)∗h0)^2
- e_numerical = igabem_single_cnt(circle,L(i),r(i), lambda(i),sigma(i),step,1)
- t =0:.005:1
- p = fnval(circle,t)
- phi0 =atan2(p(2,:),p(1,:))
- e_analytic = EzAtCNT(phi0,120∗pi∗sigma(i),2∗r(i)∗pi/lambda(i),2∗L(i)∗pi/lambda(i),14)
- r __pad0__

1.37.1 Function Documentation

fprintf() [1/2]

```
fprintf (
              ' Analytical:%f\n' ,
              scat_analytic(i)  )
```

fprintf() [2/2]

```
fprintf (
              ' Numeric:%f\n\n' ,
              scat_numerical(i)  )
```

imag()

```
imag (
              e_analytic  )
```

kernel_analytic()

```
calculation with analytic solution kernel_analytic (
              r(i) ,
              sigma(i) ,
              lambda(i) ,
              L(i)  )
```

kernel_numeric()

```
kernel_numeric (
              lambda(i) ,
              e_numerical ,
              circle  )
```

L()

```
L (
              i  )
```

plot() [1/4]

```
end hold off plot (
              t ,
              abs(e_analytic) ,
              'r-'  )
```

plot() [2/4]

```
hold on plot (
              t ,
              abs(p) ,
              'k-'  )
```

plot() [3/4]

```
plot (
              t ,
              imag(p) ,
              'k:'    )
```

plot() [4/4]

```
plot (
              t ,
              real(e_analytic) ,
              'r--'  )
```

r()

```
r (
              i  )
```

real()

```
r real (
             p  )
```

scat_analytic()

```
scat_analytic (
             i  )
```

scat_numerical()

```
scat_numerical (
             i  )
```

sigma()

```
sigma (
             i  )
```

1.37.2 Variable Documentation

__pad0__

```
r __pad0__
```

circle

```
circle with arc segments and a radius of r circle = convertNRB(circle)
```

e_analytic

```
e_analytic = EzAtCNT(phi0,120*pi*sigma(i),2*r(i)*pi/lambda(i),2*L(i)*pi/lambda(i),14)
```

e_numerical

```
e_numerical = igabem_single_cnt(circle,L(i),r(i), lambda(i),sigma(i),step,1)
```

h0

```
h0 = 120*pi
```

i

```
parfor i
```

Initial value:

```
=1:20
    lambda(i) = randi([1,200])
```

im

```
im = -5 +(5+5)*rand(1,1)
```

L

```
L =500
```

lambda

```
lambda = 100
```

multiplier

```
multiplier = k0/(16*pi*h0)*abs(sigma(i)*h0)^2
```

p

```
p = fnval(circle,t)
```

phi0

```
phi0 =atan2(p(2,:),p(1,:))
```

r

```
r =30
```

re

```
re = -5 +(5+5)*rand(1,1)
```

sigma

```
sigma = 0.001+0.001i
```

step

```
step =1e-4
```

t

```
t =0:.005:1
```

1.38 test_freeform.m File Reference

Functions

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

Variables

- clear clc addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox generate random freeform curves with varying number of sectors assume a max radius of r_max = 5
- curve = convertNRB(curve)

1.38.1 Function Documentation

nrbctrlplot()

```
nexttile nrbctrlplot (
             curve  )
```

tiledlayout()

```
tiledlayout (
             3 ,
             3  )
```

1.38.2 Variable Documentation

curve

```
curve = convertNRB(curve)
```

r_max

```
clear clc addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox
generate random freeform curves with varying number of sectors assume a max radius of r_max =
5
```

1.39 test_newIncrDoF.m File Reference

Functions

- id r_max ()
- x0 (1,:)
- x0 (2,:)
- new_freeform ([ ], r_max)
- nrbctrlplot (geom) % nrbctrlplot(geom) figure nrbctrlplot(unclgeom) unclcurve
- nrbctrlplot (new_geom) new _curve2
- nrbctrlplot (new_geom2) uncl_new_curve2
- nrbctrlplot (uncl_new_geom2) geom.coefs new _geom2.coefs ctrl_points
- params (1,:)
- params (2, 1)

Variables

- x0 = xinit2
- new_curve = insertKnots(curve,1)
- new_geom = convertNRB(new_curve)
- figure
- new_geom2 = convertNRB(new_curve2)
- for i
- end end radii = sqrt(ctrl_points(1,:).^2+ctrl_points(2,:).^2)
- n = 9

1.39.1 Function Documentation

new_freeform()

```
new_freeform (
              r_max  )
```

nrbctrlplot() [1/4]

```
nrbctrlplot (
              geom  )
```

nrbctrlplot() [2/4]

```
nrbctrlplot (
              new_geom  )  [new]
```

nrbctrlplot() [3/4]

```
nrbctrlplot (
              new_geom2  )
```

nrbctrlplot() [4/4]

```
nrbctrlplot (
             uncl_new_geom2  )  [new]
```

params() [1/2]

```
params (
             1 ,
             :    )
```

params() [2/2]

```
params (
             2 ,
             1  )
```

r_max()

```
id r_max ( )  [virtual]
```

x0() [1/2]

```
x0 (
             1 ,
             :    )
```

x0() [2/2]

```
x0 (
             2 ,
             :    )
```

1.39.2 Variable Documentation

figure

```
figure
```

i

```
for i
```

Initial value:

```
= 1: length(phi)
    if phi(i)<0
        phi(i) = phi(i) + 2*pi
```

n

```
n = 9
```

new_curve

```
new_curve = insertKnots(curve,1)
```

new_geom

```
new_geom = convertNRB(new_curve)
```

new_geom2

```
new_geom2 = convertNRB(new_curve2)
```

radii

```
end end radii = sqrt(ctrl_points(1,:).^2+ctrl_points(2,:).^2)
```

x0

```
x0 = xinit2
```

1.40 test_optimize.m File Reference

Functions

- areaconstraint ([ ], areaconstr)
- kernel_analytic ([ ], r, sigma, lambda, L)
- p_sctr ([ ], sigma, lambda, L, r, step)
- nrbctrlplot (geom)
- disp ("For circle: "+scat_analytic)
- disp ("For curve: "+p_sctr_val)
- disp ("P_sct_curve/P_sct_circle: "+ratio)

Variables

- clear
- addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox [areaconstr, r, r_max, L, lambda, sigma, step] = dataFunc()
- P_sct for circle with same area k0 = 2∗pi/lambda
- h0 = 120∗pi
- multiplier = k0/(16∗pi∗h0)∗abs(sigma∗h0)^2
- scat_analytic
- sector = 5
- fun = @p_sctr
- A = [ ]
- b = [ ]
- Aeq = [ ]
- beq = [ ]
- lb = zeros(1, 2∗sector)
- ub = ones(1, 2∗sector)
- nonlcon = @areaconstraint
- options = optimoptions('fmincon','Display','iter','UseParallel',true)
- opt_param = ga(fun, 2∗sector, A, b, Aeq, beq, lb, ub, nonlcon, options)
- geom = freeform_geom(opt_param)
- p_sctr_val = abs(p_sctr(opt_param))
- ratio = p_sctr_val/scat_analytic

1.40.1 Function Documentation

areaconstraint()

```
areaconstraint (
            areaconstr  )
```

disp() [1/3]

```
disp (
            "For circle:  "+ scat_analytic )
```

disp() [2/3]

```
disp (
            "For curve:  "+ p_sctr_val )
```

disp() [3/3]

```
disp (
            "P_sct_curve/P_sct_circle:  "+ ratio )
```

kernel_analytic()

```
kernel_analytic (
             r ,
             sigma ,
             lambda ,
             L  )
```

nrbctrlplot()

```
nrbctrlplot (
             geom  )
```

p_sctr()

```
p_sctr (
             sigma ,
             lambda ,
             L ,
             r ,
             step  )
```

1.40.2 Variable Documentation

A

```
A = []
```

Aeq

```
Aeq = []
```

b

```
b = []
```

beq

```
beq = []
```

clear

```
clear
```

fun

```
fun = @p_sctr
```

geom

```
geom = freeform_geom(opt_param)
```

h0

```
h0 = 120*pi
```

k0

```
P_sct for circle with same area k0 = 2*pi/lambda
```

lb

```
lb = zeros(1, 2*sector)
```

multiplier

```
multiplier = k0/(16*pi*h0)*abs(sigma*h0)^2
```

nonlcon

```
nonlcon = @areaconstraint
```

NURBStoolbox

```
addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox[areaconstr,
r, r_max, L, lambda, sigma, step] = dataFunc()
```

opt_param

```
opt_param = ga(fun, 2*sector, A, b, Aeq, beq, lb, ub, nonlcon, options)
```

options

```
options = optimoptions('fmincon','Display','iter','UseParallel',true)
```

p_sctr_val

```
p_sctr_val = abs(p_sctr(opt_param))
```

ratio

```
ratio = p_sctr_val/scat_analytic
```

scat_analytic

```
scat_analytic
```

Initial value:

```
=  multiplier * integral(@kernel_analytic,0,2*pi)

freeform_geom([],r_max)
```

sector

```
sector = 5
```

ub

```
ub = ones(1, 2*sector)
```

1.41 test_optimize2.m File Reference

Functions

- areaconstraint ([ ], areaconstr)
- kernel_analytic ([ ], r, sigma, lambda, L)
- p_sctr ([ ], sigma, lambda, L, r, step)
- nrbctrlplot (geom)
- disp ("For circle: "+scat_analytic)
- disp ("For curve: "+p_sctr_val)
- disp ("P_sct_curve/P_sct_circle: "+ratio)

Variables

- clear
- addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox [areaconstr, r, r_max, L, lambda, sigma, step] = dataFunc()
- P_sct for circle with same area k0 = 2∗pi/lambda
- h0 = 120∗pi
- multiplier = k0/(16∗pi∗h0)∗abs(sigma∗h0)^2
- scat_analytic
- fun = @p_sctr
- x0
- sector = length(x0)
- A = [ ]
- b = [ ]
- Aeq = [ ]
- beq = [ ]
- lb = zeros(2, sector)
- ub = ones(2, sector)
- nonlcon = @areaconstraint
- options = optimoptions('fmincon','Display','iter','UseParallel',true)
- opt_param = fmincon(fun, x0, A, b, Aeq, beq, lb, ub, nonlcon, options)
- geom = new_freeform(opt_param)
- p_sctr_val = abs(p_sctr(opt_param))
- ratio = p_sctr_val/scat_analytic

1.41.1 Function Documentation

areaconstraint()

```
areaconstraint (
            areaconstr  )
```

disp() [1/3]

```
disp (
            "For circle:  "+ scat_analytic )
```

disp() [2/3]

```
disp (
            "For curve:  "+ p_sctr_val )
```

disp() [3/3]

```
disp (
            "P_sct_curve/P_sct_circle:  "+ ratio )
```

kernel_analytic()

```
kernel_analytic (
             r ,
             sigma ,
             lambda ,
             L  )
```

nrbctrlplot()

```
nrbctrlplot (
             geom  )
```

p_sctr()

```
p_sctr (
             sigma ,
             lambda ,
             L ,
             r ,
             step  )
```

1.41.2 Variable Documentation

A

```
A = []
```

Aeq

```
Aeq = []
```

b

```
b = []
```

beq

```
beq = []
```

clear

```
clear
```

fun

```
fun = @p_sctr
```

geom

```
geom = new_freeform(opt_param)
```

h0

```
h0 = 120*pi
```

k0

```
P_sct for circle with same area k0 = 2*pi/lambda
```

lb

```
lb = zeros(2, sector)
```

multiplier

```
multiplier = k0/(16*pi*h0)*abs(sigma*h0)^2
```

nonlcon

```
nonlcon = @areaconstraint
```

NURBStoolbox

```
addpath Desktop WICAN Toolboxes igesToolbox addpath Desktop WICAN Toolboxes NURBStoolbox[areaconstr,
r, r_max, L, lambda, sigma, step] = dataFunc()
```

opt_param

```
opt_param = fmincon(fun, x0, A, b, Aeq, beq, lb, ub, nonlcon, options)
```

options

```
options = optimoptions('fmincon','Display','iter','UseParallel',true)
```

p_sctr_val

```
p_sctr_val = abs(p_sctr(opt_param))
```

ratio

```
ratio = p_sctr_val/scat_analytic
```

scat_analytic

```
scat_analytic
```

Initial value:

```
=  multiplier * integral(@kernel_analytic,0,2*pi)

new_freeform([],r_max)
```

sector

```
sector = length(x0)
```

ub

```
ub = ones(2, sector)
```

x0

```
x0
```

Initial value:

```
=
```

Index

__pad0__
  test_calc.m, 159
  test_calc_multiple.m, 164
~isempty
  igabem_single_cnt.m, 133

A
  test_optimize.m, 171
  test_optimize2.m, 175
a
  freeform_x_sym1_withArea.m, 121
abs
  freeform_geom_withArea.m, 117
  freeform_x_sym1_withArea.m, 120
  freeform_x_sym_withArea.m, 124
Aeq
  test_optimize.m, 171
  test_optimize2.m, 175
analytic_solution
  igabem_circle_refinement.m, 129, 130
any
  freeform_x_sym1_withArea.m, 120
area
  areagreen.m, 101
  freeform_geom_withArea.m, 117
  freeform_x_sym1_withArea.m, 121
  freeform_x_sym_withArea.m, 124
  halfSpike.m, 128
  info_curve.m, 140
  p_sctr.m, 156
area_tol
  freeform_geom_withArea.m, 117
  freeform_x_sym1_withArea.m, 121
  freeform_x_sym_withArea.m, 124
areaconstr
  changeDistance.m, 107
  IncreaseDoFsInModel.m, 137
areaconstraint
  IncreaseDoFsInModel.m, 137
  test_optimize.m, 170
  test_optimize2.m, 174
areaconstraint.m, 100
  c, 100
  ceq, 100
  curve, 100
  function, 100
  in_area, 100
  nargin, 100
  x, 100
areagreen.m, 101
  area, 101
  function, 101
  y, 101

b
  freeform_x_sym1_withArea.m, 121
  igabem_single_cnt.m, 132, 133
  test_optimize.m, 171
  test_optimize2.m, 175
beq
  test_optimize.m, 171
  test_optimize2.m, 175

c
  areaconstraint.m, 100
  curve_centroid.m, 111
  IncreaseDoFsInModel.m, 137
ceq
  areaconstraint.m, 100
  IncreaseDoFsInModel.m, 137
changeAngle.m, 101
  circle, 103
  clear, 103
  curve, 102, 103
  dataFunc, 102
  delAng, 103
  figure, 103
  freeform_geom, 102
  geom, 103
  h0, 103
  i, 103
  k0, 104
  kernel_numeric, 102
  multiplier, 104
  n, 104
  plot, 102
  ratio, 104
  scat_analytic, 104
  scat_numerical, 102
  title, 103
  x0, 104
  x1, 104
  x11, 104
  x2, 104
changeAngleGeom.m, 105
  coefs, 105
  new_geom, 105
  th, 105
changeAngleNewParam.m, 106
  i, 106
  isempty, 106
  n, 106
  new_params, 106, 107
  phi, 106, 107
changeDistance.m, 107
  areaconstr, 107
  circle, 107
  h0, 107
  i, 108
  k0, 108
  L, 108

lambda, 108
multiplier, 108
n, 108
r, 108
r_max, 108
sigma, 108
step, 108
circle
changeAngle.m, 103
changeDistance.m, 107
test_calc.m, 159
test_calc_multiple.m, 164
clear
changeAngle.m, 103
test_optimize.m, 171
test_optimize2.m, 175
coefs
changeAngleGeom.m, 105
igabem_single_cnt.m, 133
cond
igabem_single_cnt.m, 132
convertNRB.m, 109
form, 109
function, 109
countSteps.m, 109
d0, 109
l, 109
steps, 110
uniqueKnots, 110
create_circle.m, 110
CTM1
EzAtCNT.m, 114
ctrl_points
newIncrDoF.m, 152
ctrl_pts
freeform_geom.m, 115
freeform_geom_withArea.m, 117
freeform_geom_wo_pers.m, 119
new_freeform.m, 148
new_freeform_unclamped.m, 150
newIncrDoF.m, 152
oldToNew.m, 154
current
E_back.m, 113
curve
areaconstraint.m, 100
changeAngle.m, 102, 103
info_curve.m, 140
newIncrDoF.m, 152
p_sctr.m, 156
test_freeform.m, 166
curve_area.m, 110
function, 110
val, 110
y, 110
curve_centroid.m, 111
c, 111
function, 111
Mx, 111
My, 111
y, 111
cx
freeform_x_sym1_withArea.m, 121
freeform_x_sym_withArea.m, 124

d0
countSteps.m, 109
darea
freeform_geom_withArea.m, 117
freeform_x_sym1_withArea.m, 121
freeform_x_sym_withArea.m, 125
dataFunc
changeAngle.m, 102
info_curve.m, 139
dataFunc.m, 111
function, 112
L, 112
lambda, 112
r, 112
r_max, 112
sigma, 112
step, 112
dataKernelNumeric.m, 112
e, 113
function, 113
g, 113
lambda, 113
delAng
changeAngle.m, 103
dev
IncreaseDoFsInModel.m, 137
diff_tol
freeform_geom_withArea.m, 117
freeform_x_sym1_withArea.m, 121
freeform_x_sym_withArea.m, 125
disp
igabem_circle_refinement.m, 129
info_curve.m, 139
p_sctr.m, 155
test_optimize.m, 170
test_optimize2.m, 174
DoFs
igabem_circle_refinement.m, 130
igabem_single_cnt.m, 133

e
dataKernelNumeric.m, 113
kernel_numeric.m, 146
e_analytic
test_calc.m, 159
test_calc_multiple.m, 164
E_back.m, 113
current, 113
function, 113
H0, 113
k, 114
nu, 114

res, 114
e_in
igabem_single_cnt.m, 134
e_numerical
info_curve.m, 140
p_sctr.m, 156
test_calc.m, 159
test_calc_multiple.m, 164
EzAtCNT.m, 114
CTM1, 114
function, 114
j, 114

figure
changeAngle.m, 103
halfSpike.m, 128
p_sctr.m, 156
test_calc.m, 160
test_newIncrDoF.m, 168
fngvals
igabem_single_cnt.m, 134
fnvals
igabem_single_cnt.m, 134
form
convertNRB.m, 109
fprintf
test_calc_multiple.m, 162
freeform_geom
changeAngle.m, 102
halfSpike.m, 127
info_curve.m, 140
freeform_geom.m, 115
ctrl_pts, 115
function, 115
geom, 115
isempty, 115
knots, 115
nargin, 116
phi, 116
r, 116
sin, 115
freeform_geom_withArea.m, 116
abs, 117
area, 117
area_tol, 117
ctrl_pts, 117
darea, 117
diff_tol, 117
function, 117
g1, 117
g2, 118
geom, 118
isempty, 117
knots, 118
max_r, 118
phi, 118
pts2, 118
res, 118
sin, 117
tmpgeom, 118
freeform_geom_wo_pers.m, 118
ctrl_pts, 119
function, 119
geom, 119
isempty, 119
knots, 119
phi, 119
r, 119
sin, 119
freeform_x_sym1_withArea.m, 120
a, 121
abs, 120
any, 120
area, 121
area_tol, 121
b, 121
cx, 121
darea, 121
diff_tol, 121
function, 122
g1, 122
g2, 122
geom, 122
isempty, 120
knots, 122
knots1, 122
max_r, 122
n0, 122
nIter, 122
phi, 122
res, 121, 123
return, 123
sin, 121
tmpgeom, 123
freeform_x_sym_withArea.m, 123
abs, 124
area, 124
area_tol, 124
cx, 124
darea, 125
diff_tol, 125
function, 125
g1, 125
g2, 125
geom, 125
isempty, 124
knots1, 125
max_r, 125
n0, 125
params, 124
phi, 125
r, 124
res, 124, 126
tmpgeom, 126
fun
test_optimize.m, 171
test_optimize2.m, 175

function
areaconstraint.m, 100
areagreen.m, 101
convertNRB.m, 109
curve_area.m, 110
curve_centroid.m, 111
dataFunc.m, 112
dataKernelNumeric.m, 113
E_back.m, 113
EzAtCNT.m, 114
freeform_geom.m, 115
freeform_geom_withArea.m, 117
freeform_geom_wo_pers.m, 119
freeform_x_sym1_withArea.m, 122
freeform_x_sym_withArea.m, 125
Green.m, 126
igabem_circle_refinement.m, 130
igabem_single_cnt.m, 134
IncreaseDoFsInModel.m, 137
kernel_analytic.m, 145
kernel_numeric.m, 146

g
dataKernelNumeric.m, 113
igabem_single_cnt.m, 132
kernel_numeric.m, 146
g1
freeform_geom_withArea.m, 117
freeform_x_sym1_withArea.m, 122
freeform_x_sym_withArea.m, 125
g2
freeform_geom_withArea.m, 118
freeform_x_sym1_withArea.m, 122
freeform_x_sym_withArea.m, 125
geom
changeAngle.m, 103
freeform_geom.m, 115
freeform_geom_withArea.m, 118
freeform_geom_wo_pers.m, 119
freeform_x_sym1_withArea.m, 122
freeform_x_sym_withArea.m, 125
new_freeform.m, 148
newIncrDoF.m, 152
test_optimize.m, 172
test_optimize2.m, 176
geom2
IncreaseDoFsInModel.m, 137
Green
igabem_single_cnt.m, 132
Green.m, 126
function, 126
H0, 126
k, 126
nu, 126
res, 126
greville
igabem_single_cnt.m, 133, 134

H0
E_back.m, 113
Green.m, 126
h0
changeAngle.m, 103
changeDistance.m, 107
igabem_single_cnt.m, 134
info_curve.m, 141
p_sctr.m, 156
test_calc.m, 160
test_calc_multiple.m, 164
test_optimize.m, 172
test_optimize2.m, 176
halfSpike.m, 127
area, 128
figure, 128
freeform_geom, 127
n, 128
nrbctrlplot, 127
NURBStoolbox, 128
x0, 127, 128

i
changeAngle.m, 103
changeAngleNewParam.m, 106
changeDistance.m, 108
igabem_circle_refinement.m, 130
igabem_single_cnt.m, 134
insertKnots.m, 142
insertOneKnot.m, 144
new_freeform.m, 148
new_freeform_unclamped.m, 150
newIncrDoF.m, 152
oldToNew.m, 154
test_calc_multiple.m, 164
test_newIncrDoF.m, 168
igabem_circle_refinement.m, 128
analytic_solution, 129, 130
disp, 129
DoFs, 130
function, 130
i, 130
L2, 129, 130
Linf, 129, 130
max, 129
nSteps, 131
phi0, 131
r, 131
s, 131
size, 129
t0, 131
trapz, 130
vals, 131
igabem_single_cnt.m, 131
∼isempty, 133
b, 132, 133
coefs, 133
cond, 132
DoFs, 133
e_in, 134

fngvals, 134
fnvals, 134
function, 134
g, 132
Green, 132
greville, 133, 134
h0, 134
i, 134
k, 134
k0, 134
L_plus_r, 135
LHS, 133, 135
N, 135
nPoints, 135
Nt, 135
omega, 135
p1, 135
quad1, 135
quad2, 135
RBases, 135
res, 136
Rt, 136
sigma_val, 136
solution, 136
t0, 136
true, 133
im
test_calc_multiple.m, 165
imag
test_calc.m, 158
test_calc_multiple.m, 162
in_area
areaconstraint.m, 100
IncreaseDoFsInModel.m, 138
IncreaseDoFsInModel.m, 136
areaconstr, 137
areaconstraint, 137
c, 137
ceq, 137
dev, 137
function, 137
geom2, 137
in_area, 138
isempty, 137
n, 138
nargin, 138
opts, 138
p2, 138
params2, 137, 138
val, 138
info_curve.m, 139
area, 140
curve, 140
dataFunc, 139
disp, 139
e_numerical, 140
freeform_geom, 140
h0, 141
k0, 141
kernel_analytic, 140
kernel_numeric, 140
multiplier, 141
nrbctrlplot, 140
phiArr, 141
rArr, 141
scat_analytic, 141
scat_numerical, 141
x0, 141
x1, 141
x11, 142
x2, 142
insertKnots.m, 142
i, 142
j, 142
knotsIn, 142
l, 143
new_curve, 143
uniqueKnots, 143
insertOneKnot.m, 143
i, 144
j, 144
knotsIn, 144
l, 144
new_curve, 144
order, 143
place, 144
uniqueKnots, 144
isempty
changeAngleNewParam.m, 106
freeform_geom.m, 115
freeform_geom_withArea.m, 117
freeform_geom_wo_pers.m, 119
freeform_x_sym1_withArea.m, 120
freeform_x_sym_withArea.m, 124
IncreaseDoFsInModel.m, 137
kernel_numeric.m, 146
new_freeform.m, 147
new_freeform_unclamped.m, 149
newIncrDoF.m, 151

j
EzAtCNT.m, 114
insertKnots.m, 142
insertOneKnot.m, 144

k
E_back.m, 114
Green.m, 126
igabem_single_cnt.m, 134
k0
changeAngle.m, 104
changeDistance.m, 108
igabem_single_cnt.m, 134
info_curve.m, 141
test_optimize.m, 172
test_optimize2.m, 176
kernel_analytic

info_curve.m, 140
test_calc.m, 158
test_calc_multiple.m, 162
test_optimize.m, 170
test_optimize2.m, 174
kernel_analytic.m, 145
function, 145
L, 145
lambda, 145
nargin, 145
res, 145
s, 145
y, 145
kernel_numeric
changeAngle.m, 102
info_curve.m, 140
p_sctr.m, 155
test_calc.m, 158
test_calc_multiple.m, 162
kernel_numeric.m, 146
e, 146
function, 146
g, 146
isempty, 146
nargin, 146
p, 147
res, 147
y, 147
knots
freeform_geom.m, 115
freeform_geom_withArea.m, 118
freeform_geom_wo_pers.m, 119
freeform_x_sym1_withArea.m, 122
new_freeform.m, 148
knots1
freeform_x_sym1_withArea.m, 122
freeform_x_sym_withArea.m, 125
knotsIn
insertKnots.m, 142
insertOneKnot.m, 144

L
changeDistance.m, 108
dataFunc.m, 112
kernel_analytic.m, 145
p_sctr.m, 156
test_calc.m, 160
test_calc_multiple.m, 163, 165
l
countSteps.m, 109
insertKnots.m, 143
insertOneKnot.m, 144
L2
igabem_circle_refinement.m, 129, 130
L_plus_r
igabem_single_cnt.m, 135
lambda
changeDistance.m, 108
dataFunc.m, 112
dataKernelNumeric.m, 113
kernel_analytic.m, 145
p_sctr.m, 156
test_calc.m, 160
test_calc_multiple.m, 165
lb
test_optimize.m, 172
test_optimize2.m, 176
LHS
igabem_single_cnt.m, 133, 135
Linf
igabem_circle_refinement.m, 129, 130

max
igabem_circle_refinement.m, 129
max_r
freeform_geom_withArea.m, 118
freeform_x_sym1_withArea.m, 122
freeform_x_sym_withArea.m, 125
max_r_per
new_freeform_unclamped.m, 150
multiplier
changeAngle.m, 104
changeDistance.m, 108
info_curve.m, 141
p_sctr.m, 157
test_calc.m, 160
test_calc_multiple.m, 165
test_optimize.m, 172
test_optimize2.m, 176
Mx
curve_centroid.m, 111
My
curve_centroid.m, 111

N
igabem_single_cnt.m, 135
n
changeAngle.m, 104
changeAngleNewParam.m, 106
changeDistance.m, 108
halfSpike.m, 128
IncreaseDoFsInModel.m, 138
newIncrDoF.m, 152
test_newIncrDoF.m, 168
n0
freeform_x_sym1_withArea.m, 122
freeform_x_sym_withArea.m, 125
nargin
areaconstraint.m, 100
freeform_geom.m, 116
IncreaseDoFsInModel.m, 138
kernel_analytic.m, 145
kernel_numeric.m, 146
new_freeform.m, 148
new_freeform_unclamped.m, 150
p_sctr.m, 157
new_curve
insertKnots.m, 143

insertOneKnot.m, 144
newIncrDoF.m, 152
test_newIncrDoF.m, 169
new_freeform
test_newIncrDoF.m, 167
new_freeform.m, 147
ctrl_pts, 148
geom, 148
i, 148
isempty, 147
knots, 148
nargin, 148
phi, 147, 148
r, 148
sin, 148
new_freeform_unclamped.m, 149
ctrl_pts, 150
i, 150
isempty, 149
max_r_per, 150
nargin, 150
phi, 149, 150
r, 150
sin, 149
unclgeom, 150
new_geom
changeAngleGeom.m, 105
newIncrDoF.m, 152
test_newIncrDoF.m, 169
new_geom2
test_newIncrDoF.m, 169
new_params
changeAngleNewParam.m, 106, 107
newIncrDoF.m, 151, 153
oldToNew.m, 154
newIncrDoF.m, 151
ctrl_points, 152
ctrl_pts, 152
curve, 152
geom, 152
i, 152
isempty, 151
n, 152
new_curve, 152
new_geom, 152
new_params, 151, 153
phi, 151, 153
r, 153
radii, 153
sin, 152
nIter
freeform_x_sym1_withArea.m, 122
nonlcon
test_optimize.m, 172
test_optimize2.m, 176
nPoints
igabem_single_cnt.m, 135
nrbctrlplot
halfSpike.m, 127
info_curve.m, 140
p_sctr.m, 156
test_freeform.m, 166
test_newIncrDoF.m, 167
test_optimize.m, 171
test_optimize2.m, 175
nSteps
igabem_circle_refinement.m, 131
Nt
igabem_single_cnt.m, 135
nu
E_back.m, 114
Green.m, 126
NURBStoolbox
halfSpike.m, 128
test_optimize.m, 172
test_optimize2.m, 176

oldToNew.m, 153
ctrl_pts, 154
i, 154
new_params, 154
phi, 154
r, 154
radii, 155
sin, 154
omega
igabem_single_cnt.m, 135
opt_param
test_optimize.m, 172
test_optimize2.m, 176
options
test_optimize.m, 172
test_optimize2.m, 176
opts
IncreaseDoFsInModel.m, 138
order
insertOneKnot.m, 143

p
kernel_numeric.m, 147
p_sctr.m, 157
test_calc.m, 160
test_calc_multiple.m, 165
p1
igabem_single_cnt.m, 135
p2
IncreaseDoFsInModel.m, 138
p_sctr
test_optimize.m, 171
test_optimize2.m, 175
p_sctr.m, 155
area, 156
curve, 156
disp, 155
e_numerical, 156
figure, 156
h0, 156

kernel_numeric, 155
L, 156
lambda, 156
multiplier, 157
nargin, 157
nrbctrlplot, 156
p, 157
phi0, 157
r, 157
scat_numerical, 157
step, 157
t, 157
title, 156
x, 157
p_sctr_val
test_optimize.m, 172
test_optimize2.m, 176
params
freeform_x_sym_withArea.m, 124
test_newIncrDoF.m, 168
params2
IncreaseDoFsInModel.m, 137, 138
phi
changeAngleNewParam.m, 106, 107
freeform_geom.m, 116
freeform_geom_withArea.m, 118
freeform_geom_wo_pers.m, 119
freeform_x_sym1_withArea.m, 122
freeform_x_sym_withArea.m, 125
new_freeform.m, 147, 148
new_freeform_unclamped.m, 149, 150
newIncrDoF.m, 151, 153
oldToNew.m, 154
phi0
igabem_circle_refinement.m, 131
p_sctr.m, 157
test_calc.m, 160
test_calc_multiple.m, 165
phiArr
info_curve.m, 141
place
insertOneKnot.m, 144
plot
changeAngle.m, 102
test_calc.m, 159
test_calc_multiple.m, 163
pts2
freeform_geom_withArea.m, 118

quad1
igabem_single_cnt.m, 135
quad2
igabem_single_cnt.m, 135

r
changeDistance.m, 108
dataFunc.m, 112
freeform_geom.m, 116
freeform_geom_wo_pers.m, 119
freeform_x_sym_withArea.m, 124
igabem_circle_refinement.m, 131
new_freeform.m, 148
new_freeform_unclamped.m, 150
newIncrDoF.m, 153
oldToNew.m, 154
p_sctr.m, 157
test_calc.m, 160
test_calc_multiple.m, 163, 165
r_max
changeDistance.m, 108
dataFunc.m, 112
test_freeform.m, 166
test_newIncrDoF.m, 168
radii
newIncrDoF.m, 153
oldToNew.m, 155
test_newIncrDoF.m, 169
rArr
info_curve.m, 141
ratio
changeAngle.m, 104
test_optimize.m, 173
test_optimize2.m, 177
RBases
igabem_single_cnt.m, 135
re
test_calc_multiple.m, 165
real
test_calc.m, 159
test_calc_multiple.m, 163
res
E_back.m, 114
freeform_geom_withArea.m, 118
freeform_x_sym1_withArea.m, 121, 123
freeform_x_sym_withArea.m, 124, 126
Green.m, 126
igabem_single_cnt.m, 136
kernel_analytic.m, 145
kernel_numeric.m, 147
return
freeform_x_sym1_withArea.m, 123
Rt
igabem_single_cnt.m, 136

s
igabem_circle_refinement.m, 131
kernel_analytic.m, 145
scat_analytic
changeAngle.m, 104
info_curve.m, 141
test_calc.m, 160
test_calc_multiple.m, 164
test_optimize.m, 173
test_optimize2.m, 177
scat_numerical
changeAngle.m, 102
info_curve.m, 141
p_sctr.m, 157

test_calc.m, 160
test_calc_multiple.m, 164
sector
test_optimize.m, 173
test_optimize2.m, 177
sigma
changeDistance.m, 108
dataFunc.m, 112
test_calc.m, 161
test_calc_multiple.m, 164, 165
sigma_val
igabem_single_cnt.m, 136
sin
freeform_geom.m, 115
freeform_geom_withArea.m, 117
freeform_geom_wo_pers.m, 119
freeform_x_sym1_withArea.m, 121
new_freeform.m, 148
new_freeform_unclamped.m, 149
newIncrDoF.m, 152
oldToNew.m, 154
size
igabem_circle_refinement.m, 129
solution
igabem_single_cnt.m, 136
step
changeDistance.m, 108
dataFunc.m, 112
p_sctr.m, 157
test_calc.m, 161
test_calc_multiple.m, 165
steps
countSteps.m, 110

t
p_sctr.m, 157
test_calc.m, 161
test_calc_multiple.m, 166
t0
igabem_circle_refinement.m, 131
igabem_single_cnt.m, 136
test_calc.m, 158
__pad0__, 159
circle, 159
e_analytic, 159
e_numerical, 159
figure, 160
h0, 160
imag, 158
kernel_analytic, 158
kernel_numeric, 158
L, 160
lambda, 160
multiplier, 160
p, 160
phi0, 160
plot, 159
r, 160
real, 159
scat_analytic, 160
scat_numerical, 160
sigma, 161
step, 161
t, 161
test_calc_multiple.m, 161
__pad0__, 164
circle, 164
e_analytic, 164
e_numerical, 164
fprintf, 162
h0, 164
i, 164
im, 165
imag, 162
kernel_analytic, 162
kernel_numeric, 162
L, 163, 165
lambda, 165
multiplier, 165
p, 165
phi0, 165
plot, 163
r, 163, 165
re, 165
real, 163
scat_analytic, 164
scat_numerical, 164
sigma, 164, 165
step, 165
t, 166
test_freeform.m, 166
curve, 166
nrbctrlplot, 166
r_max, 166
tiledlayout, 166
test_newIncrDoF.m, 167
figure, 168
i, 168
n, 168
new_curve, 169
new_freeform, 167
new_geom, 169
new_geom2, 169
nrbctrlplot, 167
params, 168
r_max, 168
radii, 169
x0, 168, 169
test_optimize.m, 169
A, 171
Aeq, 171
areaconstraint, 170
b, 171
beq, 171
clear, 171
disp, 170
fun, 171

geom, 172
h0, 172
k0, 172
kernel_analytic, 170
lb, 172
multiplier, 172
nonlcon, 172
nrbctrlplot, 171
NURBStoolbox, 172
opt_param, 172
options, 172
p_sctr, 171
p_sctr_val, 172
ratio, 173
scat_analytic, 173
sector, 173
ub, 173
test_optimize2.m, 173
A, 175
Aeq, 175
areaconstraint, 174
b, 175
beq, 175
clear, 175
disp, 174
fun, 175
geom, 176
h0, 176
k0, 176
kernel_analytic, 174
lb, 176
multiplier, 176
nonlcon, 176
nrbctrlplot, 175
NURBStoolbox, 176
opt_param, 176
options, 176
p_sctr, 175
p_sctr_val, 176
ratio, 177
scat_analytic, 177
sector, 177
ub, 177
x0, 177
th
changeAngleGeom.m, 105
tiledlayout
test_freeform.m, 166
title
changeAngle.m, 103
p_sctr.m, 156
tmpgeom
freeform_geom_withArea.m, 118
freeform_x_sym1_withArea.m, 123
freeform_x_sym_withArea.m, 126
trapz
igabem_circle_refinement.m, 130
true
igabem_single_cnt.m, 133

ub
test_optimize.m, 173
test_optimize2.m, 177
unclgeom
new_freeform_unclamped.m, 150
uniqueKnots
countSteps.m, 110
insertKnots.m, 143
insertOneKnot.m, 144

val
curve_area.m, 110
IncreaseDoFsInModel.m, 138
vals
igabem_circle_refinement.m, 131

x
areaconstraint.m, 100
p_sctr.m, 157
x0
changeAngle.m, 104
halfSpike.m, 127, 128
info_curve.m, 141
test_newIncrDoF.m, 168, 169
test_optimize2.m, 177
x1
changeAngle.m, 104
info_curve.m, 141
x11
changeAngle.m, 104
info_curve.m, 142
x2
changeAngle.m, 104
info_curve.m, 142

y
areagreen.m, 101
curve_area.m, 110
curve_centroid.m, 111
kernel_analytic.m, 145
kernel_numeric.m, 147