May 2020

# A Method of Topology Optimization for Curvature Continuous Designs

Jack Steven Rossetti
*Syracuse University*

# Abstract

Recently, there have been many developments made in the field of topology optimization. Specifically, the structural dynamics community has been the leader of the engineering disciplines in using these methods to improve the designs of various structures, ranging from bridges to motor vehicle frames, as well as aerospace structures like the ribs and spars of an airplane. The representation of these designs, however, are usually stair-stepped or faceted throughout the optimization process and require post-process smoothing in the final design stages. Designs with these low-order representations are insufficient for use in higher-order computational fluid dynamics methods, which are becoming more and more popular. With the push for the development of higher-order infrastructures, including higher-order grid generation methods, there exists a need for techniques that handle curvature continuous boundary representations throughout an optimization process.

Herein a method has been developed for topology optimization for high-Reynolds number flows that represents smooth bodies, that is, bodies that have continuous curvature. The specific objective function used herein is to match specified x-rays, which are a surrogate for the wake profile of a body in cross-flow. The parameterized level-set method is combined with a boundary extraction technique that incorporates a modified adaptive 4th-order Runge-Kutta algorithm, together with a classical cubic spline curve-fitting method, to produce curvature-continuous boundaries throughout the optimization process. The level-set function is parameterized by the locations and coefficients of Wendland C2 radial basis functions. Topology optimization is achieved by implementing a conjugate gradient optimization algorithm that simultaneously changes the locations of the radial basis function centers and their respective coefficients. To demonstrate the method several test cases are shown where the objective is to generate a smooth representation of a body or bodies that match specified x-rays. First, multiple examples of shape optimization are presented for different topologies. Then topology optimization is demonstrated with an example of two bodies merging and several examples of a single body splitting into separate bodies.

# A Method of Topology Optimization for Curvature Continuous Designs

by

Jack S. Rossetti

B.S., SUNY University at Buffalo, 2014
M.S., Syracuse University, 2016

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Mechanical and Aerospace Engineering.

Syracuse University
May 2020

*I dedicate this work to the loved ones that I have lost, for always believing in me and giving me the courage and confidence to pursue my dreams; for being my light it times of darkness and for helping me persevere. These loved ones are my great-grandmother ("G.G.") Sarah Cooper, grandmother ("Nan") Rose Rossetti, and grandfather ("Papa") Vincent Rossetti. Their memories live on in the loved ones they've left.*

# Acknowledgments

I would like to thank my advisers, Dr. John F. Dannenhoffer III and Dr. Melissa A. Green for their guidance throughout my dissertation research. Dr. Dannenhoffer was an incredible resource and was always able to keep me moving forward, even though I tend to meander. Dr. Green was extremely supportive and helped expand my interests into the biomimetic field, which was seemingly unrelated to my research, but upon closer inspection, some direct connections could be made. Furthermore, both of you were always receptive to me when I had any doubts or issues throughout my studies. Thank you for cheering me on and keeping me focused. The two of you have made my time at Syracuse University unforgettable. Additionally, Dr. Mark Glauser also deserves recognition and acknowledgment because he was my first research adviser from Syracuse back in the Summer of 2013. That Summer was the reason I decided to pursue a Ph. D. and continue academic research in mechanical and aerospace engineering. Thank you, Mark, for opening my eyes to the fun and rewarding side of research.

To my defense committee chair and other members, Dr. James H. Henderson, Dr. Ben Akih-Kumgeh, and Dr. Utpal Roy: thank you for your time and patience throughout the defense process. Your insights and suggestions that helped shape this document were greatly appreciated. I would like to also thank the engineering staff who I have interacted with throughout my time at Syracuse University. I would not have been able to complete my work and studies without the help of each of you along the way. Of note are: Kathleen Joyce, Terrie Monto, Kathy Madgian, Kristen Shapiro, Kelly Jarvi, and Jim Spoelstra.

To the gentlemen I lived with in the Birdcage; John-Michael Velarde, Justin King, Matt Berry, Andy Magstadt, Jake Connors, Zach Eager, and TJ Coleman (briefly): thank you for all the intellectual (and unintelligible) conversations we had over the years that helped me figure out a problem, help one of you figure something out (just as rewarding), and de-stress from long arduous research days. The Cage will always hold fond memories of my graduate experience. I'd also like to thank DeShawn Coombs, Andrew Tenney, and Justin

King (again) for going through the Ph. D. process side-by-side with me. Thanks for sharing my pain and struggles, along with the joys and triumphs. Also, Ryan Falkenstein-Smith, thank you for keeping me sane with your “pep” talks and dad jokes.

To my family, thank you for dealing with me and trying to understand what my research was all about. Dad, you always lifted me up when I felt down. Your continued support helped me get through the hard, deflating times during my degree. Ma, you were always quick to call or message me to see how I was doing — for making sure I was happy and unencumbered throughout my studies. Your ability to surprise me with visits (from several hours away by car) never ceased to amaze me and were always enjoyed. To my brother, Michael, you were always an inspiration to me and drove me to work hard and achieve great things. No, I’m not giving you credit for my degree, relax.

Finally, to my one true love, Oberyn, my Husky pup. He has been the light that has kept me going since the third year of my degree. After realizing that I could sit in one place for more than 18 hours (one of many skills my studies have taught me), I knew I needed a furry companion to help me break up the day, exercise, and all around improve my attitude daily. While he’s a knucklehead most of the time, there are some glimmers of affection that he will show me and my fiancée. Those moments are the most cherished. He has been and will continue to be an incredible joy in my life.

Seriously, because I know she just rolled her eyes, to my love, my support-system, my everything, Stephanie. I don’t know how I would have made it through any of this without you. Honestly, I cannot figure out why you’re still hanging around me after all the late-night studying, barely traveling, and constant stressing. I guess it must be my cooking. There are no words that can describe how grateful I am that you came into my life during this time period. You have helped me through all the craziness and stayed by my side through all the delays and extensions. Thank you. I cannot wait to see what future adventures are coming our way.

And last, but cetainly not least, thank You, G-d, for all the gifts You have given me.

# Contents

# List of Figures

# Nomenclature

## Acronyms and Abbreviations

| | |
|---|---|
| $CFD$ | Computational fluid dynamics |
| $CSRBF$ | Compactly-supported RBF |
| $CT$ | Computer-aided tomography |
| $GSRBF$ | Globally-supported RBF |
| $LBM$ | Lattice Boltzmann Method |
| $LSM$ | Level-set method |
| $LSF$ | Level-set function |
| $MATLAB$ | Matrix Laboratory |
| $RAMP$ | Rational Approximation of Material Properties |
| $RANS$ | Reynolds-Averaged Navier-Stokes |
| $RBF$ | Radial basis function |
| $Re$ | Reynolds number |
| $RK$ | Runge-Kutta |
| $RK4$ | 4th-order RK algorithm |
| $RMS$ | Root-mean-square |
| $SIMP$ | Simplified Isotropic Material with Penalization |

## Functions and variables

| | |
|---|---|
| $C^n$ | $n$-degree of continuity of a curve or function |
| $d_{i,RBF}$ | Distance between neighboring RBFs |
| $dx_{ray}$ | Spacing between each of the vertical rays |
| $dy_{ray}$ | Spacing between each of the horizontal rays |
| $e$ | Error in CFD simulation |

| | |
|---|---|
| $F$ | Extension velocity |
| $f(\,)$ | An arbitrary function |
| $h$ | Grid resolution/step size |
| $h_{ray}$ | Horizontal x-ray value |
| $I$ | Interval of uncertainty |
| $N_{SR}$ | User-specified multiplier for determining the SR |
| $O$ | Objective function |
| $\vec{p}$ | Design parameters |
| $RMS_{r,avg}$ | RMS of the calculated radius and the average radius |
| $r$ | The radius of a curve or function |
| $s$ | Distance along the curve |
| $SR$ | The support radius of the RBF |
| $t$ | Parametric coordinate |
| $T$ | Time |
| $v_{ray}$ | Vertical x-ray value |
| $\vec{x}$ | Euclidean spatial coordinates |
| $\vec{x}_i$ | The coordinate of the ith RBF |
| $\vec{x}_{int}$ | Coordinate of ray intersection point |
| $x^*$ | Modified coordinate after linear smoothing |
| $\alpha_i$ | The coefficient of the ith RBF |
| $\beta$ | Conjugate gradient multiplier |
| $\gamma$ | Step size along the search direction |
| $\delta$ | Scaling parameter used to increment the step size |
| $\eta$ | Distance from the level-set curve to the current point calculated via the RK4 |
| $\kappa$ | Parametric curvature |
| $\xi$ | Radial distance from an RBF center divided by its SR |
| $\phi$ | The LSF |

| | |
|---|---|
| $\tilde{\phi}$ | Shifted LSF |
| $\psi_i$ | The ith RBF |

## Superscripts

| | |
|---|---|
| $h$ | Terms associated with one full step of the RK4 algorithm |
| $\frac{h}{2}$ | Terms associated with two half steps of the RK4 algorithm |
| $k$ | The order of a CFD scheme |

## Subscripts

| | |
|---|---|
| 0 | Initial state |
| $avg$ | Average quantity |
| $DES$ | Terms associated with the desired x-rays |
| $eval$ | Objective function evaluations |
| $int$ | Terms associated with the ray intersection points |
| $iter$ | Iterations |
| $opt$ | Optimal state |
| $ray$ | Terms associated with the x-ray values |
| $reinit$ | Reinitializations |
| $spline$ | Terms associated with the spline points |

## Operators

| | |
|---|---|
| $\|(\ )\|$ | Absolute value |
| $\|(\ )\|$ | $L_2$-norm of a vector |

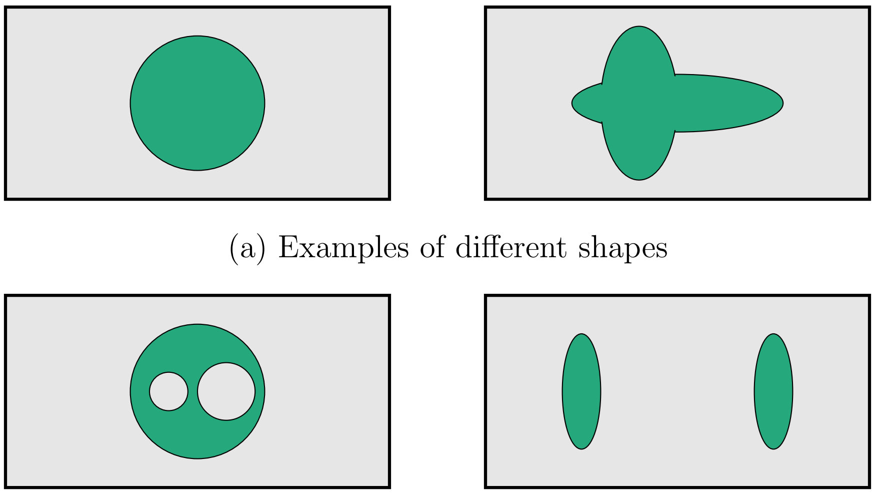
| | |
|---|---|
| $\nabla(\,)$ | Gradient of a function |
| $\bar{(\,)}$ | Adjoint mode derivative |
| $\dot{(\,)}$ | Tangent linear derivative |
| $\frac{\partial(\,)}{\partial\vec{p}}$ | Partial derivative with respect to the design parameters |
| $\frac{\partial(\,)}{\partial\vec{s}}$ | Partial derivative with respect to distance along a curve |
| $\frac{\partial(\,)}{\partial T}$ | Partial derivative with respect to time |
| $\frac{\partial(\,)}{\partial\vec{x}}$ | Partial derivative with respect to Euclidean spatial coordinates |

# Chapter 1

# Introduction

Creating and the act of creation are second-nature to human-beings. Everything we do as a collective society and species aims to create or make something that can be used to perform a task or an operation. Design is the study of defining a procedure for making things that meet various requirements while acknowledging potential trade-offs. Consequently, the question of how to obtain the best or optimal design became a topic of interest that persists today. Note that design is a vague term and can be used to define anything from computational algorithms to physical objects and all that exists in between. This research focuses on the design of physical objects, in particular, aerodynamic bodies, and how to develop and improve current techniques so more complex and intricate designs can be realized. The main approach to design optimization in the aerodynamics community has been shape optimization, yet a more general optimization technique referred to as topology optimization has been growing in popularity elsewhere in the engineering community. Before discussing the differences between the two techniques, it is worth explaining the difference between the shape and topology of a design.

Shape and topology are two pieces of the identifying characteristics of an object or physical design. Shape refers to specific boundary representation while topology refers to the number of boundaries contained within a design. The collective boundaries of interest that


(a) Examples of different shapes

(b) Examples of different topologies

Figure 1.1: Comparison of shape and topology

make up the various shapes and topologies in this work will be referred to as *the design*. Fig. 1.1 shows four examples of different designs for reference: two shapes and two topologies. In these examples the boundaries are defined as the black curves separating the green area from the gray area. Fig. 1.1a portrays the design of a circle on the left and a design created by the intersection of two ellipses on the right. It is easy to recognize that these designs are different because their boundaries are different and therefore, they have different shapes. As mentioned previously, the boundary representation is the definition of a shape. However, the topologies of these two designs are the same. In each domain (the gray rectangles) the design has only one boundary associated with it. In contrast, the designs in Fig. 1.1b have different topologies from each other as well as the designs shown in Fig. 1.1a. Each domain has multiple boundaries within them; for instance, the lower left design is a circle with two internal holes resulting in three distinct boundaries while the lower right design contains two ellipses, which means there are two distinct boundaries in the design. Due to the different number of boundaries contained in these designs, they are said to be topologically different. Fig. 1.1b also illustrates two different ways of interpreting the topology of a design, where in the left domain the topology is determined by the number of holes within the circle and in the right domain the topology is determined by the number of objects in the design domain. The latter of these two interpretations is used for this research. Now, with a better understanding of the differences in shape and topology, the idea of optimizing the shape or topology of a design can be discussed.

Methods to find the best or optimal engineering application designs have been continually developed depending on need, as described below. Before computers were developed and made affordable, a common optimization technique was trial-and-error. This is the process of building or modeling conceptual designs and testing them to see how well they perform according to some metric. The trial and error process can be very time-consuming and monetarily expensive, so numerical design optimization techniques have been developed to mitigate these costs. The most popular design optimization technique used in the engineering field is shape optimization. The idea is similar to trial and error design optimization in the sense that one begins with an initial design and simulates its performance under design conditions. Then the shape and/or topology is changed and tested again to quantitatively measure improvement until an optimal design is found. Examples of shape optimization can be found in aerodynamic engineering areas such as the design of airfoils[1], aeroacoustic design[2], drag reduction[3,4,5,6], and many others. There is a question, however, that both shape optimization and trial-and-error methods fail to address. Take, for example, a multi-component wing section - how does one determine how many components should be in the final design? For trial and error optimization, luck or coincidence is needed to find the answer because the design space is immense. Shape optimization techniques require an initial number of components that is very difficult to change throughout the design procedure and usually remains the same throughout the shape optimization process.

This specific challenge is being addressed by the development of topology optimization techniques. In this work, the topology of a design is the number of components present in the design domain. These components can be either bodies/objects or holes. Topology optimization can then be seen as a method for finding the optimal distribution and shape of components within a design. These methods do not require a predetermined design topology to find an optimal design and can be viewed as general design optimization techniques. Topology optimization was originally developed as a method for finding optimal designs for static structures under loading, such as beams and bridges. The constraints were related to the compliance of the design and the objective was to minimize the mass. After being investigated in the structures community, the technique began gaining popularity in the fluid dynamics field with applications in channel design and microfluidic chip design. For the above applications, the boundary curvature does not need to be continuous and slope variations do not need to be controlled until after the optimization has completed. This is also true for structural dynamics problems because the equations are linear and stress concentrations occur at corners and cusps. For fluid dynamics applications, the boundary information is not important at low Reynolds numbers because the fluid flow can be modeled using the Stokes flow approximation. This means the momentum of the fluid dissipates quickly into the field and small perturbations and discontinuities in the boundaries do not have a significant effect on the fluid flow properties. However, curvature and slope are very important for high Reynolds number flows and can cause the design to perform poorly if not properly treated. Furthermore, there have been proven benefits to using curvature continuous geometries and grids for improved computational fluid dynamics (CFD) simulations[7,8]. In view of this, a new technique has been developed to perform topology optimization for curvature continuous designs that can be used in high Reynolds numbers ($\mathrm{Re} \geq 10000$) fluid dynamics simulations.

Developing a topology optimization scheme for designs in high Reynolds number flow regimes can have an impact on the design of future aerospace vehicles as well as internal flow systems. Allowing the number of components to vary throughout the optimization can result in novel designs that would not have been considered based on previous experience or intuition. Examples of the potential applications for this method include: multi-component airfoil design, grid fins for rockets and projectiles like those of the SpaceX Falcon-series, turbine blade configuration, and turning vane distribution and design in a flow bend. Efforts to improve the design of these complex systems may result in operating-cost savings by reducing the pressure-loss or energy dissipation across the system. The work presented here focuses on the handling of the design boundaries and ensuring that curvature continuity is maintained. That being said, a surrogate for a CFD simulation is used for investigating the geometry-handling which is the main contribution of this work. The research and algorithms were performed and written such that the work here can be easily adapted and used in conjunction with an open-source high-fidelity CFD solver such as SU2, which is discussed in the conclusions and future work section.

The optimization scheme developed from this research is briefly described here and is expanded upon in Ch. 4. The goal of the scheme presented in this document is to match the width and height distributions of a level-set topology to the respective distributions of a given topology. The level-set topology is represented by the level-set curve for which the level-set function (LSF) is equal to zero. The width and height distributions are generated using a raycasting algorithm and are therefore labeled as *the x-rays* of the topology in this work. Since the design topology is the focus of the optimization and the x-rays are generated from the topology, they are referred to as secondary data. The developed method uses a two-stage process that combines a heuristic optimization to identify pixelated or stair-stepped representations for the initial boundaries followed by a continuous, gradient-based optimization using the level-set method (LSM) for sensitivity calculations. The approach is to obtain the initial boundaries from given secondary data and then input this initial guess into a gradient-based optimization tool that can deform the boundaries freely as well as produce topological changes such as merging or tearing boundaries. Topology optimization is performed by changing the parameters of a LSF, which is a variant of the LSM. In particular, the parameters are the locations and coefficients of radial basis functions (RBFs). The LSM was chosen for two reasons – it represents the design boundaries implicitly so topological changes such as merging and tearing of boundaries can occur easily throughout the optimization procedure and these boundaries can be curvature continuous by careful definition of the level-set parameters. The root-mean-square (RMS) of the differences between the x-rays of the level-set topology and the desired ones are calculated and added together to produce the objective function value. Minimizing this objective value will have the effect of driving the x-rays of level-set topology to match the desired ones. This particular objective function was chosen deliberately as a surrogate for matching the wake behind an object in cross flow. The aim is to provide a foundation for this technique so it can be extended to fluid flow applications. Further investigations in topology optimization using a LSM can then be performed to see if the boundary slopes and curvatures can be controlled for accurate and reliable CFD results that can generate an optimum design.

The document is set up to first give a review of previous topology optimization techniques; a discussion of the LSM along with its general applications and its use for topology optimization, and then the set up of the optimization problem is presented. This is followed by the results and accompanying discussion which leads into the conclusions with a vision for the future of this work and some areas that require further development.

# Chapter 2

# Topology optimization in literature

Within the last three decades, topology optimization has been developed as a design optimization tool that creates novel designs by adding and removing bodies/holes in a design where it is necessary. Optimal designs such as the Generico Chair[9], shown in Fig. 2.1, have been created using topology optimization. The method enables unconventional designs to be realized by allowing intricate details and shapes to be formed as seen in the Generico chair. Furthermore, allowing the topology to change throughout the design processes reduces the need for intuition or experience to produce a good initial guess.

## 2.1 Applications in structural dynamics

Initially, topology optimization was applied to structural problems with mass and loading constraints. The seminal paper by Bendsøe[10] introduced the method using a homogenization approximation of the density field of the design to create a binary optimization problem. Topology optimization techniques have been developed over the years and now there exist several approaches: density-based[11–13], evolutionary[14–16], phase field[17], topological derivatives[18], and explicit/implicit level-set methods (LSMs)[19–21]. Moreover, hybrid methods have been developed by combining aspects of each approach, such as using shape derivatives with the LSM to produce topological changes without topological derivatives[22]. Another tech-

Figure 2.1: The Generico Chair[9]

nique becoming more popular is the use of filtered density fields in projection methods that share similarities with the level-set function (LSF) and LSM[23, 24, 25, 26].

The first approaches to topology optimization of structures was to discretize the domain into grid cells and set the density of every grid cell to either one or zero. However, Sigmund and Petersson[27] have shown that there is a lack of solutions for these types of problems. Setting up the problem in this way can be pathological when the goal is to minimize a scalar field, such as the structural compliance, by changing the densities within the grid while satisfying a constraint on the total volume of the design. The issue seen in the discrete setup for structural dynamics problems is often referred at as the checkerboard problem[28, 29]. Despite these findings, researchers still endeavored to develop gradient-free (referred to as discrete) optimization techniques to solve the problem using the discrete setup.

Discrete optimization boasts the ability to find global minima because of it's heuristic nature and large sweep of the design domain. These attributes come at a cost because discrete methods are often very computationally expensive due to the large number of objective function evaluations. Topology optimization is no different and only problems with a small number of design variables have been solved to global optimality, shown by Stolpe and Bendsøe[30]. Furthermore, Sigmund[31] published arguments against using such methods for these types of problems using the above as evidence. Since gradient-free optimization struggles for some topology optimization problems, gradient-based approaches began growing in number.

In general, gradient-based topology optimization can be viewed in two ways: a density field optimization, or a shape optimization. These approaches have been referred to as the Eulerian (fixed mesh) and Lagrangian (boundary following mesh) techniques[32]. The Eulerian approaches are more common in topology optimization applications. After the homogenization approach was presented, the Simplified Isotropic Material with Penalization (SIMP) or power-law technique was introduced to simplify the problem and to improve convergence to discrete solutions[27,12,13]. Bendsøe and Sigmund[33] reported that the SIMP method has a physical justification that effectively relates the density design variable to the material properties. Stolpe and Svanberg[34] proposed the Rational Approximation of Material Properties (RAMP) method as variation of SIMP to mitigate the objective function convexity and concavity issues within the SIMP scheme[35], ensuring convergence to discrete 0-1 solutions.

The SIMP methodology has three main approaches that are used in current research: one-field[36], two-field[37,38], and three-field[23,24,25]. The each different technique uses the density field or some combination of a pseudo-density field and other projections to solve the topology optimization problem. One-field uses the density field as the design variables, two-field uses a design parameter field that influences the density field, and three-field uses design fields, density fields, and a projection field. The supplementary fields being used are successively filtered or smoothed to produce continuously varying fields for the optimization process, basically attempting to convert the discrete problem into a continuous one.

The use of topological derivatives for topology optimization problems was originally referred to as the bubble method by Eschenauer *et al.*[39]. The idea was to predict how adding a hole would influence the objective function. Essentially, this was a method for calculating the sensitivities of a design to topological changes at any point in the design domain. These derivatives can be quite complex and require high-level mathematics for their derivation[18,40]. While these derivatives may aide in hole placement, it is still unclear whether they are useful or if randomly placed holes can achieve a similar outcome[41]. Hole creation, or nucleation, is still a difficult task to perform and is discussed further in a subsequent section.

## 2.2 Initial applications in fluid dynamics

As topology optimization methods grew in popularity in the structural dynamics field it also began sparking research interest in the fluid dynamics community. Borrvall and Petersson[42] applied SIMP and RAMP schemes to fluid dynamics problems governed by the Stokes equations where pressure loss was the objective being minimized. Additionally, these researchers investigated defining the objective function as the energy dissipation across the system for similar flow regimes. Other studies regarding topological derivatives and their use in shape optimization problems governed by the Stokes equations were pursued by Guillaume and Idris[43]. Furthermore, Aage*et al.*[44] studied large-scale 2D and 3D Stokes flow problems using parallel computations and Abdelwahed and Hassine[45] presented theoretical results for 2D and 3D cases for Stokes flow that were shown to be valid for a large class of objective functions. Clearly, advances in topology optimization were being made by the fluids community, but only for low Reynolds number regimes.

These initial investigations into topology optimization for fluid dynamics applications were aimed at the design of microfluidic devices and used a similar framework as the structural dynamics community. The design domain was considered to be either filled with material or empty and the goal of the optimization scheme was to minimize either the dissipated energy or pressure loss across the domain with a constrained volume fraction for the fluid domain compared to the solid domain. Material was either added or removed to produce the desired Stokes flow solution. Modifications have been made to the Stokes equations to include Darcy's law regarding porosity to better apply Eulerian topology optimization techniques to fluid flow problems.

Guest and Prévost[46] studied the Darcy-Stokes equations as the governing equations for topology optimization applications as a means of using porosity of mesh elements in place of their density. This drew a direct connection between the design variables and the governing equations. Guest and Prévost[47] continued investigations into periodic, porous material micro-structures using this technique and setup.

Stokes and Darcy-Stokes flow share similarities with the initial structural dynamics applications in that the equations being solved were linear. Linearity of the governing equations results in more robustness in the physics solver because design features such as sharp corners or cusps are less likely to cause the fluid flow solver to diverge. However, applying these techniques using the Navier-Stokes equations for large Reynolds numbers ($Re \geq 10000$) presents difficulties because of the nonlinear terms. Considerable assumptions and simplifications need to be used for the application of topological optimization techniques using the Navier-Stokes equations. Usually the Reynolds number is low and the flow is assumed to be laminar. Using these assumptions, Gersborg-Hansen *et al.*[48] studied devices with velocity-driven switches governed by the Navier-Stokes equation using mathematical-programming and analytical derivatives.

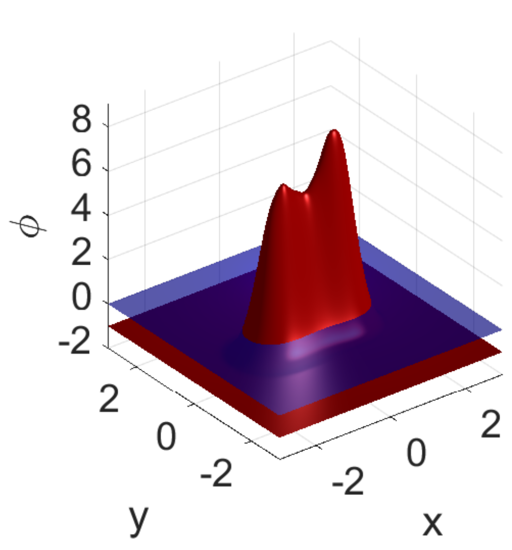
The main focus of research for topology optimization in fluids has been directed towards the design of channels. For instance, Gersborg-Hansen *et al.*[49] investigated matching the desired outflow rate in both 2D and 3D for Stokes flow applications as an alternative to using pressure loss and energy dissipation across the system as objective functions. However, little work on the design of the objects within a fluid flow such as airfoil- or wing-like designs that are seen in flow bends, or turbine engines. The work presented here is an initial step towards a fluid dynamics topology optimization scheme that focuses on the design topology of the internal components (i.e., the turning vanes or turbine blades) that interact with the fluid within a defined flow domain. The goal of this work is to produce a topology optimization scheme that maintains curvature continuous boundaries and can be easily combined with a high-fidelity flow solver to investigate topology optimization applications for high Reynolds number flows given the desired outflow profile. However, there is a lack of research presently that discusses or investigates higher-order boundary representations for fluid dynamics topology optimization applications.

The research and investigations discussed until this point use Eulerian approaches that do not emphasize boundary representation. Additionally, the Eulerian approach is only beneficial when the connection between fluid flow and material boundary does not have a profound effect on the flow characteristics, which is the case for low Reynolds number, Stokes flow, and Darcy-Stokes flow applications. As an alternative, a LSM for fluid dynamics topology optimization was proposed by Challis and Guest[50] that was shown to be more efficient than the density-based, Eulerian approaches due to the implementation of the no-slip boundary condition and reduced computational cost because only the fluid regions needed to be modeled. Using this approach, the boundary(ies) can be directly manipulated instead of changing the density or porosity of a grid cell and approximating the boundary(ies). Consequently, the LSM and Lagrangian viewpoint for topology optimization gained popularity among the fluid dynamics community. There are clear advantages to using the LSM for topology optimization for fluid dynamics problems that are discussed in the following section.

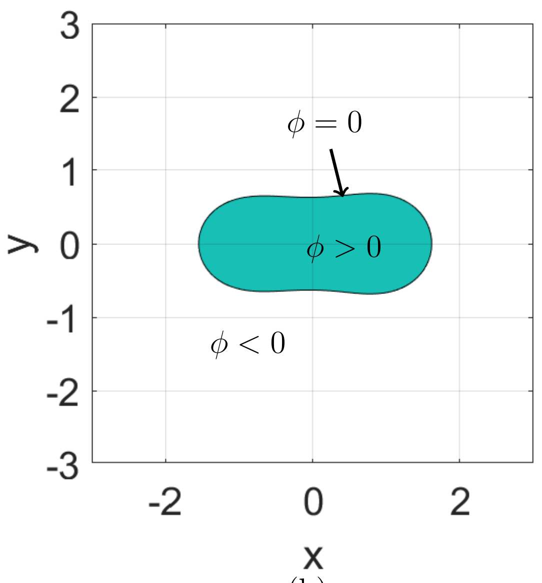
## 2.3 The level-set method

The level-set method (LSM) was first introduced by Osher and Sethian[51] to track flame front propagation. It was created to handle complex geometries that can change and deform into any shape. Since its introduction, the LSM has been developed and implemented to analyze and solve a large range of problems: geometry, grid generation, image enhancement and noise removal, computer vision, computational geometry, optimality and first arrivals, etching and deposition in microchip fabrication, physical phenomena analysis and simulation, as well as others. Some of its applications are briefly discussed below.

The idea of the LSM is rather simple: an isosurface or isocontour of a function is used to represent the design boundary(ies). Specifically, the isosurface or isocontour that corresponds to the LSF value of zero, $\phi = 0$. So, for a 3D problem, a 4D LSF would be required and the 3D design topology would be represented by $\phi = 0$. Similarly, for a 2D problem, a 3D LSF would be necessary and the 2D design topology would be represented by $\phi = 0$. Fig. 2.2 shows an example of the 2D case. The 3D LSF is shown in red and the zero plane is drawn in blue. This plane is referred to as the waterline because everything above it is "seen" and everything below it is "unseen" as if looking at an island from the top down. The resulting "island" is the level-set representation of the design, which shown in Fig. 2.2b. Using $\phi = 0$ as the demarcation between inside the boundary and outside the boundary is common in most LSM applications because it defines the clear distinction that the level-set value is positive within the boundary, zero along the boundary, and negative outside.



(a)



(b)

Figure 2.2: Example of a 3D LSF $\phi$ (a) and the resulting 2D design boundary represented by $\phi = 0$ (b)

An advantage of using the LSM is that the LSF can be defined to be derivative-continuous to any degree, based on what is required for a given problem. Furthermore, the function is known analytically, so gradients at the level-set design boundary(ies) are readily available making gradient-based optimization techniques useful tools for optimizing the LSF. With a simple understanding of the LSM and how LSFs can be used to represent boundaries and topologies and calculate respective gradients, some applications using the LSM can be discussed.

### 2.3.1 General applications

Studies in geometry and the motion of boundaries based on normals and curvature have proven useful for physical investigations because motion by curvature has similarities with a diffusion-like term and can be used to relax and reshape the boundaries[52]. From these studies, it was shown that convex curves moving under the motion associated with the curvature must shrink to a point[53–55]. Extension of this result was shown by Huisken[56]. However, Grayson demonstrated that non-convex shapes may not shrink to a sphere or point[57].

Level-set grid generation techniques[58] are very similar to hyperbolic schemes: solving a wave equation to create successive grid cells in a marching manner. The level-set grid generation methods are mainly used for body-fitting grids or near-body grids. The initial idea for the level-set grid generation is to use the body itself as the initial position of the front and move the front in the outward normal direction according to the local curvature. This technique mitigates the shock formation and colliding characteristics that hinder most hyperbolic generators.

Noise reduction techniques were developed and investigated initially by Alverez, Lions, and Morel[59] to improve on conserving the sharp edges that are smoothed by traditional filtering methods. Variations of the level-set noise reduction scheme were also investigated with minor adjustments made to the equations of motion used to solve the problem[60,61]. Use of these schemes without stopping would result in removing all the information in the domain because of the theorem presented by Grayson[55]. Consequently, a stopping criteria was necessary. However, Malladi and Sethian were able to develop a scheme that did not need a stopping criteria which is referred to as the Min/Max flow scheme[62]. As the name implies, the method takes advantage of a Min/Max function that correctly chooses the velocity of the level-set front to remove the noise. Examples of the technique can be found in several papers[62–64].

Applications in computer vision use the LSM and its advantages to extract shapes from raw data[65]. In particular, computer-aided tomography (CT) scans are an example of using secondary data to reconstruct a shape. The LSM has been implemented for medical image generation and extraction and is discussed in several papers[65–70].

### 2.3.2 Applications in topology optimization

The LSM is a very powerful tool because it can easily capture topological changes. Other boundary tracking methods require added computational complexity when boundaries coalesce or tear apart. These methods track points on the boundary and need additional algorithms to check whether points have crossed over one another and then another algorithm to determine if the boundary tears or merges. The LSM, however, allows these phenomena to occur seamlessly because the only thing being tracked is the implicit curve that is defined at $\phi = 0$. This curve can be approximated at each iteration so no additional checks related to the points on the boundary are necessary. The advantages of this property of the LSM have been exploited for the sole purpose of improving topology optimization techniques.

As topology optimization grew in popularity, the development and implementation of the LSM for topology optimization dramatically increased. From structural optimization to fluid dynamics, the LSM has been used in a variety of ways. The LSM was introduced as an alternative way to represent the density field of the design domain smoothly. Using the LSM for topology optimization was first suggested by Haber and Bendsøe[71]. Initially, a hybrid methodology was suggested that combined level-set ideas with at the time current evolutionary algorithms for structural topology optimization[72,73]. Eventually, continuous, gradient-based optimization schemes were investigated[74] as well as topological derivative approaches[75]. Finally, the idea of framing the level-set structural optimization problem as one based on shape-sensitivities was presented to the community and remains the most popular approach to finding optimal solutions[19–21,76].

The LSM has two main types of implementation: implicit or explicit. The implicit method generates the LSF by defining the material domain as positive, the void domain as negative, and the boundary between the two is $\phi = 0$. An initial boundary is necessary to generate the initial level-set function. The key difference between the implicit and explicit LSFs is that there is no underlying parameterization for the implicit LSF. In fact, this is one of the challenges in implementing this technique. The generally accepted approach to generating the initial LSF is to create a signed-distance function based on the initial input boundaries. Once an initial level-set function is generated, changes and deformations to the LSF are applied by solving the level-set equation that is of the form

$$\frac{\partial \phi}{\partial T} + F|\nabla \phi| = 0, \tag{2.1}$$

where the LSF is a function of space and time, $\phi = f(\vec{x}, T)$ and $F$ is a scalar velocity term which is multiplied by the $L_2$-norm of the gradient of the LSF. The velocity term can be calculated by determining how the local movement of the boundary affects the objective function. Implicit methods solve Eq. 2.1 to deform and update the LSF at each iteration. Numerical methods for differential equations are used to solve these equations and are combined with other techniques for solving for the velocities $F$ throughout the level-set field.

Initially, the main approach for updating the level-set curve(s) was to use numerical methods to solve level-set equation, Eq. 2.1, at each iteration. As computational methods improved and mathematical programming techniques become more efficient, researchers began implementing them over the differential equation approach[77–81]. Furthermore, implicit implementations of the LSM are less suitable for fluid dynamics problems because these types of problems already require a large number of computational resources to solve the governing differential equations. Adding another layer of differential equation solves could greatly increase the computational load required for fluid dynamics design problems. Moreover, the most common approach for design optimization in fluid dynamics, particularly aerodynamics, is mathematical programming and gradient-descent techniques. Since the implicit methods do not fit this model, they were avoided and the explicit LSM was used instead.

The explicit LSM uses a parameterized LSF, which can be deformed and updated by changing its parameters. In contrast, the implicit LSM deforms and updates the LSF based on the solution to Eq. 2.1. By parameterizing the LSF, that is, making it dependent on variables that are user-defined and can be manipulated directly, the function can be used in conjunction with optimization algorithms that employ gradient-descent methods.

The LSF can be parameterized in a variety of ways including multivariate polynomials and radial basis functions (RBFs). Recently, parameterization using B-splines was implemented for structural dynamics problems by Zhang *et. al*[82] as an alternative level-set approach. However, using RBFs is the generally preferred method of parameterization for the LSF[83]. RBFs are defined as any function that decays to zero as the independent variable(s) moves away from a predetermined center. An example of this type of function is the well-known Gaussian bell curve. A Gaussian bell curve is said to have global support because it never truly decays to zero. Alternatively, compactly supported RBFs decay to zero at specified distances. The LSF is most commonly defined as the sum of $N$ radial basis functions (RBFs), $\psi_i$, and is parameterized by the coefficients, $\alpha_i$, the RBFs are multiplied by, and their center locations, $\vec{x}_i$, the expression is shown in Eq. 2.2,

$$\phi(\alpha_i, \vec{x}_i, \vec{x}) = \sum_{i}^{N} \alpha_i \psi_i(\vec{x}_i, \vec{x}), \tag{2.2}$$

and a pictorial representation is displayed in Fig. 2.3. The set up of this figure is similar to Fig. 2.2 with an additional plot beneath the 3D LSF that shows a vertical cut-plane to display the underlying RBFs used to generate the LSF. In Fig. 2.3b the support diameter is shown and it indicates the region over which each RBF is non-zero. Fig. 2.3c shows the profile

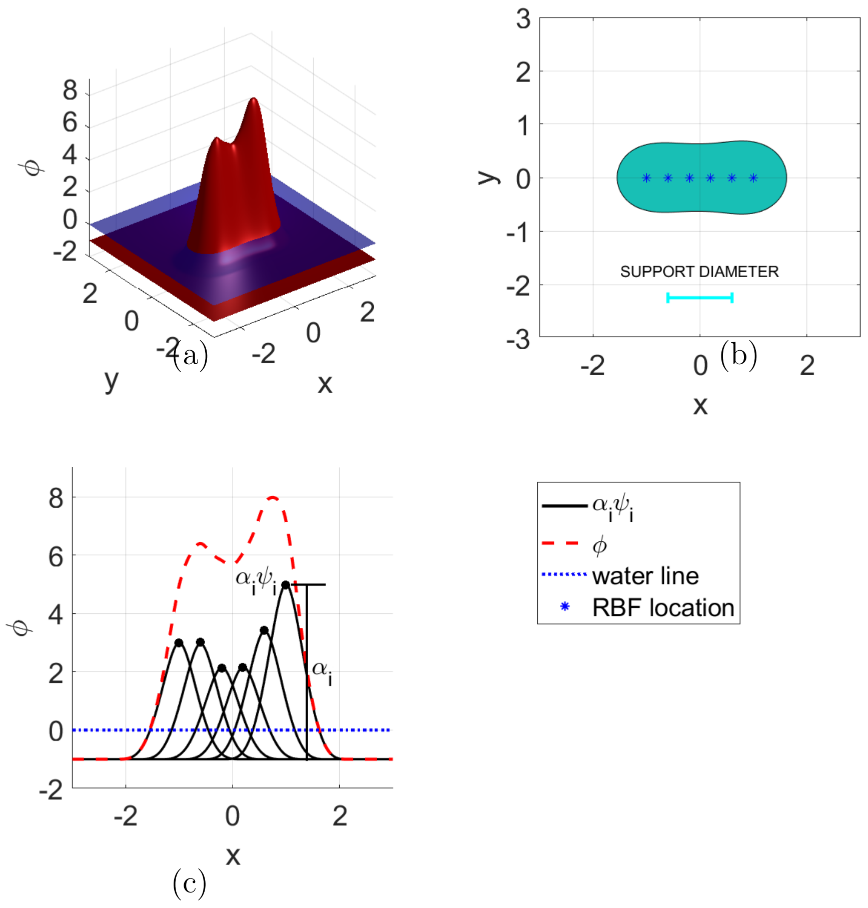


Figure 2.3: Example LSF (a), the level-set curve at the waterline (b), and a vertical slice showing the radial basis functions and their respective coefficients (c).

view of the LSF and associated RBFs multiplied by their respective coefficients. Numerical techniques are used to optimize the LSF (read: the RBF locations and coefficients) such that the zero-level-set curve of the LSF creates an optimal geometry for a given application. Further discussion of RBFs and level-set parameterization can be found in Ch. 3 with the discussion of the boundary representation technique used in this work.

Just as topology optimization developed in the structural dynamics community earlier than the fluid dynamics community, the implementation of the explicit LSM similarly was first used for structures problems. Wang and Wang[77] were the first to suggest an RBF-based level-set parameterization. They showed that by using a parameterized LSF the computational cost of the entire topology optimization scheme can be greatly reduced. Additionally, Wang *et al.*[84] extended the previous work to show that RBF parameterization allowed for hole nucleation without the need for additional algorithms as is the case with the conventional implicit schemes. Jiang *et al.*[85] developed a level-set scheme that uses cardinal basis functions as the parameterization, which are derived using RBFs. Wei *et al.*[86] published a MATLAB code illustrating the RBF LSM for educational purposes and also included a reference for all other educational topology optimization codes that have been produced to date. For a more indepth review of LSMs in structural topology optimization, please see the review written by Dijk *et al.*[87].

The topology optimization with a parametric level-set approach in the fluids dynamics community was initially studied by Pingen *et al.*[88]. The approach models the Navier-Stokes equations using the Lattice Boltzmann Method (LBM) that was shown to be effective for topology optimization problems in fluids[89,90]. These methods were further investigated by Kreissl *et al.*[91] in addition to exploring micro-fluidic applications[92], unsteady flows[93], and an extended finite-element method[94]. While the study of topology optimization for fluid dynamics has matured, there still exists unexplored techniques and flow regimes. Specifically, fluid flows with a Reynolds number larger than 1000 have been investigated sparingly. Furthermore, those investigations are mainly interested in the design of the entire flow domain and not the components within a prescribed boundary[95–98]; the design of components like turning vanes, turbine blades, or flow guides have not been examined within the topology optimization framework. Also, the RBF level-set approaches usually define a fixed grid of RBFs that are used to generate the LSF. While this approach works for channel design because the entire flow domain is modeled by the LSF, the work presented here uses a methodology similar to the work published by Zhang *et. al*[82]. The RBFs are defined along the boundary of the bodies as opposed to being defined in a grid layout in an effort to reduce the computational resources necessary for the optimization by reducing the number of RBFs in the domain, which in turn reduces the total number of design parameters. This is a key difference in this work compared to previous and current investigations in topology optimization for fluid dynamics problems. Moreover, there are several limitations in current topology optimization techniques that will be discussed and addressed in the following sections.

## 2.4 Limitations in current applications

There are several difficulties associated with applying topology optimization to flows with moderate to high Reynolds numbers. First, the solutions of a topology optimization problem depend on the initial topology. The number of holes/bodies and their orientation affect the optimal solution. For example, Fig. 2.4 shows the optimization of a cantilever structure from the work by Dunning and Kim[99]. The optimum design is different for each case, which implies that topology optimization problems have robustness issues. Initially, there was no method for introducing holes into a design, so initial topologies (number of holes in the initial design) were expected to affect the solution, as seen in the middle column of the Fig. 2.4. Dunning and Kim[99] developed a hole insertion method based on stress concentrations; holes are placed in areas of low stress concentration as a guess. The hole placement is evaluated and either accepted or rejected. Yet, the initial topology still had an effect on the solution even after a hole insertion method was implemented which is displayed in the right column of the figure. These results suggest that finding an appropriate initial guess can have a profound effect on the optimization results. Moreover, this is not only seen in structural dynamics problems.

An example in topology optimization for fluid dynamics problem where the initial guess effects the result is shown in Fig. 2.5. The channel design is optimized to minimize the energy loss across the domain. The end result is a channel that meanders around the obstacle. Figs. 2.5b and 2.5c show two different solutions to the same problem. The difference between the two cases was the initial conditions, similar to structural topology optimization problems. Sá *et al.* acknowledge this discrepancy in results as two local minima of the problem[100].

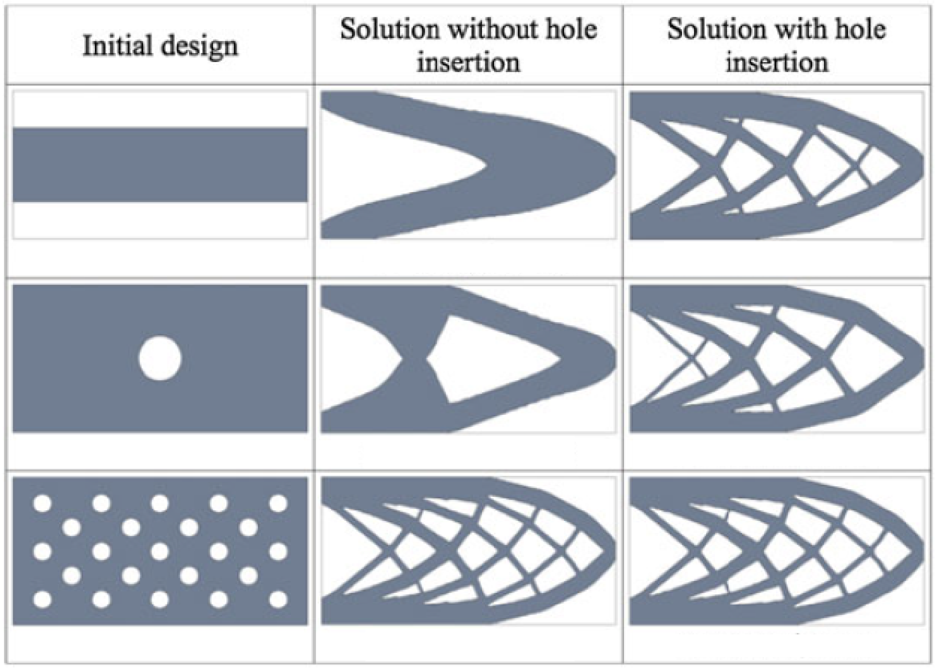
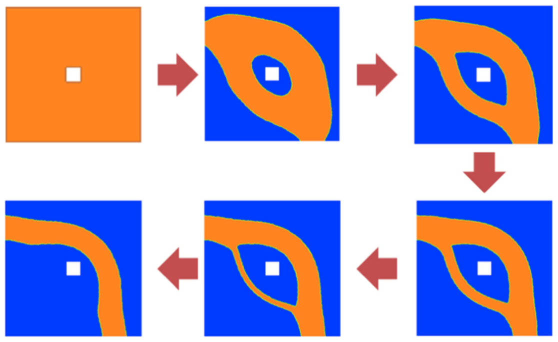
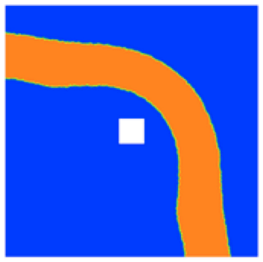


Figure 2.4: The effects of different initial conditions on the topology optimization solution, modified from Dunning and Kim[99]

Multiple flow solutions will be necessary to find a true optimum solution without a predetermined initial topology. The computational cost of solving this type of design problem can be prohibitive, even if a Reynolds-averaged Navier-Stokes (RANS) flow solver is used. The research presented here explores obtaining initial guesses from given objective function information in an effort to improve the overall efficiency and accuracy of the optimization scheme. The results are discussed later in the document.

Another difficulty is boundary representation, which is very important for moderate to high Reynolds number flows. Most of the methods used for topology optimization output stair-stepped geometries and often result in bumpy, faceted design surfaces, even after smoothing. In Fig. 2.6 taken from a presentation given by Kim[101], we see two separate topology optimization solutions for the same problem for different initial conditions. Again, the solutions seem to be dependent on the initial conditions, but the main point is that the final boundary representation is jagged, lumpy, and faceted in various places. Moreover, close inspection of Fig. 2.5 shows that the boundaries are not smooth. The Reynolds num-


(a) Optimization iterations with all fluid initialization


(b) Domain with all fluid initialization

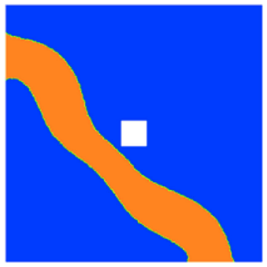

(c) Domain with all solid initialization

Figure 2.5: Topology optimization of a channel with an obstacle[100]

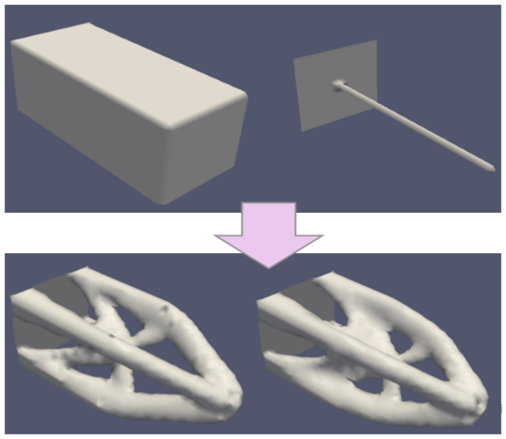


Figure 2.6: Example of a bumpy final designs[101]

ber is 2 in this example, so the boundary representation is not as important as for higher Reynolds number applications. At higher Reynolds numbers, discontinuities and bumps in a surface can cause issues such as separation and inaccurate flow results. In the examples and applications of topology optimization for fluid flows presented in the previous section the level-set boundary was approximated by some form of linear interpolation, either directly using the level-set values[91] or density/porosity[95,96] values. Even if the boundary points were approximated using higher-order methods, there has been little work in connecting the resulting points smoothly to ensure a degree of continuity greater than point-to-point ($C^0$). The topology optimization algorithm presented here aims to improve the results of an optimization run by finding a suitable initial guess and producing the curvature continuous boundaries that are desired for current high-fidelity fluid flow solvers.

Herein, a mathematical programming approach is taken to optimize the RBF locations and coefficients to optimize the LSF. This approach is tailored to be adapted for fluid dynamics applications so care was taken to generate points along the zero level-set in a way that allows end-users to cluster points and spread points in areas how high and low curvature, respectively. The approach for extracting curvature continuous boundaries is presented in the following chapter.

# Chapter 3

# Curvature continuous boundary representation

Computational fluid dynamics (CFD) algorithms have improved steadily with the increased processing power of computers. As algorithms improve, the complexity of the geometries and topologies increase. Grid resolution is strongly correlated to algorithm efficiency and accuracy. The more complex the geometry or topology, the more grid cells are needed to accurately represent it. However, the number of grid cells can be reduced by increasing the order of the grid cell representation. The order of grid cell representation is related to the degree of continuity that they represent. Similar to the continuity of a curve, $C^0$ represents a curve that is continuous from point to point along it, $C^1$ is continuous to the first derivative (slope) at each point, $C^2$ is continuous to the second derivative (curvature) at each point, and so on. Higher-order grids can be used for higher-order CFD. To clarify, a method is said to be of $kth$ order if the solution error and the mesh size to the power $k$ are proportional[102] as illustrated by Eq. 3.1,

$$e \propto h^k. \tag{3.1}$$

The reader is referred to the review article written by Wang *et al.*[102] for its thorough description of the advantages of higher-order schemes while dispelling the myths about the disadvantages of these methods.

In the past decade, researchers have begun investing resources in developing methods that make higher-order methods ($3rd$-order and higher) more attractive because of their better accuracy and efficiency when compared to the lower-order counter-parts. Specifically, research and investigations of reliable and robust higher-order grid generation have been conducted[103–105]. In view of this trend, the work presented here aims to supplement and improve the methods for generating curvature continuous boundaries for higher-order mesh generation for aerodynamic simulation and design optimization.

The level-set method (LSM) provides a medium for obtaining a design that has continuous second derivative. The user controls how the level-set function (LSF) $\phi$ is parameterized and can therefore control the boundary representation. As mentioned in the previous chapter, however, the LSM is an implicit means of representing curves, which means there is no explicit function that defines the curve, mathematically shown in Eq. 3.2,

$$\phi(\vec{x}) = 0. \tag{3.2}$$

The level-set curve can be fit using splines, in particular, cubic splines that are guaranteed to be curvature continuous. The methodology for producing these splines is briefly discussed in the next sections and expanding on in the following chapter.

## 3.1 Radial basis function parameterization

The LSF that used in this work is generated by summing a predetermined number of radial basis functions (RBFs). Each RBF has an associated coefficient, $\alpha_i$, which can be prescribed or solved for, depending on the application. The footprint of an RBF, or how large its base is, can be controlled by changing the radius at which each function decays to zero. This distance from the central point is called the *support radius.* RBFs that have a defined support radius are referred to as compactly-supported RBFs. If, however, an RBF does not have a defined support radius, it is said to have global support. Originally globally-supported RBFs were solely used for level-set topology optimization problems. It was within the last 15 years that compactly-supported RBFs (CSRBF) began being investigated as possible alternatives because using them over globally-supported RBFs (GSRBF) may improve computational efficiency of an optimization scheme[106,107]. This improvement would be the result of reducing the amount of computations necessary for each RBF since the influence of each RBF would be limited. Wendland introduced a variety of RBFs that have compact support and varying continuity[107]. Of particular interest, the Wendland C2 RBF has continuity to the second derivative and as such makes it a good candidate for use in a LSM applied to fluid dynamics problems where preserving the boundary continuity is paramount. While a comprehensive study of various RBFs is beyond the scope of this study, it would be useful for future research to explore different RBFs and how they affect the results of the optimization.

The Wendland C2 RBF is represented by Eq. (3.3) and is shown in the Fig. 3.1.

$$\psi(r) = \begin{cases} \left(1 - \frac{r}{SR}\right)^4 \left(4\frac{r}{SR} + 1\right) & , \quad r \leq SR \\ 0 & , \quad r > SR \end{cases} \tag{3.3}$$

In Eq. (3.3), $\psi$ is the RBF, $r$ represents the radius from the center of the function, and $SR$ is the support radius. For this study, $r$ is chosen to be the radius of a circle, but it can be defined to represent other shapes such as the radius of an ellipse or a super-ellipse. The name of the function (Wendland C2) denotes that it has $C^2$ continuity, which means the RBF is continuous to the second derivative (its curvature). This function was chosen to maintain C$^2$ continuity throughout the design domain and produce curvature continuous level-set curves. Furthermore, the LSF has an analytic expression, may allow for the control of the boundary slopes and curvatures during an optimization process. The control of these quantities can help produce the smooth boundaries that are desired for optimal designs that interact with high Reynolds number flows.

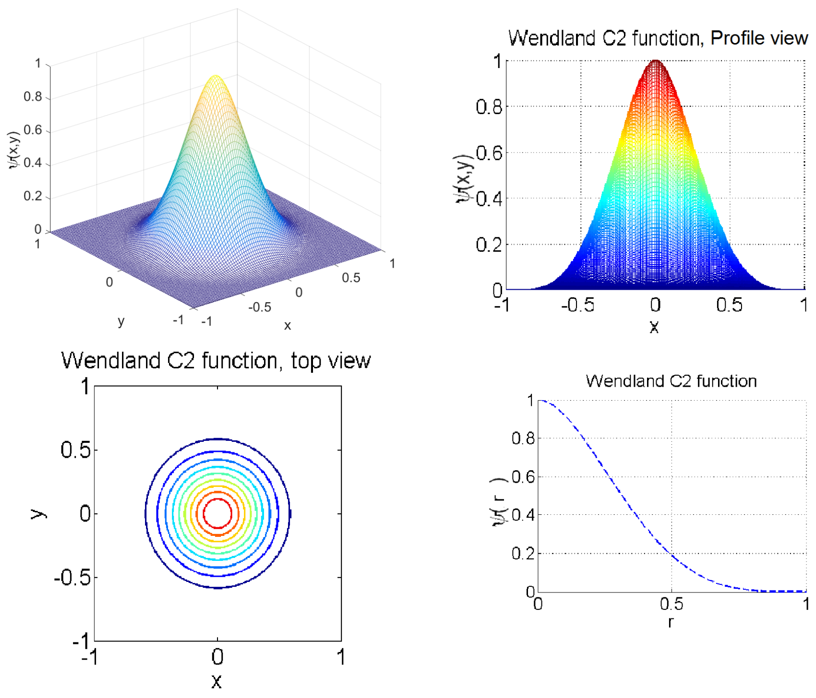


Figure 3.1: Using the Wendland C2 function profile and a circular footprint as an RBF

## 3.2 Determining level-set parameter values

The parameterization of the LSF can have an effect on the boundary representation of the level-set curve. The level-set parameters are the locations and coefficients of the RBFs. Since the focus of this work is the topology optimization of bodies within a large domain, the RBFs are placed along the boundary of the design topology. In contrast, the more widely used implementation places the RBFs in a fixed Cartesian grid. The justification for the difference in setup is that using a grid of RBFs can better represent topological changes throughout the entire domain and is therefore more useful for the design of channels. This has been the main focus of topology optimization applications in the fluid dynamics community. However, this work focuses on the components within the fluid flow domain, so more focused attention to the boundaries of these components is beneficial. Furthermore, reduced computational resources are required because fewer RBFs are required to represent individual bodies. Also, using a fixed grid of RBFs can result in some RBFs being inactive throughout the optimization scheme if the RBFs are far enough away from the topology boundaries. Thus, distributing the RBFs along the boundaries can keep them close to the boundaries throughout the optimization scheme while reducing the total number of design parameters, significantly lowering the computational cost of the entire process.

The locations of the RBFs can be determined once the design topology is defined and the number of RBFs along each boundary is set by the user. Next the RBF coefficients need to be solved for. The support radius of the RBFs needs to be defined to solve for the RBF coefficients. Determining the support radius value is a non-trivial task and can result in undesirable results including premature topology change during the optimization process. Furthermore, it may be useful to define a relationship between the number of RBFs distributed along the boundary and the size of the support radius. In the following examples, the effect of varying the number of RBFs, the support radius, or defining the support radius based on the number of RBFs is explored to help inform what should be implemented in the algorithm. Fig. 3.2 shows the results of three cases. The RBFs are defined inside and outside a circle of radius $r = 0.5$. It is important to note that the RBF offset locations (inside and outside the circle) are based on the support radius.

Each figure is setup to show the LSF, associated zero level-set contour, and the local radius of the level-set curve, $r_{i,LSF}$, versus $\theta$ along the boundary. The local radius is compared to the average radius, $r_{avg}$, of the curve to examine the variation in the boundary, qualititatively. The *root-mean-square* of the local radius and the average radius is calculated using Eq. 3.4,

$$RMS_{r,avg} = \sum_{i}^{N} \left( r_{i,LSF} - r_{avg} \right), \tag{3.4}$$

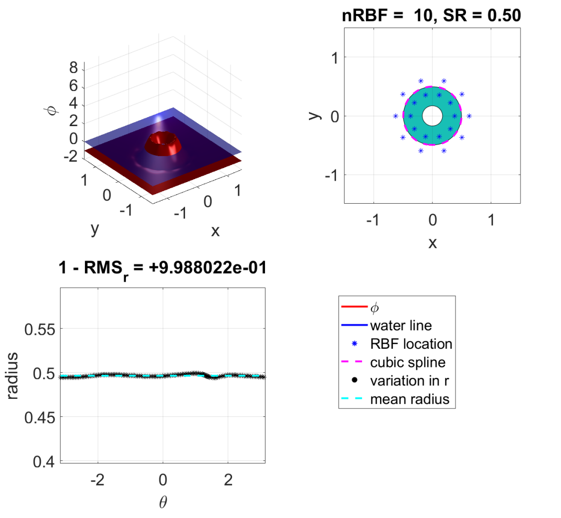
and is used to determine how well the boundary approximates a circle with average radius calculated by fitting a spline about the zero level-set curve. Examining each example, it can be seen that defining the support radius based on the number of RBFs is the most beneficial approach and has been used in the optimization algorithm accordingly.

Each figure shows that a reasonable approximation to a circle can be made, however, the radius of the circle is not important; it is the smoothness of the boundary that is of interest. The r vs. $\theta$ plots show the variation of the boundary and the RMS value provides a quantitative analysis of how well the curve matches a circle. Of the three cases, the case relating the support radius to the number of RBFs outperforms the others. The relationship between the support radius and the RBFs is defined in Eq 3.5,

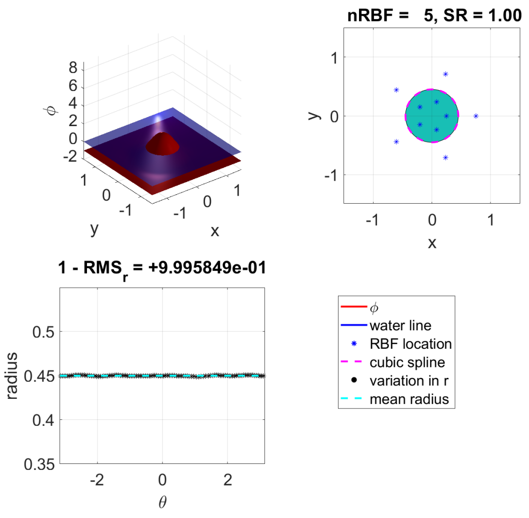
$$SR = max(d_{i,RBF}) \times N_{SR}, \tag{3.5}$$

where $max(d_{i,RBF})$ is the maximum distance between neighboring RBFs and $N_{SR}$ is a user-specified multiplier. Essentially, the support radius becomes related to the arc length of the boundary and can be used to ensure that a large enough value is calculated such that the boundary is represented relatively smoothly. For static level-set representations the support radius and number of RBFs can be changed at will to obtain the best result possible for a particular curve. However, this type of tweaking is unavailable to the user during a design optimization run. Thus, using a relationship that links the number of RBFs along the curve to the support radius allows the support radius to change throughout the optimization. By implementing this relationship, if a poor initial support radius was used the effect on the optimization process is reduced when compared to holding the support radius fixed throughout the optimization. The distribution and calculation of the level-set parameters are expanded upon in the following chapter using an example.

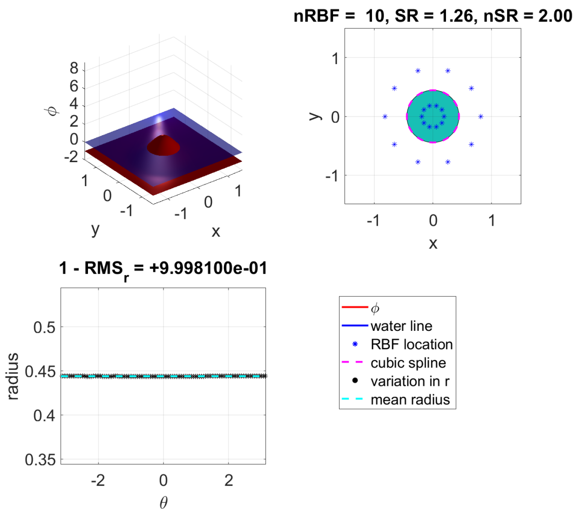
The ideas presented here can be used to generate a LSF for any design topology. Since the LSF is known analytically the derivatives can be directly calculated anywhere. Specifically, the derivatives at a point along the zero level-set curve can be used to generate points along the entire boundary using an adaptive Runge-Kutta scheme. A cubic spline can be fit through these points using a general cubic spline formula and a modified Thomas algorithm. Thus, a curvature continuous representation of the design topology can be extracted. These methods as well as the overall optimization scheme are thoroughly explained using an example in the next chapter.



(a) Smaller support radius for more RBFs



(b) Larger support radius for fewer RBFs



(c) Support radius defined by the distance between neighboring RBFs

Figure 3.2: Different results for several the number of RBFs and support radius

# Chapter 4

# Optimization methodology

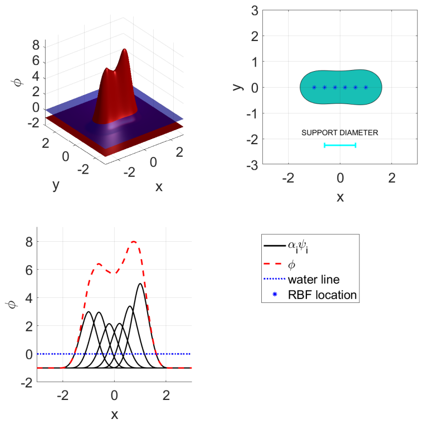
Shapes and topologies can be represented and defined without conventional parameters using the LSM. For example, a circle can be represented without having to define circle-specific parameters such as center point or radius. This is possible by first creating an LSF, $\phi$, and prescribing a constant-valued contour as the shape or topology that is desired. A user can choose which contour represents the desired boundary and use that shape for a design. In this work, the zero contour was used as the constant value of $\phi$ because it is the most common value used in literature. Furthermore, the zero level-set contour provides a means of identifying the desired boundary by ensuring everything outside the level-set curve is negative and everything inside is positive.

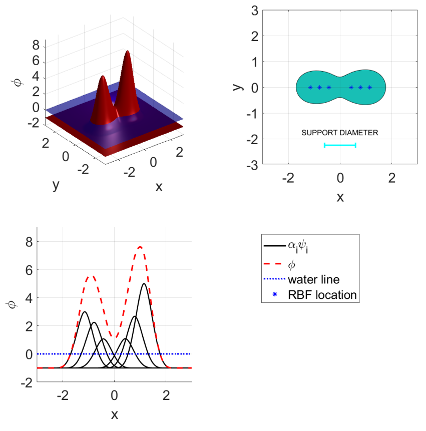
The boundary(ies) that are defined by an isocontour of the LSF (a curve of constant level-set value) can be deformed by changing the coefficients, $\alpha_i$, and/or the locations of the RBF centers, $\vec{x}_i$. Additional advantages of using RBFs as design parameters are seen in design optimization where the initial design could consist of a completely different shape or topology from the final design. For example, an airfoil cannot be generated by optimizing the conventional design parameters of a circle. However, using the LSM would make it possible that both could be generated using the same set of parameters (of different values). An example of generalized parameterization was illustrated for this particular problem (circle to an airfoil) in a recent study by He *et al.*[108].

Most implementations of the LSM for topology optimization use a parameterization that is based on the coefficients or scaling of the RBFs. Translating the RBFs was initially proposed by Xing *et al.*[109] and was used in subsequent studies by Ho *et al*[110,111]. However, it has been found that interchanging the positions of two RBFs can result in no change in the objective which leads to an ill-posed optimization problem[87]. A recent study, however, investigated changing the RBF coefficient and location simultaneously and its effect on structural topology optimization problem solutions[112]. The results demonstrated that there is promise in varying both parameters. Furthermore, with the focus on the boundary(ies) of the bodies within the fixed and the local distribution of RBFs versus the conventional fixed grid distribution, it seemed appropriate that the locations and coefficients are both used as design parameters in this study. However, the researchers implemented the differential level-set approach using the Hamilton-Jacobi equation to update the LSF. Conversely, the work presented here uses a mathematical programming approach to update the level-set parameters and optimize the design topology.

Another advantage of the LSM, in regard to design optimization, is that it enables shape and topology optimization to occur simultaneously. Since the LSM defines a geometry by a contour at a constant value of the LSF, a contour of constant value could outline multiple "islands." In Fig, 4.1 for example, the LSF is an M-shaped function generated using Eq.(2.2). At a constant curve value of $\phi(x, y) = 0$ the LSF generates two shapes. By moving the RBFs and changing their coefficients the boundary pinches in the middle (Fig. 4.1b) and then tears into two separate bodies (Fig. 4.1b). This figure demonstrates the main idea for this work: manipulating the RBF locations and coefficients can produce topological changes to generate an optimal design.

The optimization scheme used in this study is outlined in the Fig. 4.2 and expanded upon in the following sections. A combination of zero-finding, numerical function integration, one-dimensional gradient search, and gradient-descent algorithms are employed for the


(a) Initial LSF


(b) Pinching in the middle due to RBFs being pulled apart and changing coefficients

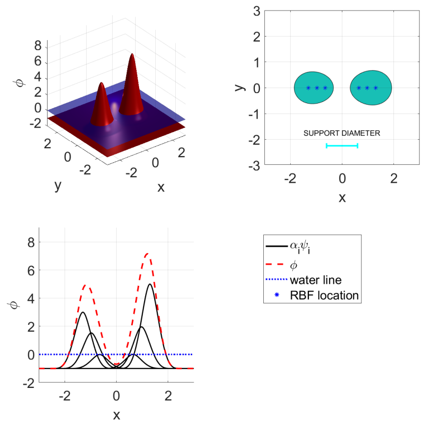

(c) Final torn boundary creating two new bodies

Figure 4.1: Representing multiple geometries (and topological changes) with an LSF

optimization of the LSF as well as various checks to ensure the correct topology is being represented. Fig. 4.2 illustrates the overall optimization scheme in broad terms. A desired outcome is input to generate an initial guess for the optimization scheme in the form of LSF parameters. These parameters are input into the optimizer where the zero level-set curve is extracted and used to evaluate the objective function as well as calculate the design sensitivities (how the boundary[ies] move with respect to the level-set parameters). The design sensitivities are then used within the optimizer to determine the search direction and the step size at each iteration. The procedure produces an optimal output once certain criteria

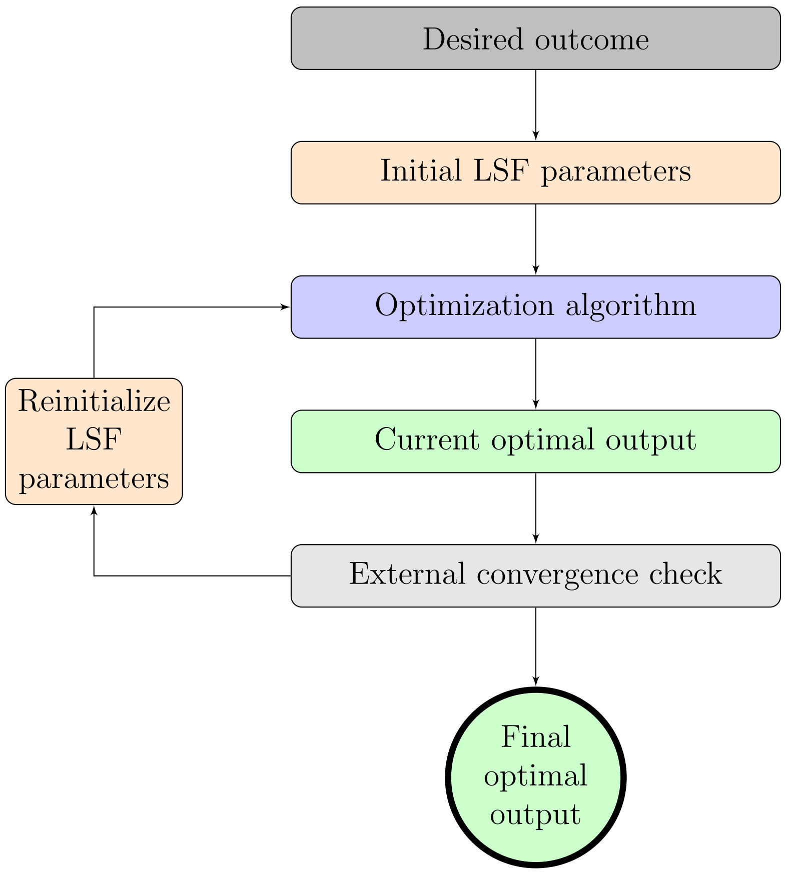


Figure 4.2: Overall optimization outline

are exceeded during an internal check. The optimal output is then checked again outside of the optimization algorithm and it is determined whether reinitialization is necessary or if this design is the final optimum value. Each of these components are discussed in the following sections. Before discussing each part of the optimization, however, it is important to provide an overview of general optimization techniques because both gradient-based and gradient-free techniques are implemented.

For this work, the desired outcome is the x-rays of a given topology. The generation of these x-rays is discussed in Sec. 4.3 along with how they are used to generate the initial level-set parameters. Three methods are used for generating the level-set parameters and are compared in Ch. 5. One of these methods involves a form of heuristic (gradient-free) optimization. Once the parameters are calculated they are input to the optimization algorithm. A gradient-based optimization scheme is used to update and change the level-set parameters. In particular, a conjugate gradient algorithm was used. Both of these types of optimization methods will now be presented and expanded upon.

## 4.1 Gradient-based optimization

The analytical approach to design optimization can produce efficient and robust results. These methods are some of the most popular in the design optimization field because of this. The idea is to use the gradient of the objective function with respect to the design variables to calculate a direction to change the variables in. Gradient descent schemes take the direction and reframe the problem as a one dimensional search problem to determine the optimal step along the gradient direction that minimizes the objective. There are two main gradient descent schemes: steepest descent and conjugate gradient descent.

Steepest descent is the most basic form of gradient descent optimization. The procedure is the same as detailed above and is expressed mathematically in Eq. 4.1,

$$\vec{p}^{\,k+1} = \vec{p}^{\,k} + \mu^k \vec{d}^{\,k}, \tag{4.1}$$

where $\vec{p}$ is an array of the design variables, $\vec{d}$ is the search direction, $\mu$ is the step size along the gradient, and $k$ is the current iteration. For classic gradient descent, $\vec{d}$ is calculated as the negative of the gradient of the objective function. So, if we define the objective function as,

$$objective\ function \equiv O, \tag{4.2}$$

and set the gradient with respect to the design variables as,

$$\frac{\partial O}{\partial \vec{p}}^{\,k} = \vec{c}^{\,k}, \tag{4.3}$$

then we can define the gradient descent direction as,

$$\vec{d}^{\,k} = -\vec{c}^{\,k}. \tag{4.4}$$

As the name implies, gradient descent techniques change the design variables along the direction of decrease in the objective function. Thus, the negative of the gradient is used as the descent direction (as shown in Eq 4.4) because the gradient is always in the direction of increase, by its definition. Once the optimal step is found, the design parameters are updated and the gradient is calculated at the new point and the procedure starts over. The implementation of a steepest descent algorithm is very straight forward, but it can be susceptible to high computational cost if the objective function is complex. The conjugate gradient method was developed to help mitigate this cost.

Conjugate gradient is a variant of steepest descent that uses some portion of the previous gradient to improve the current search direction. The equation for $\vec{d^k}$ is modified as,

$$\vec{d}^{\,k} = -\vec{c}^{\,k} + \beta^k \vec{d}^{\,k-1} \tag{4.5}$$

where the second term is the old search direction multiplied by a constant $\beta$ parameter to determine how much of the old direction is used. The $\beta$ parameter can be calculated in several ways as shown by Fletcher-Reeves[113], Hestenes-Stiefel[114], and Polak-Ribiére[115] and displayed in Eqs. 4.6

$$\beta^k = \begin{cases} \frac{\left(\vec{c}^{\,k} \cdot \vec{c}^{\,k}\right)}{\left(\vec{c}^{\,k-1} \cdot \vec{c}^{\,k-1}\right)} & \text{: Fletcher-Reeves} \\ \frac{\left(\vec{c}^{\,k} \cdot \vec{y}^k\right)}{\left(\vec{d}^{\,k-1} \cdot \vec{u}^{\,k}\right)} & \text{: Hestenes-Stiefel} \\ \frac{\left(\vec{c}^{\,k} \cdot \vec{y}^{\,k}\right)}{\left(\vec{c}^{\,k-1} \cdot \vec{c}^{\,k-1}\right)} & \text{: Polak-Ribiére} \end{cases} \tag{4.6}$$

where $\vec{y}$ is defined as the difference in the gradients of the objective function at two successive iterations as

$$\vec{y}^{\,k} = \vec{c}^{\,k} - \vec{c}^{\,k-1}. \tag{4.7}$$

For a general function these $\beta$ values can be quite different and based on numerical experiments the following procedure for calculating $\beta$ is recommended in Arora's text on design optimization[116]:

$$\beta^k = \begin{cases} \beta^k_{pr}, & \text{if } 0 \leq \beta^k_{pr} \leq \beta^k_{fr} \\ \beta^k_{fr}, & \text{if } \beta^k_{pr} > \beta^k_{fr} \\ 0, & \text{if } \beta^k_{pr} < 0 \end{cases} \tag{4.8}$$

where $\beta^k_{pr}$ is the value obtained using the Polak-Ribiére formula and $\beta^k_{fr}$ is the value obtained using the Fletcher-Reeves formula. Note, the Hestenes-Stiefel formula has not been used here because the other two formulas have shown better numerical performance[116].

### 4.1.1 One-dimension search for an optimal step

An optimization scheme uses the gradients as a search direction. Once the change in the objective function with respect to the design variables is determined, the optimizer takes steps in that direction until a minimum is obtained. The objective function can be transformed from a function of $N$ variables to a function of one variable as shown in Eq. 4.9,

$$O(\vec{p}) = O(\vec{p} + \gamma\vec{d}), \tag{4.9}$$

where $O$ is the objective function parameterized by the design variables $\vec{p}$, $\vec{d}$ is the gradient acting as the search direction for this iteration, and $\gamma$ is the free variable. Therefore, the objective function is simply a function of $\gamma$. As $\gamma$ changes the design variables are changed and a new objective is calculated. The variable $\gamma$ acts as the distance along the gradient direction and a one-dimension search algorithm varies it to find the optimal step to use to update the design variables and start the next iteration. The process of searching for a minimum along the gradient direction is called a *one-dimensional search* and there are a number of ways to perform this portion of the optimization scheme. Of note are the equal-section search, golden-section search, and quadratic approximation. In this work golden-section search was used to determine the optimal step along the gradient direction, but equal-section search would have sufficed with added computational cost.

Golden-section search uses the golden ratio 1.618 to step along the search direction until a minimum is bracketed. The variable used to step along the search direction is defined as $\gamma$ in this work, but it can be seen elsewhere defined as $\alpha$; $\gamma$ was chosen to avoid confusion with the RBF coefficient variable $\vec{\alpha}$. For golden-section search, $\gamma$ is incremented as shown in Eq. 4.10,

$$\gamma^j = \gamma^{j-1} + \delta(1.618)^{j-1}. \tag{4.10}$$

Here, $\delta$ is a scaling parameter that is chosen such that the steps are not too large where the first step is larger than the current objective function value, and not too small so the algorithm spends many computational resources stepping downhill. Choice of an appropriate $\delta$ is problem dependent and can be determined by understanding the problem at hand and with knowledge of how large the gradients will be and how much change in the objective is expected and acceptable. An initially small $\delta$ of $1e^{-3}$ was chosen to begin the optimization, but as the scheme progresses it is updated to one tenth the previous $\gamma$ value. In this way, the one-dimensional search algorithm takes the knowledge of the previous step and informs the subsequent one and improves the overall speed of the search algorithm. If, however, the $\delta$ value is too large and the step finds an objective function that is larger than the initial objective value, $\delta$ is reduce by a factor 0.1 until an objective value is found that is less than the initial. If $\delta$ drops below $1e^{-16}$ then the search algorithm exits and the optimization ends because no step can be taken in the direction of the current gradient.

The golden-section search algorithm increments $\gamma$ until either a step size criteria is met or a minimum is bracketed. The latter occurs when the objective function at the $j-2$ and $j-1$ iterations are greater than the $j^{th}$ iteration. Since the objective function is continuous, there must be a minimum between $\gamma^{j-2}$ and $\gamma^j$. These two values are set as the lower and upper bounds, $\gamma_L$ and $\gamma_U$, respectively. An interval of uncertainty is defined as,

$$I = \gamma_U - \gamma_L, \tag{4.11}$$

and is used as the exit criteria. If $I$ is less than a very small value, $\gamma = \frac{\gamma_U + \gamma_L}{2}$. An $I$ tolerance of $1e^{-8}$ was chosen to ensure that the difference between objective function determined by the either $\gamma_U$ or $\gamma_L$ was sufficiently small. The golden-section search algorithm iterates to the minimum by dividing the function between $\gamma_L$ and $\gamma_U$ into four points, $\gamma_L$, $\gamma_U$, $\gamma_A$, and $\gamma_B$. Initially, $\gamma_A$ and $\gamma_B$ are set to using $\gamma_L$ and $I$ as,

$$\begin{aligned} \gamma_A &= \gamma_L + \left(1 - \frac{1}{1.618}\right) I \\ \gamma_B &= \gamma_L + \left(\frac{1}{1.618}\right) I. \end{aligned} \tag{4.12}$$

At each iteration, the objective function is evaluated using both $\gamma_A$ and $\gamma_B$ and the two resulting values, $O_A$ and $O_B$, are compared. The variables $\gamma_A$ and $\gamma_B$ are updated according to the following logic statements:
if $O_A < O_B$:

$$\begin{aligned} \gamma_L &= \gamma_L \\ \gamma_U &= \gamma_B \\ \gamma_B &= \gamma_A \\ \gamma_A &= \gamma_L + \left(1 - \frac{1}{1.618}\right)(\gamma_U - \gamma_L) \end{aligned}$$

else if $O_A > O_B$:

$$\gamma_U = \gamma_U$$
$$\gamma_L = \gamma_A$$
$$\gamma_A = \gamma_B$$
$$\gamma_B = \gamma_L + \left(\frac{1}{1.618}\right)(\gamma_U - \gamma_L)$$

else if $O_A == O_B$:

$$\gamma_L = \gamma_A$$
$$\gamma_U = \gamma_B$$
$$\gamma_B = \gamma_L + \left(\frac{1}{1.618}\right)(\gamma_U - \gamma_L)$$
$$\gamma_A = \gamma_L + \left(1 - \frac{1}{1.618}\right)(\gamma_U - \gamma_L).$$

Implementing this search algorithm yields a $\gamma$ value that can be used to update the design variables and advance to the next iteration in the optimization or satisfy one of the various convergence criteria.

## 4.2 Gradient-free optimization

Gradient-based optimization has its benefits when the derivatives of an objective function are easily calculated or approximated. However, when the gradients are unavailable or the objective function is discontinuous another type of optimization technique can be used: gradient-free optimization. These techniques use heuristics or loosely defined rules to iterate from an initial objective value to an optimum. The Nelder-Mead algorithm[117], genetic algorithm[118], simulated annealing[119], and particle swarm optimization[120] are examples of this type of optimization and a brief description of the methods is presented here.

Nelder-Mead uses simplexes and operations such as flip, shrink, and expand to iteratively modify the simplex to move towards the optimum value. The operations are used based on various logic statements written in the algorithm and no gradient information is necessary.

By iteratively changing its shape according to the values at each of the points on the simplex, the optimization scheme is able to always locate at least a local minima.

The genetic algorithm is a subset of the later created category of evolutionary optimization algorithms. The algorithm uses operations similar to the evolution of a species, since the idea is based on Darwin's theory. The design solutions are referred to as chromosomes and the design parameters are the genes that make up each chromosome. A large number of chromosomes are generated initially as the initial population. The population is updated each iteration by "mating" certain chromosomes with other chromosomes and producing children. The "mating" procedure consists of a crossover operation and a mutation operation, also referred to as genetic operations. These operations are based on random number generation to switch genes around, generate new chromosomes, and iterate to an optimum by keeping the best in the population.

Simulated annealing uses ideas from annealing in metallurgy. The method starts with an initial temperature that dictates how bad the accepted step can be. At each iteration, the design parameters are perturbed in a random manner and the change in objective function is calculated. If the change is acceptable according to the current temperature the step is taken; if not, the perturbation is thrown out and the iteration is repeated. The temperature is reduced according to an annealing schedule, which may be user-defined. The temperature can start at any value and then end at zero.

Particle swarm optimization using the swarm mentality to drive an optimizer to find the minimum of a function. A prescribed number of initial guesses are generated. The solution is checked at all locations of the swarm members and the best value in the swarm is marked and each member moves toward the better value. The motion of the swarm is based on certain user defined parameters. By marking the best value at the current iteration and moving the swarm accordingly, this optimization scheme effectively sweeps the design domain and attempts to find the global optimum.

Gradient-based and gradient-free optimization methodologies have their respective strengths and weaknesses. Gradient-based optimization is mainly used for local design optimization while gradient-free methods attempt to explore the entire design space in a brute-force way by testing numerous designs with different design parameter values. With regards to topology optimization, however, the gradient-based approaches dominate the field. Thus, a conjugate gradient scheme was implemented to produce the optimization results shown in the next chapter.

## 4.3 Initializing the level-set parameters

Most optimization schemes (if not all) are susceptible to poor initial guesses that can lead to either a divergent solution or finding a local minimum that is much larger than other local minima. Finding a suitable initial guess can be just as important to the success of the optimization scheme as defining a sufficient objective function and appropriate design variables. The design variables for the LSF are the locations, $\vec{x}_{RBF}$, and coefficients, $\vec{\alpha}$, of RBFs. This parameterization gives the LSF three degrees of freedom per RBF and the support radius is held fixed. RBFs with fixed or limited support radius are called compact support RBFs. Fig. 4.3 displays the overall flow for generating the initial level-set parameters. Each component of Fig. 4.3 has an image associated with it to provide better understanding. The components are expanded on in following subsections.

The general procedure for the initialization scheme is to take the desired outcome, in this case the x-rays of a given topology, and use them to generate initial boundaries (topology). Since this is an initial guess, the boundaries may be jagged or stair-stepped so a linear smoothing algorithm is used to removed the sharp edges. The resulting boundary is broken into equally spaced segments which represent the midpoints of RBF pairs. These RBF pairs consist of an RBF that is defined inside the topology and one that is outside. After distributing the RBFs about the topology (see Fig. 4.10), the coefficients can be solved for, generating the initial level-set parameters.

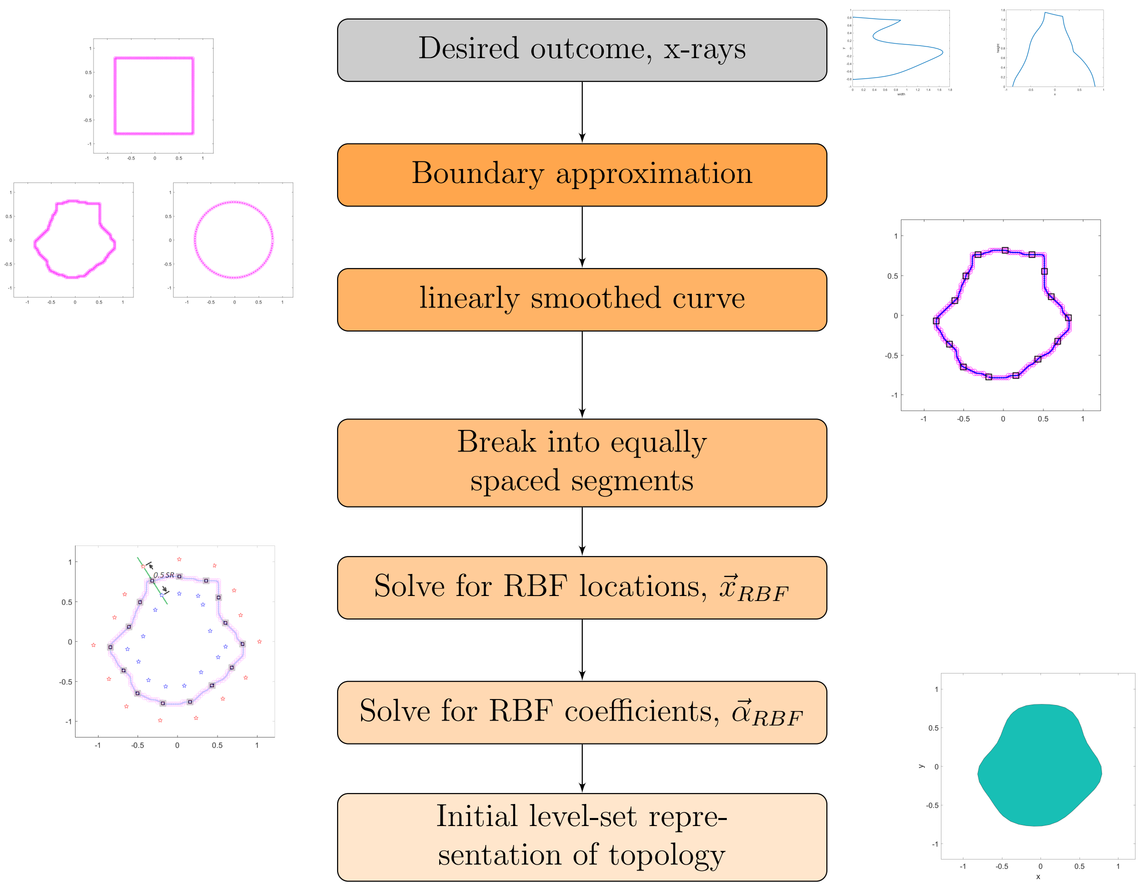

Desired outcome, x-rays
Boundary approximation
linearly smoothed curve
Break into equally spaced segments
Solve for RBF locations, $\vec{x}_{RBF}$
Solve for RBF coefficients, $\vec{\alpha}_{RBF}$
Initial level-set representation of topology
0.5 SR

Figure 4.3: Initialization scheme flow chart

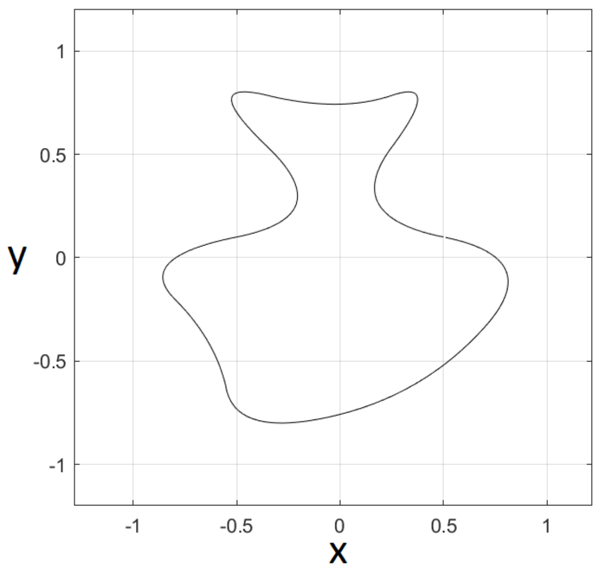
Unfortunately, there is no clear way to generate the initial guess boundary(ies) that determine the initial level-set parameters (i.e., the RBF locations and coefficients). If the initial guess for the RBF locations and coefficients generates an arbitrary body or collection of bodies, the optimizer may never find a combination of the design parameters that truly minimizes the objective function. The relationship between the physical topology and the x-rays is ambiguous, so there will exist local minima throughout the design space. The optimizer may find a configuration where the amount of error in the current iteration is acceptable because no change in design parameter can reduce the objective function, which would result in the optimizer exiting prematurely. However, since the desired x-rays are known, there may be a way to use the given data to generate a guess. Therefore, an additional preprocessing method was developed to create an initial guess.

The desired outcome is used for initialization to attempt to find initial parameters that provide a good guess topology based on the information that is given so the optimizer does not have to start from an arbitrary guess. The objective function used in this work is the sum of the two root-mean-squares of the differences between the x-rays of the zero level-set curve compared to the desired x-rays (see Sec. 4.4.2 for objective function evaluation details). Thus, developing a method to generate a good initial guess (level-set x-rays that approximate the desired x-rays) may be effective in reducing the computational resources necessary for the total optimization procedure.

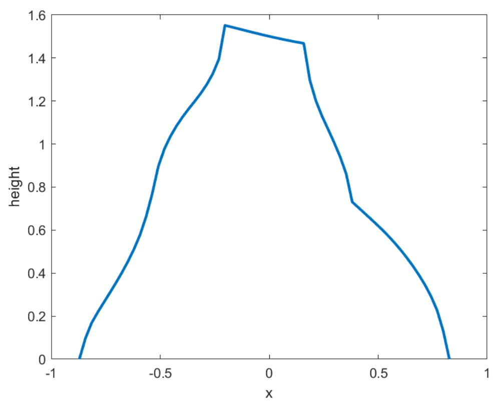
The x-rays of a body or collection of bodies can alternatively be viewed as the height variation with respect to the $x$-axis (the vertical x-rays) and the width variation with respect to the $y$-axis (the horizontal x-rays). An example of the x-rays of an arbitrary body are shown in Fig. 4.4. The procedure for generating the x-rays of any body(ies) is portrayed in Figs. 4.5 and 4.6. A ray is cast in the $y$- or $x$-direction, respectively, and the locations where the ray intersects the body(ies), $\vec{x}_{int}$, are recorded. The distance between point pairs is calculated such that no point is used twice in the distance calculations. For example, if there are four intersection points, the first two are considered a point pair and the third and fourth points are the next point pair. This is illustrated in the middle and bottom figures in Figs. 4.5 and 4.6, respectively, where two lengths of the body are identified by the ray-casting. All the distances are summed to obtain the total width or height of the body(ies) at the ray location, shown in the corresponding figures by the different color lengths in the x-ray plots. Mathematically, the horizontal and vertical x-rays are defined as the variables $h_{ray}$ and $v_{ray}$, respectively, and are calculated using Eqs. 4.14 and 4.15,

$$h_{ray} = \sum_{j=1}^{M} \sqrt{(x_{int,2j-1} - x_{int,2j})}, \tag{4.14}$$

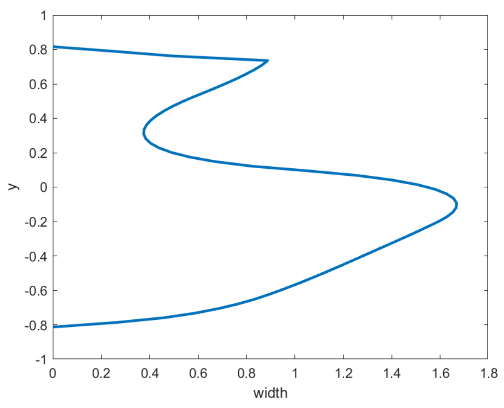
$$v_{ray} = \sum_{j=1}^{M} \sqrt{(y_{int,2j-1} - y_{int,2j})}, \tag{4.15}$$



(a) Initial geometry



(b) Vertical x-rays



(c) Horizontal x-rays

Figure 4.4: Sample x-rays for arbitrary body

where $M$ is the number of intersection points, $\vec{x}_{int,2j-1}$ and $\vec{x}_{int,2j}$ represent $j$th ray intersection pairs. The $y$-coordinate of the horizontal ray, $h_{ray}$, is constant, so the $y$-term in the horizontal x-ray calculation equals zero. Similarly, $x$-term in the vertical x-ray, $v_{ray}$, is zero. Indexing by $2j-1$ and $2j$ has the effect of using the pairs of ray intersection points, as discussed previously. The x-rays for the desired design are stored and compared with the x-rays generated from the level-set representation at each successive optimization iteration to evaluate the objective function.

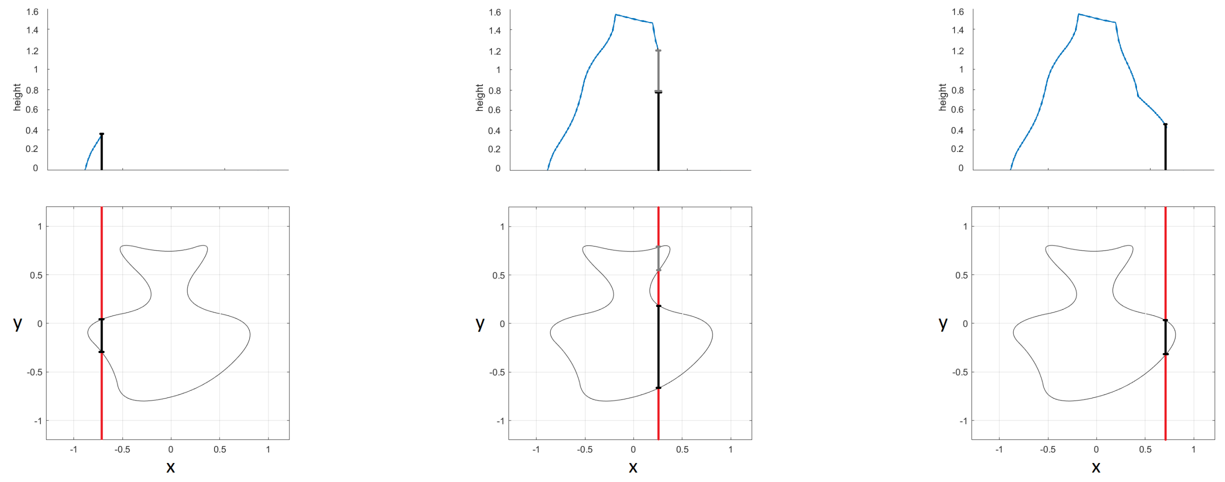


Figure 4.5: Example of generating the vertical x-rays. Raycasting on the bottom, x-ray tracing on the top.

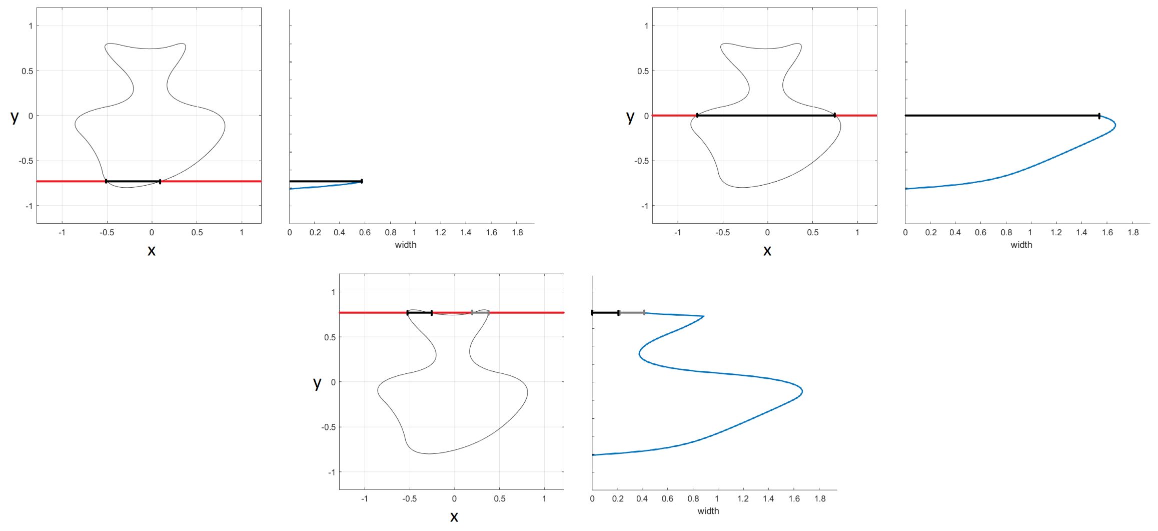


Figure 4.6: Example of generating the horizontal x-rays. Raycasting on the bottom, x-ray tracing on the top.

Three approaches to obtaining an initial guess have been explored to assess its effect on the optimization efficiency, robustness, and accuracy. The results are discussed in the following chapter. The three approaches used to define the RBFs locations and coefficients are to define them along a:

1. bounding box;
2. bounding ellipse;
3. stair-stepped representation.

Each of these approaches only require the x-rays of the topology to generate the initial guess. The bounding box and bounding ellipse guesses are easily obtained by solving for the bounding $x$- and $y$-coordinates of the vertical and horizontal x-rays, respectively. The major axis of the box or ellipse is set by the $x$ range and the minor axis is set by the $y$ range. An example of both of these initial guesses for the arbitrary body shown in Fig. 4.4a are displayed in Figs. 4.7a and 4.7b. A stair-stepped initial guess is one that attempts to approximate the boundary of the design using the desired x-rays and can be seen in Fig. 4.7c.

For the stair-stepped approach, the design domain is broken up into grid cells and the desired x-rays are discretized so they now represent the number of grid cells the design occupies. An example of the discretized x-rays for the arbitrary body are shown in Fig. 4.8. A low-dimensional stair-stepped or block representation is solved for by heuristic optimization techniques using the discretized x-rays. This class of techniques was chosen because of its gradient-free nature and simplicity of implementation. Two arrays are created that store the integer x-ray information for the horizontal and vertical x-rays, respectively. For example, the fourth entry of the horizontal x-ray array would correspond to the fourth row of the design domain grid and the value would indicate how many cells the design occupies within the row. Similarly, the vertical x-ray array corresponds to how full the design grid columns are. In the example shown in Fig. 4.8, the design grid would be 61-by-61 because that is the maximum integer associated with the discrete x-rays. Looking closely at the fourth entry in each plot would show that the design occupies approximately 25 grid cells in the fourth row and the design occupies approximately 10 grid cells in the fourth column. A grid of zeros and ones can be initialized using this information to generate a bounding box, where the grid cell is a one if either entry in the target x-ray arrays is greater than zero, denoting that this location may be inside the design boundary. The x-rays associated with the current state of the grid then become the sum of each column or row. The grid is then iteratively altered such that the outer boundary deforms, while its convexity is maintained, to better approximate the actual design.

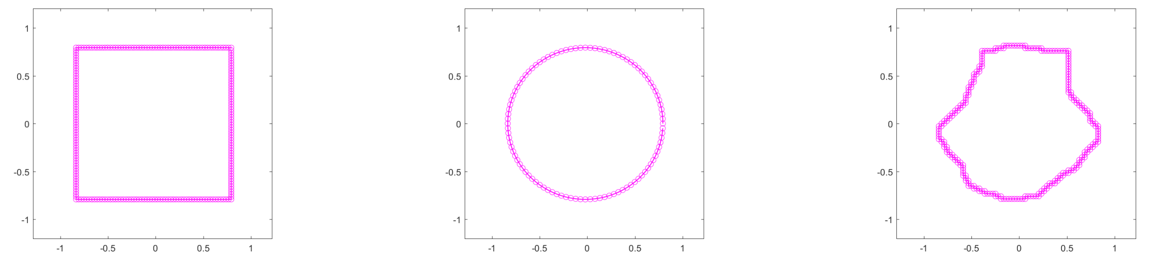
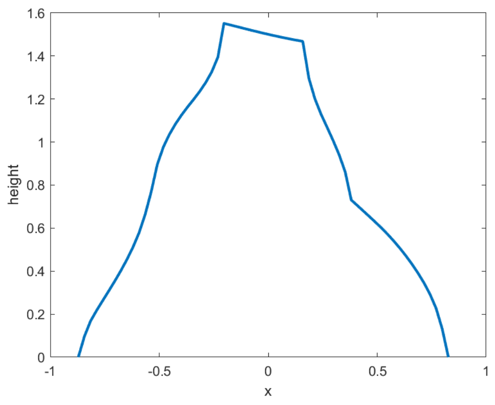

(a) Bounding box guess (b) Bounding ellipse guess (c) Stair-stepped guess

Figure 4.7: Example of initial guesses for RBF midpoint locations for the arbitrary body case

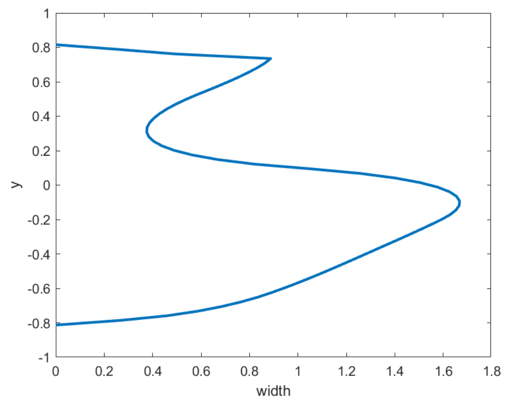
Grid cells are randomly chosen and the surrounding cells are used to determine if the cell is a candidate to turn off. A cell is a candidate if turning it off will maintain the convexity of the local structure. Whether or not the convexity will be maintained is determined by the surrounding eight cells. The configurations that result in a cell being turned off are shown in Fig 4.9.

In all of the cases shown in Fig. 4.9 the chosen cell is turned off. The procedure for choosing a cell and determining if it can be turned on or off is repeated $N^3$ times, where $N$ is the dimension of the discrete grid. This number of iterations gives the algorithm enough attempts to produce a reasonable convex approximation of the design based on the x-rays. As can be seen in Fig. 4.7c, the output from the stair-stepper algorithm misses the concavities in the design. However, this approximation is closer to the desired design than the bounding box or ellipse. Further processing is required for the stair-stepper algorithm because the points produced are not ordered, and to distribute the RBFs based on the boundary, ordered points are required. However, this step adds negligible time to the overall initialization procedure.

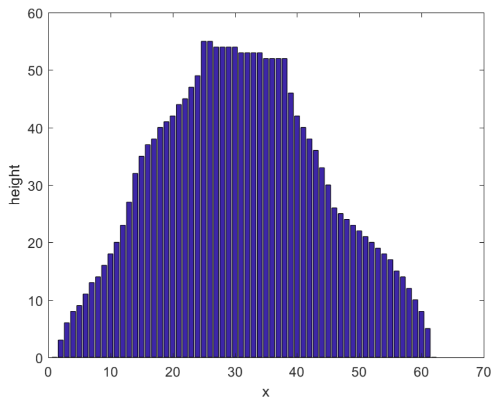
The points along the boundary of the stair-stepped geometry are ordered by starting from an arbitrary point on the boundary and checking the adjacent points to see if they are on the boundary as well. If a point is identified as being on the boundary, it is added as the next point and then the procedure is repeated. Since the points all exist on a predefined



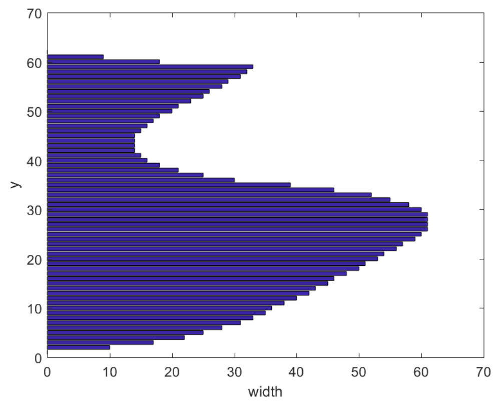
(a) Vertical x-rays



(b) Horizontal x-rays



(c) Discrete Vertical x-rays



(d) Discrete Horizontal x-rays

Figure 4.8: Discretized x-rays for initial guess

grid, the order of checking is left, down, right, up for the first point. This particular order ensures that the algorithm goes around the body and does not get stuck in a loop when two points on either side of the body are within a grid spacing of one another. After the second point is found, the next search is a permutation of the order left, down, right, up, based on the direction that was used to advance to the second point. For example, if the second point was right of the first point the algorithm would look down, right, and up, but never left because it is constrained to never go backwards.

The ordered stair-stepped representation of the topology is then smoothed using several linear smoothing passes. Linear smoothing is performed by averaging the coordinate values

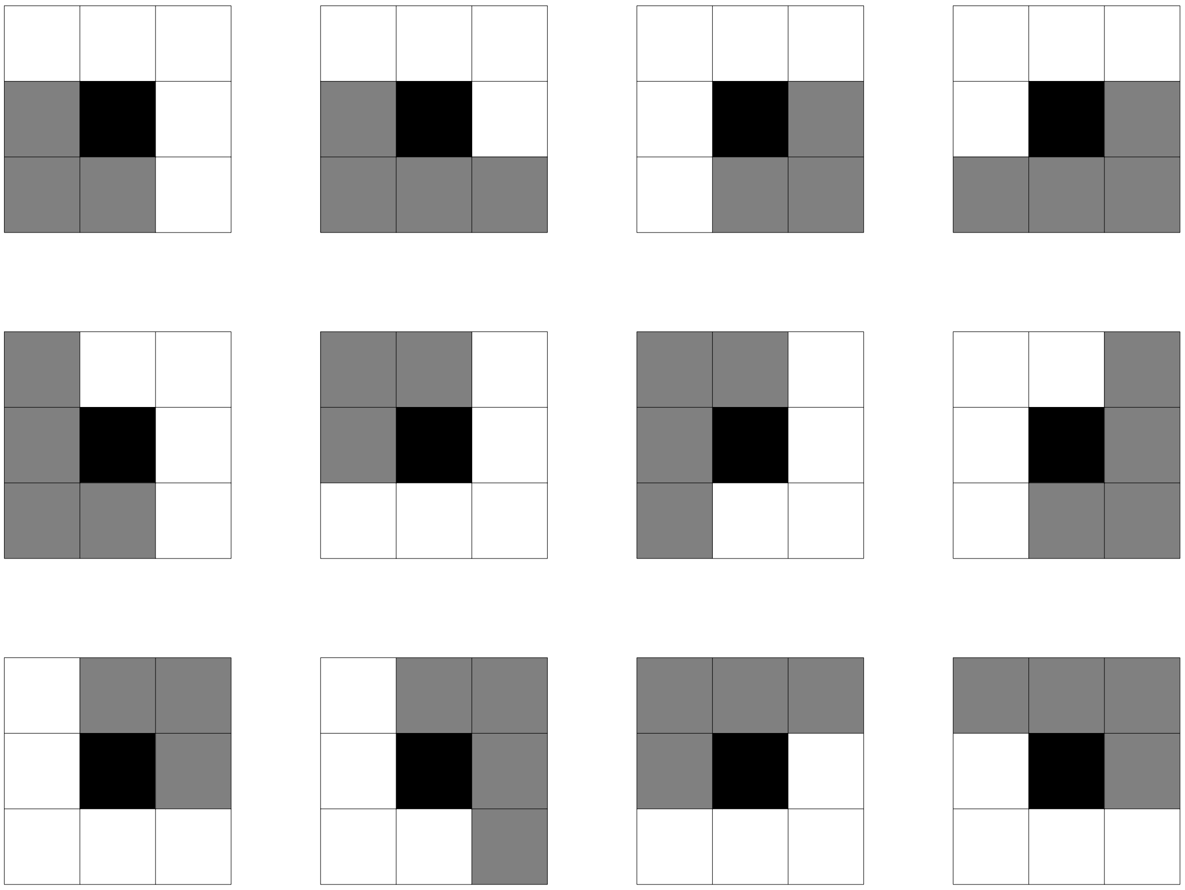


Figure 4.9: Acceptable convexities for stair-stepper algorithm. Black is the chosen cell that is "on," gray surrounding cells are "on" and white cells are "off."

of the $jth$ point and its neighbors, shown in Eq. 4.16,

$$x_j^* = \frac{x_{j-1} + 2x_j + x_{j+1}}{4}, \tag{4.16}$$

where $x_j^*$ is the smoothed point and $x_j$ are the points currently along the boundary. The same can be shown for the $y$-coordinate. All of the new coordinates are calculated using the old coordinates and then the boundary is updated. This prevents any biasing in the smoothing procedure. Smoothing the stair-step removes any shape edges from the initial design and aims to improve the level-set representation. The new smoothed curve is then used to inform where to place the RBFs for the LSF.

Fig. 4.10 shows an example of placing the RBFs. The RBF locations are determined by approximating the normal direction at each location and then placing an RBF some distance along the positive and negative normal direction. The distance is determined by the support radius of the RBFs as shown in Fig. 4.10d. The normal is approximated by taking the central difference about the $i^{th}$ RBF to calculate the tangent and then calculating the normal from the tangent vector. The central difference is simply the difference between the $i+1$ and $i-1$ $x$- and $y$-coordinates. The approximate tangent vector at the $i^{th}$ RBF is then $\vec{t} = dx\hat{\imath} + dy\hat{\jmath}$ and the normal is $\vec{n} = -dy\hat{\imath} + dx\hat{\jmath}$.

The LSF is defined such that it is positive within the body(ies) and negative outside the body(ies). Thus, the zero level-set curve is the boundary of the body(ies).

$$\textit{inside topology}, \phi \quad > 0; \tag{4.17a}$$

$$\textit{outside topology}, \phi \quad < 0; \tag{4.17b}$$

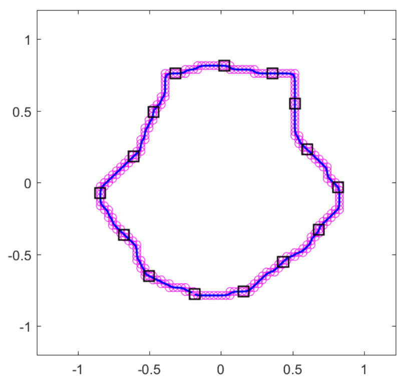
$$\textit{on topology boundary}, \phi \quad = 0. \tag{4.17c}$$

To have a clear delineation between positive and negative level-set and to avoid the entire domain outside the body(ies) being zero, the LSF is shifted by some arbitrary value. For all the following examples and results, the level-set is shifted by +1 so the modified LSF is now,

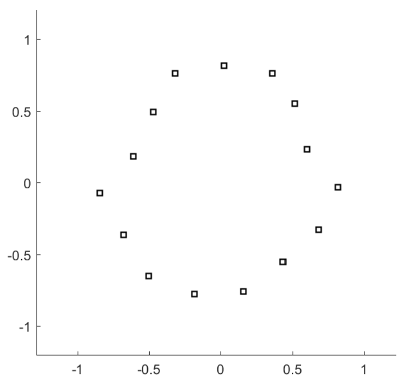
$$\phi(\alpha, \psi(\xi)) = \sum_i \alpha_i \psi_i(\xi); \tag{4.18a}$$

$$\tilde{\phi}(\alpha, \psi(\xi)) = \sum_i \alpha_i \psi_i(\xi) - 1. \tag{4.18b}$$

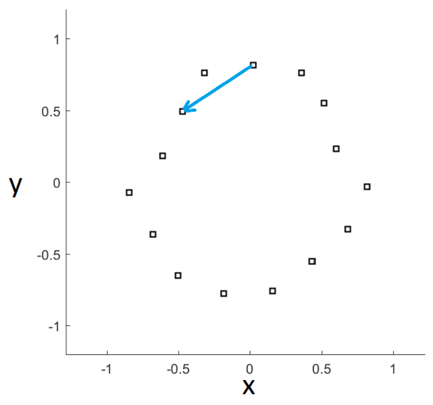
The shift value is referred to as the offset in the results section as well as in the algorithms. The level-set curve that is used for objective evaluation is $\phi = 0$ which can be referred to as the waterline because it determines how much of the LSF is "seen" in the two-dimensional plane — similar to looking down at islands in the ocean. The LSF value at the RBFs inside the body(ies) is set to the offset value to ensure that the function will be positive within.



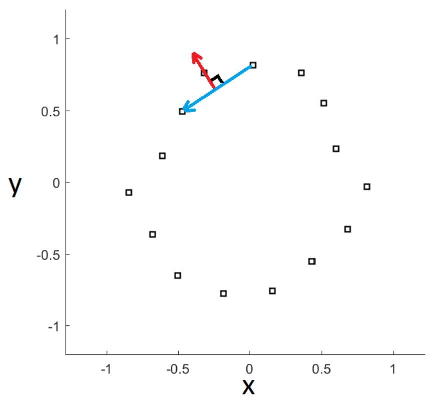
(a) Evenly distributed points along the initial curve



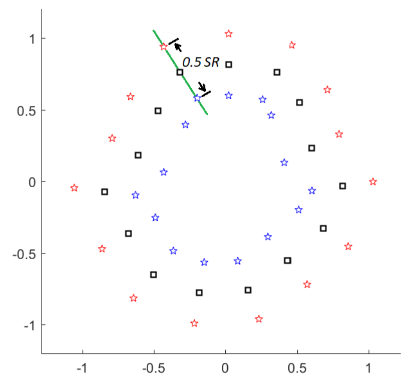
(b) Coordinates used for the midpoints of the inside and ouside RBFs



(c) Approximate tangent vector based on the neighboring points



(d) Approximate normal vector based on the tangent



(e) Final distribution of RBFs

Figure 4.10: Example of initializing RBF locations

The level-set value at the outside RBF locations is set to the negative of the water-level. This setup ensures there is a zero-crossing between the inside and outside RBFs. Fig. 4.11 shows the resulting level-set representation of the arbitrary body example. The level-set parameters calculated from this initialization procedure can be input into an optimization algorithm that calculates the objective function and design sensitivities to change the design parameters using gradient-based optimization techniques.

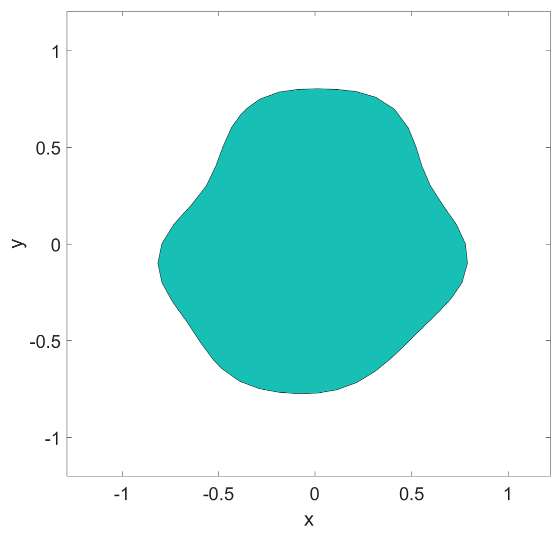


Figure 4.11: Level-set representation of the arbitrary body example

## 4.4 The optimization procedure

Gradient-based optimization was used in this work to take advantage of the analytic description of the design topology by the LSF. However, the LSF describes the topology implicitly so as to be able to calculate the objective function as well as its derivatives (design sensitivities); but the zero level-set curve needs to be extracted explicitly. Fig. 4.12 illustrates a flow chart that describes the optimization procedure. The initial level-set parameters are input and points along the zero curve are found to provide a starting location for an adaptive RK4 algorithm to march around the curve and generate a closed body. The points along the

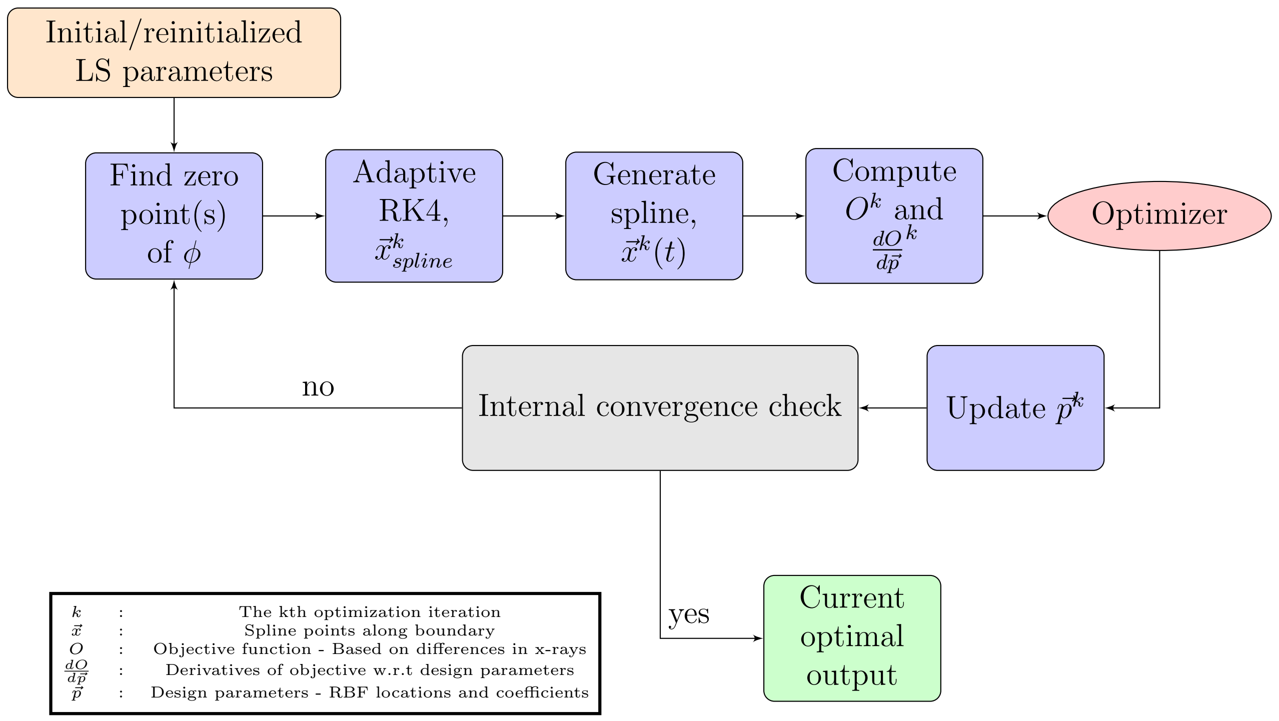

Initial/reinitialized LS parameters
Find zero point(s) of $\phi$
Adaptive RK4, $\vec{x}^k_{spline}$
Generate spline, $\vec{x}^k(t)$
Compute $O^k$ and $\frac{dO}{d\vec{p}}^k$
Optimizer
Update $\vec{p}^k$
Internal convergence check
no
yes
Current optimal output
$k$ : The kth optimization iteration
$\vec{x}$ : Spline points along boundary
$O$ : Objective function - Based on differences in x-rays
$\frac{dO}{d\vec{p}}$ : Derivatives of objective w.r.t design parameters
$\vec{p}$ : Design parameters - RBF locations and coefficients

Figure 4.12: Optimization scheme flow chart

body are used to fit a spline through and produce a curvature continuous body. Generating these bodies is performed one at a time so once a body is found, the zero curve associated with it is neglected in future zero point finding passes. Once all the bodies are found and extracted, the objective function is evaluated along with its derivatives with respect to each design variable. These derivatives are passed to a conjugate gradient optimization algorithm that calculates the search direction and step size in that direction using a golden section search technique (described in Sec. 4.1). These quantities are used to update the design and the convergence of the design is tested. This process continues until convergence is reached and the optimization procedure exits. Each component is expanded upon in the following subsections.

### 4.4.1 Extracting the topology

The overall method for extracting (discretizing) the zero level-set curve can be split into two procedures,

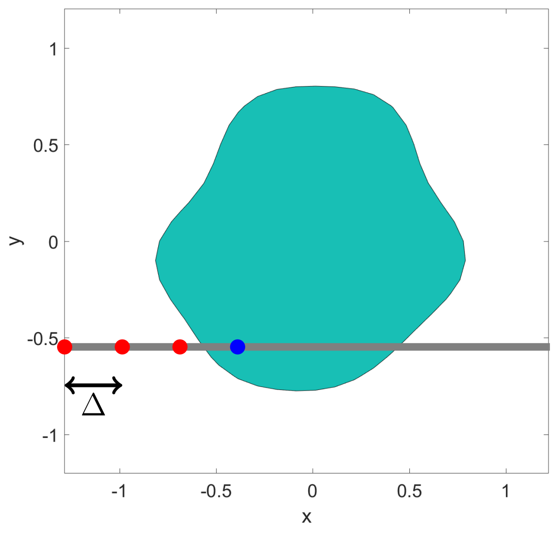
1. finding an initial point along the zero curve;
2. solving the system of ordinary equations to march around the curve.

The initial point can be found by numerous methods, but the one chosen for this work discretizes the domain into a grid and checks two adjacent points until a zero-crossing is bracketed and then uses the bisection method to iterate to the zero location. Once the zero location is determined the zero level-set curve can be discretized using an algorithm for numerically solving systems of ordinary differential equations (ODEs). Both parts of the method used to find the zero level-set curve are explained below using the arbitrary level-set body in Fig. 4.11 to illustrate the procedure.

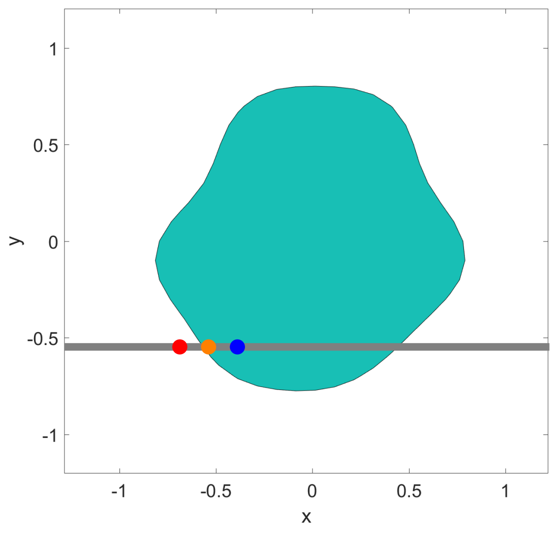
**Locate a zero-crossing**

The zero-crossing can be found using a bracketing and bisection technique. A horizontal line is drawn through the domain and discretized into equally spaced points. The spacing is predetermined and is chosen such that the smallest expected geometries can be identified. A spacing of $\Delta x = \Delta y = \Delta = 0.1$ was chosen for the following examples.

An example of the zero location procedure is shown in Fig. 4.13. The LSF is evaluated at two points starting from the left and marching to the right. If the LSF value at each point is the same sign the rightmost point is kept and the next point is evaluated. This procedure is repeated until both points have opposite sign. Since the LSF is continuous, a change in sign implies that the LSF crosses zero between the two points. The bisection method for finding the zero of a function is implemented once the zero is effectively bracketed. Bisection takes the midpoint of the two bracketing point values, in this case $x_{left}$ and $x_{right}$, and evaluates the LSF at that location. Then that value is multiplied by either the value at $x_{left}$ or $x_{right}$. If either value is positive, that means the LSF value at the midpoint shares the same sign as the respective point and should replace it because there is no zero-crossing between them. For example, suppose $x_{right}$ were the point chosen to test against; if the product of the LSF value at the midpoint and the LSF value at the right point were positive, the values share



(a)



(b)

Figure 4.13: (a) Bracketing with red and blue denoting negative and positive LSF values, respectively, and (b) bisection example with red as $x_{left}$, blue as $x_{right}$, and orange as $x_{mid}$.

the same sign and, therefore, $x_{right}$ should be replaced by $x_{mid}$ because the zero-crossing is between $x_{mid}$ and $x_{left}$, with $x_{right}$ being further away from the zero-crossing. The procedure continues with $x_{mid}$ being updated at each step until the difference between $x_{left}$ and $x_{right}$ falls below some tolerance. The zero-crossing is taken to be at the midpoint of these two values. The horizontal lines are drawn from the lower left of the domain and if no crossing is found the line is moved up by $\Delta$ and the process starts all over.

With a zero point on the level-set curve located, it can be defined as the initial point on the zero level-set curve that an ordinary differential equation solver, such as an Runge-Kutta algorithm, can be used to identify points along the curve. This will effectively define the spline points that are used to fit a cubic spline through to produce a curvature continuous design boundary. This boundary is then used to evaluate the objective function.

**The adaptive 4th-order Runge-Kutta algorithm**

Solving systems of ordinary differential equations is not a new concept. These types of algorithms have been around for quite some time with the oldest dating back to Euler's method for solving ODEs. The Runge-Kutta (RK) method improves on this classical technique by using several stages to calculate the solution. Various formulations for the RK method have ben documented by Abramowitz and Stegun[121] and Gear[122]. The problem of finding the points along the zero level-set curve can be framed as a system of ODEs,

$$\frac{dx}{ds} = \frac{d\phi}{dx} \tag{4.19a}$$
$$\frac{dy}{ds} = \frac{d\phi}{dy} \tag{4.19b}$$

where $s$ is the distance along the curve, $x$ and $y$ are spatial coordinates, and $\phi$ is the LSF. The most commonly used Runge-Kutta method is the 4th order Runge-Kutta scheme referred to as an RK4. The general solution takes the following form,

$$\frac{dx}{ds} = \frac{d\phi(s,x)}{dx} = \phi'(s,x) \tag{4.20a}$$
$$x(s_0) = x_0 \tag{4.20b}$$
$$k_1 = h\phi'\left(s_n, x_n\right) \tag{4.20c}$$
$$k_2 = h\phi'\left(s_n + \frac{h}{2}, x + \frac{k_1}{2}\right) \tag{4.20d}$$
$$k_3 = h\phi'\left(s_n + \frac{h}{2}, x + \frac{k_2}{2}\right) \tag{4.20e}$$
$$k_4 = h\phi'\left(s_n + h, x_n + k_3\right) \tag{4.20f}$$
$$x_{n+1} = x_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{4.20g}$$

for $n = 0, 1, 2, \ldots$, until the algorithm reaches its maximum number of iterations or in this case if the curve is closed. Since there is a second ODE, making this a system of ODEs that are being solved, a second set of equations almost identical to Eq. 4.20 is solved simultaneously of the form,

$$\frac{dy}{ds} = \frac{d\phi(s,y)}{dy} = \phi'(s,y) \tag{4.21a}$$

$$y(s_0) = y_0 \tag{4.21b}$$

$$m_1 = h\phi'\left(s_n, y_n\right) \tag{4.21c}$$

$$m_2 = h\phi'\left(s_n + \frac{h}{2}, y + \frac{m_1}{2}\right) \tag{4.21d}$$

$$m_3 = h\phi'\left(s_n + \frac{h}{2}, y + \frac{m_2}{2}\right) \tag{4.21e}$$

$$m_4 = h\phi'\left(s_n + h, y_n + m_3\right) \tag{4.21f}$$

$$y_{n+1} = y_n + \frac{1}{6}\left(m_1 + 2m_2 + 2m_3 + m_4\right) \tag{4.21g}$$

so both $x$ and $y$ are updated with one iteration of the RK4 algorithm. Described here is the basic RK4 algorithm, in which $h$ is a fixed step and therefore can be inaccurate if too large a step is chosen for the solver. Furthermore, there is no control over the spacing on the points because of the fixed step size and consequently this algorithm is unsuitable for geometries and topologies with high curvature as seen in most aerodynamic applications. An adaptive RK4 is implemented to accurately capture areas of high curvature as well as appropriately distribute points along the curve, clustering them in areas of high curvature and spreading them out in areas of low curvature, as is general practice for CFD meshing. The conventional adaptive RK4[122] scheme is performed by taking one step of the RK4 taken with a step-size $h$ and then two steps with step-size $\frac{h}{2}$ and evaluating the distance between the two points. If the distance between these two points is less than a prescribed tolerance, then the two steps are accepted and the step-size $h$ is doubled to check if a larger step can be taken. This procedure is illustrated by Eqs. 4.22 One step at h:

$$x^h_{n+1} = x_n + \frac{1}{6}\left(k^h_1 + 2k^h_2 + 2k^h_3 + k^h_4\right) \tag{4.22a}$$

$$y^h_{n+1} = y_n + \frac{1}{6}\left(m^h_1 + 2m^h_2 + 2m^h_3 + m^h_4\right) \tag{4.22b}$$

two steps at h/2:

$$x_{n+1}^{\frac{h}{2}} = x_n + \frac{1}{6}\left(k_1^{\frac{h}{2}} + 2k_2^{\frac{h}{2}} + 2k_3^{\frac{h}{2}} + k_4^{\frac{h}{2}}\right) \tag{4.22c}$$

$$y_{n+1}^{\frac{h}{2}} = y_n + \frac{1}{6}\left(m_1^{\frac{h}{2}} + 2m_2^{\frac{h}{2}} + 2m_3^{\frac{h}{2}} + m_4^{\frac{h}{2}}\right) \tag{4.22d}$$

$$x_{n+2}^{\frac{h}{2}} = x_{n+1}^{\frac{h}{2}} + \frac{1}{6}\left(k_1^h + 2k_2^h + 2k_3^h + k_4^h\right) \tag{4.22e}$$

$$y_{n+2}^{\frac{h}{2}} = y_{n+1}^{\frac{h}{2}} + \frac{1}{6}\left(m_1^h + 2m_2^h + 2m_3^h + m_4^h\right) \tag{4.22f}$$

distance between points:

$$d^h = \sqrt{(x_{n+1}^h - x_{n+2}^{\frac{h}{2}})^2 + (y_{n+1}^h - y_{n+2}^{\frac{h}{2}})^2}. \tag{4.22g}$$

This type of adaptive RK4 is useful when the analytic description of the function is unavailable. However, the LSF is known, so the two-step adaptation is unnecessary because the LSF itself can be used to determine the accuracy of the initial step.

**The adaptive 4th-order Runge-Kutta method for level-set curves**

Instead of using a multi-step process and checking the distance between the final points to determine the accuracy of the step, the normal distance to the level-set curve the point generated from the initial step can be approximated using a Taylor series expansion about the new point. The expansion can be seen in Eq. 4.23

$$\phi(x + \Delta x, y + \Delta y) = \phi(x, y) + \Delta x\frac{\partial \phi}{\partial x} + \Delta y\frac{\partial \phi}{\partial y} + H.O.T., \tag{4.23}$$

the higher-order terms are neglected. The equation can be rearranged as well as simplified since $\phi(x + \Delta x, y + \Delta y) = 0$ and $\Delta x$ and $\Delta y$ are constrained to be along the normal direction. Thus, $\Delta x$ and $\Delta y$ can be rewritten in terms of the gradient of the LSF and a scaling parameter, $\Delta\eta$,

$$\Delta x = \Delta\eta \frac{\partial \phi}{\partial x} \tag{4.24}$$

$$\Delta y = \Delta\eta \frac{\partial \phi}{\partial y} \tag{4.25}$$

which can be substituted into the Taylor series expansion and simplified as,

$$\phi(x + \Delta x, y + \Delta y) = \phi(x, y) + \Delta x \frac{\partial \phi}{\partial x} + \Delta y \frac{\partial \phi}{\partial y} \tag{4.26}$$

$$-\phi(x, y) = \Delta\eta \frac{\partial \phi}{\partial x}\frac{\partial \phi}{\partial x} + \Delta\eta \frac{\partial \phi}{\partial y}\frac{\partial \phi}{\partial y} \tag{4.27}$$

$$-\phi(x, y) = \Delta\eta \left[ \left( \frac{\partial \phi}{\partial x} \right)^2 + \left( \frac{\partial \phi}{\partial y} \right)^2 \right] \tag{4.28}$$

$$\Delta\eta = \left| -\frac{\phi(x, y)}{\left(\frac{\partial \phi}{\partial x}\right)^2 + \left(\frac{\partial \phi}{\partial y}\right)^2} \right| \tag{4.29}$$

where the absolute value is taken because only the magnitude is required to evaluate the accuracy of the step. The parameter $\Delta\eta$ represents a first-order approximation of the normal distance from the point $(x, y)$ to the zero level-set curve. If this distance is less than a prescribed tolerance, then the step is accepted and a another step is taken at the same size. The tolerance used for this algorithm is chosen to be $\frac{1}{1000}$. If the distance is greater than the tolerance, the number of steps is doubled and the step-size is halved. This prevents the algorithm from decreasing the step-size until it is infinitesimally small and reducing the overall efficiency of the algorithm. However, if the number of steps for a particular iteration exceeds 16 (four failures), the initial step-size is reduced by ten percent and the RK4 algorithm restarts from the first point. Again, this is an effort to prevent the algorithm from generating too many points in one segment and effectively reducing the efficiency of the overall algorithm. After several points have been generated, the algorithm begins testing whether the curve has closed or not. The checks in place to determine if the curve has closed are:

1. calculating the distance between the current and the first point and the first and second point, then comparing the distances. If the distance between the current and first point is less than twice the distance between the first and second point, the curve is assumed closed;

2. calculating the distance between the current and the previous point and the current and the first, then comparing the distances. If the distance between the current and first point is less than the distance between the current and previous point, the curve is assumed closed;

3. determine whether the current line segment has intersected the first line segment.

The first and second conditions are admittedly arbitrary, but they use the fact that the level-set curve is closed and the derivatives along the curve are continuous. Therefore, the points will approach the initial point and these two criteria aim to exit the algorithm when the points are deemed close enough. The third condition, again, uses the idea that the level-set curve is closed and considers the situation where the new point is generated and crosses over the first. However, it certain cases, such as when the level-set produces a small body, these criteria are insufficient and thus there is certainly room for improvement in this algorithm for future work. Once the points are distributed along the level-set curve a cubic spline is fit through them.

### Extracting curvature continuous topologies

The general form of a cubic spline is shown in Eq. 4.30,

$$x(t) = Ax_j + Bx_{j+1} + Cx''_j + Dx''_{j+1} \tag{4.30a}$$

where,

$$A = \frac{t_{j+1} - t}{t_{j+1} - t_j} \tag{4.30b}$$

$$B = 1 - A \tag{4.30c}$$

$$C = \frac{A^3 - A}{6} \left(t_{j+1} - t_j\right)^2 \tag{4.30d}$$

$$D = \frac{B^3 - B}{6} \left(t_{j+1} - t_j\right)^2 . \tag{4.30e}$$

Since the second derivatives at each point are not known beforehand, they must be solved for and then used in Eq. 4.30 to generate the cubic spline(s) that represents the zero level-set curve(s). Taking the derivative of Eq. 4.30 with respect to $t$ and setting the derivatives at each $jth$ point generates a system of equations of the form,

$$\frac{t_j - t_{j-1}}{6} x''_{j-1} + \frac{t_{j+1} - t_{j-1}}{3} x''_j + \frac{t_{j+1} - t_j}{6} x''_{j+1} = \frac{x_{j+1} - x_j}{t_{j+1} - t_j} - \frac{x_j - x_{j-1}}{t_j - t_{j-1}}. \tag{4.31}$$

Eq. 4.30 represents a periodic tridiagonal system of equations and can be used to solve for the second derivatives, $\vec{x}''$ which are needed to represent the resulting cubic spline. Traditionally, tridiagonal systems can be solved efficiently using the Thomas algorithm[123]. However, those systems have zeros for the entries in the first row, last column and last row first column of the system matrix and referred to as aperiodic. A periodic tridiagonal system has non-zero entries in these locations. An example of both aperiodic and period tridiagonal systems can be found in Fig. 4.14. A modified version of the Thomas algorithm using the Sherwood-Morrison formula[124] must be used to solve the periodic tridiagonal system for the second derivatives. The Sherwood-Morrison formula is used to transform the given periodic tridiagonal system into an aperiodic one by defining a new matrix,

$$A' = A - \vec{u}\vec{v}^T, \tag{4.32}$$

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & & 0 \\ a_2 & b_2 & c_2 & 0 & & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \vdots \\ \vdots & & & \ddots & & 0 \\ & & & a_n-1 & b_n-1 & c_n-1 \\ 0 & \dots & & 0 & a_n & b_n \end{bmatrix}$$

(a) Aperiodic

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & 0 & a_1 \\ a_2 & b_2 & c_2 & 0 & & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \vdots \\ \vdots & & & \ddots & & 0 \\ 0 & & & a_n-1 & b_n-1 & c_n-1 \\ c_n & 0 & \dots & 0 & a_n & b_n \end{bmatrix}$$

(b) Periodic

Figure 4.14: Example of different tridiagonal systems of equations

where $A$ is the periodic tridiagonal coefficient matrix formed from Eq. 4.30,

$$\vec{u}^T = [-b_1 \;\; 0 \; \dots \; 0 \;\; c_n]\,, \tag{4.33}$$

and

$$\vec{v}^T = [1 \;\; 0 \; \dots \; 0 \;\; a_1/b_1]\,. \tag{4.34}$$

The original problem can be written at $(A'+uv^t)\vec{x''} = \vec{d}$ where $(A'+uv^T)$ is $A$ by definition, $\vec{x''}$ represents the second derivatives in Eq. 4.31, and $\vec{d}$ represents the right-hand-side of Eq. 4.31. Two separate systems are solved using the Thomas algorithm, $A'\vec{y} = \vec{d}$ and $A'\vec{q} = \vec{u}$, and then $\vec{x}''$ can be found from the following equation,

$$\vec{x''} = \vec{y} - \left[\frac{(\vec{v}^T\vec{y})}{1+(\vec{v}^T\vec{q})}\right]. \tag{4.35}$$

The same approach can be used for the $y$-coordinate, as well. Combined, these two coordinates generate the cubic fit of the zero level-set curve. The midpoint of each segment, calculated at $t = 0.5$, is used to check the accuracy of the fit curve and determine if the curve can be used as a good representation of the level-set body(ies). The accuracy is determined in the same way as the RK4 algorithm. The normal distance to the level-set curve is approximated using the Taylor series expansion. If any of the values are greater than the tolerance, the initial step-size is halved and the RK4 algorithm is repeated. This procedure is continued until a curve with the allowable tolerances at the checkpoints is generated.

Once a boundary is identified and meets the appropriate criteria, the whole procedure is repeated to determine if there exists another body within the domain. With this technique, since only the LSF is used to identify where the boundaries are, an additional check after the first boundary is found is required. On successive passes through the domain after the first one, if two points are found to bracket a zero location both are tested to determine whether they are within an existing boundary or not. If one bracketing point is found within the boundary this means that the zero point found lies on an existing boundary. Alternately, if both bracketing points are found within the boundary then that means there exists a hole with the boundary. Since the aim of this work is to produce curvature continuous boundaries for fluid flow applications, the main focus is the outer boundary of the LSF and any internal holes are neglected because they will not have an effect on the objective function.

After all the outer boundaries of the zero level-set curves have been identified and discretized the objective function can be evaluated.

### 4.4.2 Objective function evaluation

The objective is to match the height and width distributions by minimizing the sum of the root-mean-square differences between both profiles, respectively. An example of the x-rays from the desired body(ies) and those obtained from the level-set representation can be seen in Fig. 4.15. The green region is the portion of the x-rays that is used to evaluate how well they match each other.

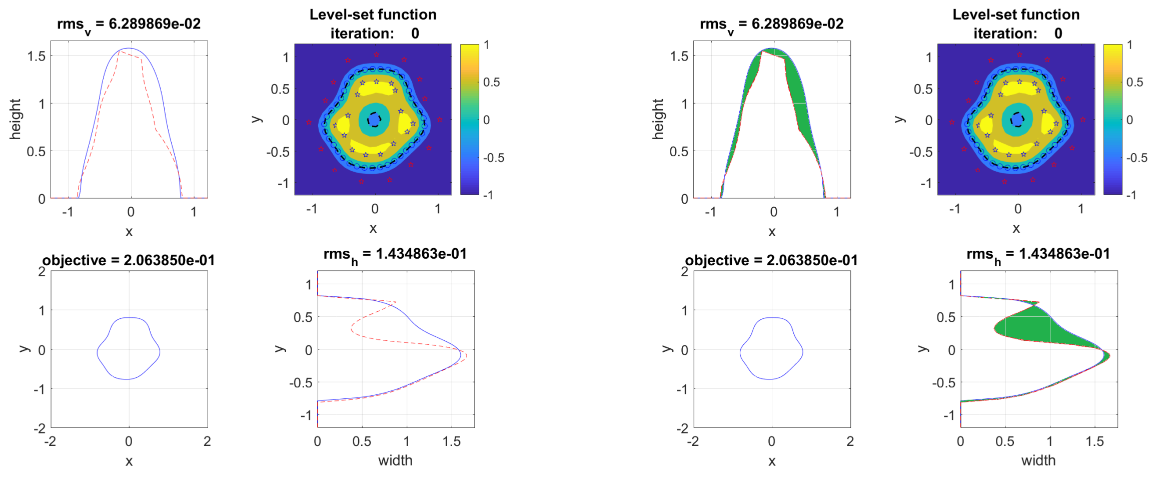


Figure 4.15: Comparison of x-rays for objective function evaluation. The high-lighted green region represents the difference between the x-ray curves.

The RMS values used to determine the error between x-rays is defined as,

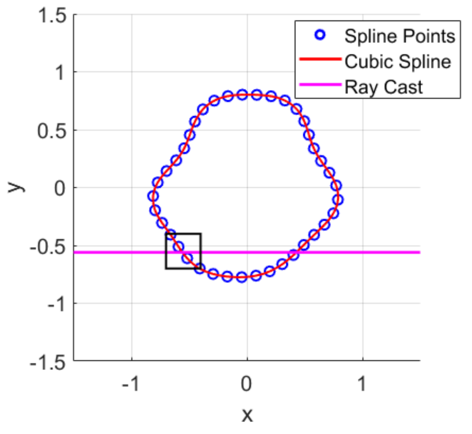
$$RMS_h = \sqrt{\frac{\sum_i^{N_{rays}} \left(h_{ray,LSF,i} - h_{ray,DES,i}\right)^2 dy_{ray,i}}{N_r ays}}, \tag{4.36a}$$

$$RMS_v = \sqrt{\frac{\sum_i^{N_{rays}} \left(v_{ray,LSF,i} - v_{ray,DES,i}\right)^2 dx_{ray,i}}{N_r ays}}, \tag{4.36b}$$

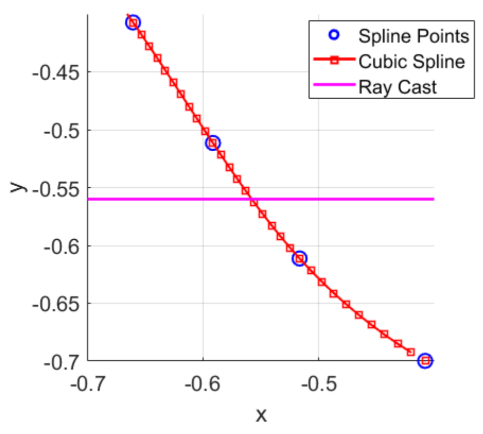
$$O(\vec{p}) = RMS_h + RMS_v, \tag{4.36c}$$

where $O$ denoting the objective function and the terms containing $h$ or $v$ are the horizontal or vertical ray values, respectively. In Eq. 4.36 the $LSF$ terms are the level-set values and the $DES$ terms are the desired values. Furthermore, the squared differences are scaled by the distance between the respective rays, so the change in $y$-coordinate for the horizontal x-rays, $dy_{ray,i}$, and the change in $x$-coordinate for the vertical, $dx_{ray,i}$. The calculation of the values $h$ and $v$ have been briefly described in section 4.3 and here it is explained more thoroughly.

Consider Fig. 4.16, the spline points from the RK4 algorithm are plotted as blue circles and the red squares are points along each cubic spline segment. The magenta line represents a ray cast through the domain. The ray intersection points are calculated by using the



(a) Cubic spline points and resulting curve



(b) Zoomed in to see points along cubic spline

Figure 4.16: Ray passing through level-set design curve

intersection of two lines: the ray being cast through the domain and the line connecting the two points it passes through. In other words, this method uses linear approximations to determine the intersection points. Instead of checking whether the ray intersects each line segment, an initial check is performed that determines if the ray exists between the end points of the line segment. Basically, for the case shown in Fig. 4.16, if the $y$-coordinate of the ray does not lie between the $y$-coordinates of a particular line segment, that segment is excluded from the intersection calculation. Furthermore, notice that the cubic spline points are spaced relatively far apart. Solely using them to find the intersection points could result in inaccuracies because a linear approximation is being used for a cubic spline. Instead, each cubic is sub-sampled, which is shown in Fig. 4.16b. The sub-sampling refines the curve and provides a more accurate description of the cubic spline between spline points, even though the intersection points are still calculated using a linear approximation of the curve. Once calculated, the intersection points can be used to calculate the x-ray values as described previously in section 4.3. With the objective function evaluated the next step is to determine how the value changes with respect to changes in the design parameters, often referred to as calculating the design sensitivities.

### 4.4.3 Calculating design sensitivities

The design sensitivities are the derivatives of the objective function with respect to the design variables, $\frac{\partial O}{\partial \vec{p}}$. The design variables are the locations of the RBFs, $\vec{x}_i$, and their coefficients, $\vec{\alpha}$. The design sensitivities are calculated by taking the derivative of the each step of the optimization process. The chain rule can be applied to $\frac{\partial O}{\partial \vec{p}}$ to obtain the various derivatives that are necessary. This expansion is demonstrated in Eq. 4.38,

$$\frac{\partial O}{\partial \vec{p}} = \left(\frac{\partial O}{\partial \vec{x}_{rays}}\right)\left(\frac{\partial \vec{x}_{rays}}{\partial \vec{x}_{spline}}\right)\left(\frac{\partial \vec{x}_{spline}}{\partial \phi}\right)\left(\frac{\partial \phi}{\partial \vec{p}}\right), \tag{4.37}$$

where $\frac{\partial O}{\partial \vec{x}_{int}}$ represents how the objective function changes with respect to the ray intersection points, $\frac{\partial \vec{x}_{int}}{\partial \vec{x}_{spline}}$ represents how the objective function changes with respect to the cubic spline points, $\frac{\partial \vec{x}_{spline}}{\partial \phi}$ represents the change in the cubic spline points with respect to the LSF, and $\frac{\partial \phi}{\partial \vec{p}}$ represents the change in the LSF with respect to the design parameters. Each of these derivatives can be calculated in a variety of ways. The most common are finite-difference, tangent linear, and adjoint. To understand the differences between each method it is informative to view Eq. 4.38 as a the multiplication of matrices often referred to as Jacobians,

$$\left[\frac{\partial O}{\partial \vec{p}}\right]_{[1\times N]} = \left[\frac{\partial O}{\partial \vec{x}_{int}}\right]_{[1\times 2K]}\left[\frac{\partial \vec{x}_{int}}{\partial \vec{x}_{spline}}\right]_{[2K\times M]}\left[\frac{\partial \vec{x}_{spline}}{\partial \phi}\right]_{[M\times N]}\left[\frac{\partial \phi}{\partial \vec{p}}\right]_{[N\times N]}, \tag{4.38}$$

where $N$ is the number of design parameters, $M$ is the number of spline points and $K$ is the number of rays used to generate the x-rays. The multiplier of 2 is present because the same number of rays are used for generating the vertical and horizontal x-rays. The multiplication of Jacobians is rarely used in practice because it requires generating often large matrices which can use up large amount of computer memory which is inefficient. Alternatively, $\frac{\partial O}{\partial \vec{p}}$ can be calculated by stepping through the objective function calculations forwards or backwards.

The three methods for calculating the derivatives listed can be separated into two one of the two categories: forward methods and backward methods. Finite-difference and tangent linear methods require the derivatives to be calculated forward, which means the entire objective function calculation needs to be stepped through to calculate the derivatives with respect to each variable. Therefore, the derivative calculations scale with the number of design variables. By perturbing one variable at a time, each successive derivative can be calculated with the end result being the change in the objective function with respect to each variable. Fig. 4.17 shows a flow chart describing the forward method for derivative calculations where $\dot{( )}$ is a derivative with respect to the design variables, $\vec{p}$. First, the derivatives of the LSF with respect to the design variables is calculated. These derivatives are then used to calculate the derivatives of the cubic spline points with respect to the design variables. Next, the derivatives of the ray intersection points with respect to the design variables can be obtained and used to calculate the overall derivatives of the objective function with respect to the design variables.

Conversely, the backward method, or adjoint method, performs the derivative calculations in reverse which means that the derivative calculations scale with the number of objective functions[125]. Fig. 4.18 shows a flow chart describing the backward method for derivative calculations where $\bar{( )}$ is the derivative with respect to the objective function, $O$. First, the derivatives of the objective function with respect to itself is set to one. Next, the derivatives of the ray intersection points with respect to the objective function can be obtained. These derivatives are then used to calculate the derivatives of the cubic spline points with respect to the objective function which then can be used to calculate the derivatives of the LSF with respect to the objective function. Finally, the change in the design parameters with respect to the objective function can be obtained. In this scheme, $\bar{\vec{p}} = \dot{O}$. The key difference is the order of operations which leads to a major reduction in calculations. Since the majority of optimization problems have far fewer objective functions than design variables the adjoint method is generally preferred. However, for this work the tangent linear method was im-

$$\begin{tikzcd}\boxed{\dot{\phi}} \arrow[r] & \boxed{\dot{\vec{x}}_{spline}} \arrow[r] & \boxed{\dot{\vec{x}}_{rays}} \arrow[r] & \boxed{\dot{O}}\end{tikzcd}$$

Figure 4.17: Forward derivative calculations

$$\begin{tikzcd}\boxed{\bar{\vec{p}}} & \boxed{\bar{\phi}} \arrow[l] & \boxed{\bar{\vec{x}}_{spline}} \arrow[l] & \boxed{\bar{\vec{x}}_{rays}} \arrow[l] & \boxed{\bar{O} = 1} \arrow[l]\end{tikzcd}$$

Figure 4.18: Backward derivative calculations

plemented resulting in longer computational time for the output of optimal results and is an area that can greatly improve the efficiency of the scheme presented here. The tangent method derivatives are discussed and presented next.

**Tangent mode**
The LSF equations can be written as,

$$\phi = \sum_i \alpha_i \psi(\xi_i) \tag{4.39a}$$

$$\psi = (1-\xi_i)^4(4\xi_i + 1), \quad \text{for } \; 0 \leq \xi_i \leq 1 \tag{4.39b}$$

$$\xi_i = \frac{r_i}{SR_i} \tag{4.39c}$$

$$r_i = [(x_i - x)^2 + (y_i - y)^2]^{\frac{1}{2}}, \tag{4.39d}$$

where Eq. 4.39b is only valid for $0 \leq \xi_i \leq 1$ and zero for all other values. The LSF in Eq. 4.39a is analytic and, therefore, the derivatives can be calculated directly. Taking the derivative of $\phi$ and applying the chain rule, the sensitivity expression becomes,

$$\begin{aligned}\dot{\phi} &= \sum_i \overset{\cdot}{\alpha_i \psi(\xi_i)} \\ &= \sum_i \dot{\alpha}_i \psi(\xi_i) + \alpha_i \overset{\cdot}{\psi(\xi_i)}\end{aligned} \tag{4.40a}$$

$$\begin{aligned}
\psi\dot{(\xi_i)} &= (1-\xi_i)^4 \overset{\cdot}{*} (4\xi_i+1) \\
&= (1 \overset{\cdot}{-} \xi_i)^4 * (4\xi_i+1) + (1-\xi_i)^4 * (4\xi_i \overset{\cdot}{+} 1) \\
&= -4(1-\xi_i)^3\dot{\xi}_i * (4\xi_i+1) + (1-\xi_i)^4 * 4\dot{\xi}_i \\
&= +4(1-\xi_i)^3[-(4\xi_i+1)+4(1-\xi_i)]\dot{\xi}_i \\
&= -20\xi(1-\xi_i)^3\dot{\xi}_i
\end{aligned} \tag{4.40b}$$

$$\begin{aligned}
\dot{\xi}_i &= \overset{\cdot}{\frac{r_i}{SR_i}} \\
&= \frac{\dot{r}_i SR_i + r_i \dot{SR}_i}{SR_i^2}
\end{aligned} \tag{4.40c}$$

$$\begin{aligned}
\dot{r}_i &= [(x-x_i)^2 \overset{\cdot}{+} (y-y_i)^2]^{\frac{1}{2}} \\
&= \frac{1}{2}[(x-x_i)^2+(y-y_i)^2]^{-\frac{1}{2}}[(x-x_i)^2 \overset{\cdot}{+} (y-y_i)^2] \\
&= \frac{(x-x_i)(\dot{x}-\dot{x}_i)}{r_i} + \frac{(y_i-y)(\dot{y}-\dot{y}_i)}{r_i}.
\end{aligned} \tag{4.40d}$$

Eqs. 4.40a–d represent the change in the LSF at a particular location $x$, $y$ and therefore the $\dot{x}$ and $\dot{y}$ terms are zero since the equations are derived on the basis that the level-set value is changing at the specified location. However, the change in the surface points is what is desired and since the surface points exist on the zero level-set curve it is assumed that they stay along the zero curve after moving. Therefore, $\dot{\phi}$ in Eq. 4.40a is set to zero and instead the derivatives $\dot{x}$ and $\dot{y}$ are solved for. Notice this leads to one equation and two unknowns, the $x$- and $y$-directions. If the points on the zero level-set curve are constrained to move along the outward normal then an equation of the motion of the points can be used and a system of equations with two equations and two unknowns is created. The normal to the curve is calculated by taking the negative of the gradient of the LSF at a particular location. The negative is used because the gradient is always in the direction of greatest ascent and since the LSF is positive inside and negative outside the topology, the gradient will always point towards the inside of the topology; its negative will point outwards. The equations can be rearranged to be written in classical matrix form as,

$$\begin{bmatrix} \partial\phi/\partial x & \partial\phi/\partial y \\ -\partial\phi/\partial y & \partial\phi/\partial x \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} -\sum_i(\dot{\alpha}_i\psi_i - \alpha_i\dot{\psi}_i) \\ 0 \end{bmatrix} \tag{4.41}$$

where $\partial\phi/\partial x$ and $\partial\phi/\partial y$ can be written as,

$$\begin{aligned} \frac{\partial\phi}{\partial x} &= \sum_i \alpha_i \frac{\partial\psi}{\partial x_i} \\ &= \sum_i \alpha_i (-20\xi_i(1-\xi_i)^3 \frac{\partial\xi}{\partial x_i} \\ &= \sum_i \alpha_i \frac{(-20\xi_i(1-\xi_i)^3)}{SR_i} \frac{\partial r}{\partial x_i} \\ &= \sum_i \alpha_i \frac{(-20\xi_i(1-\xi_i)^3)}{SR_i} \frac{(x-x_i)}{r_i} \end{aligned} \tag{4.42a}$$

and

$$\begin{aligned} \frac{\partial\phi}{\partial y} &= \sum_i \alpha_i \frac{\partial\psi}{\partial y_i} \\ &= \sum_i \alpha_i (-20\xi_i(1-\xi_i)^3 \frac{\partial\xi}{\partial y_i} \\ &= \sum_i \alpha_i \frac{(-20\xi_i(1-\xi_i)^3)}{SR_i} \frac{\partial r}{\partial y_i} \\ &= \sum_i \alpha_i \frac{(-20\xi_i(1-\xi_i)^3)}{SR_i} \frac{(y-y_i)}{r_i}, \end{aligned} \tag{4.42b}$$

respectively.

In Eqs. 4.40, $\dot{x}$ and $\dot{y}$ are the sensitivities of points along the zero level-set curve with respect to the design parameters. Thus, by assuming the points remain on the zero level-set curve and those points move normal to the current zero level-set curve, the sensitivities of the LSF with respect to the design variables can be manipulated to derive the sensitivities of the points along the zero level-set curve. Furthermore, this relationship holds for any point along the curve which means that it applies to the spline points. Therefore, $\dot{x} = \dot{x}_{spline}$ and

$\dot{y} = \dot{y}_{spline}$. Using these sensitivities, the sensitivities of the ray intersection points can be obtained by differentiating the modified Thomas algorithm. The derivative of the objective function with respect to the motion of the points along the zero level-set curve, $\dot{O}$, can then be calculated from the ray intersection point sensitivities. The objective function definition found in Eq 4.36 can be differentiated with respect to the ray intersection locations. This derivative can be combined with the derivative of the ray intersection points and the spline points, $\dot{\vec{x}}_{spline}$. Both the ray intersection sensitivities and the objective function sensitivities were derived using algorithmic differentiation[126], which is accurate to working precision. The resulting form can be found in the code listing in Appendix C. The design sensitivities calculated from this method can be used for the next step in the optimization process.

### 4.4.4 Update and convergence check procedures

Once the design sensitivities are calculated they are passed to the optimizer. As discussed previously in Sec. 4.1, a conjugate gradient optimization scheme is used for this study. The design sensitivities are used to calculate the search direction and then passed to a golden-section search algorithm to determine the best step size to take along the search direction. After the golden-section search algorithm has completed the value of $\gamma$ is multiplied by the gradient, $\vec{d}$, and the design variables are updated as,

$$\vec{p}^{k+1} = \vec{p}^{k} + \gamma \vec{d}. \tag{4.43}$$

Subsequently, the entire process is repeated from the zero-finding algorithm and the boundary points are regenerated and so on. An example of one optimization step can be seen in Fig. 4.19. The optimizer moved the RBFs and changed each coefficient to improve the objective function.

Once the boundary is updated, however, several checks need to take place to evaluate whether the optimizer should continue, reinitialize the surface grid, or exit. These criteria

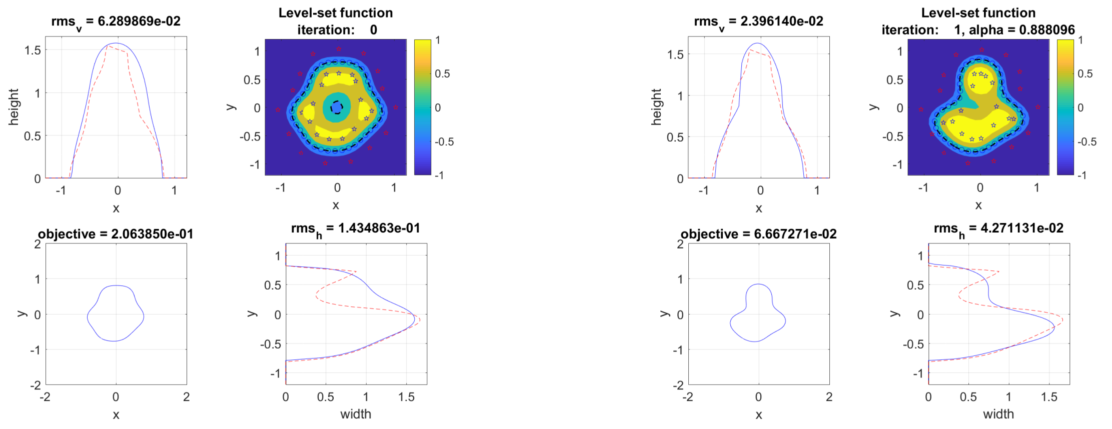


Figure 4.19: Example of an update step in the optimization scheme

include checking the

- iteration number;
- norm of the gradient;
- change in the objective;
- value of the objective function.

These are fundamental checks for an optimization algorithm and the tolerances used for each criteria will be defined in the following chapter.

If any of the above criteria besides the value of the objective are exceeded, the optimizer exits and the current optimum level-set curve is used at the new initial curve and the whole process starts over from distributing the RBFs along the curve. This process is referred to as the reinitialization of the level-set parameters. The general flow of the reinitialization can be found in Fig. 4.20. As it can be seen, the reinitialization process is similar to the initialization process seen in Fig. 4.3, however, only the last four steps are used because the boundary approximation is taken from the previous optimization run instead of being generated by the desired outcome (x-rays). Additionally, if this is not the first time reinitialization has occurred, then the previous optimum objective is compared with the current. If the current objective is greater than the previous objective, then two RBFs (one inside and one outside) are added to the total number of RBFs. This is a form of adaptive parameterization and

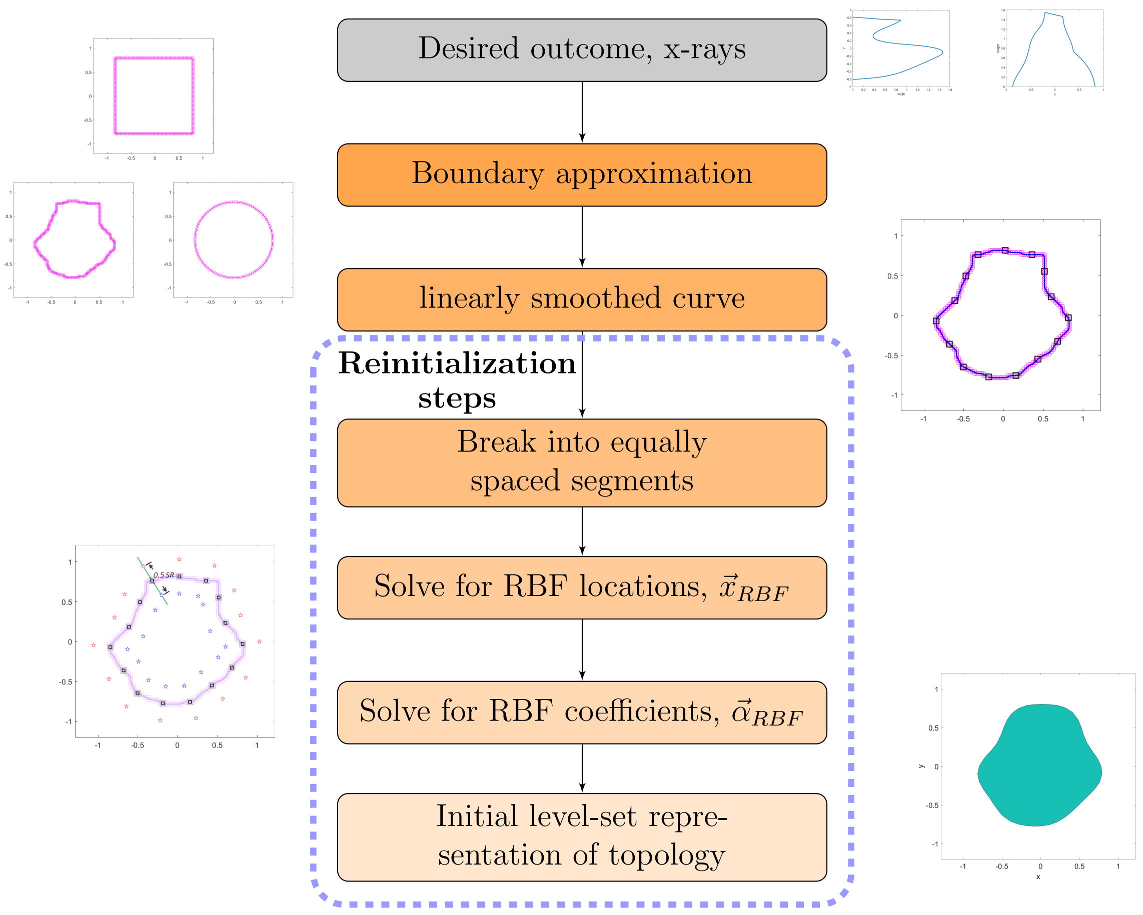

Desired outcome, x-rays
Boundary approximation
linearly smoothed curve
Reinitialization steps
Break into equally spaced segments
Solve for RBF locations, $\vec{x}_{RBF}$
Solve for RBF coefficients, $\vec{\alpha}_{RBF}$
Initial level-set representation of topology
0.5 SR

Figure 4.20: Reinitialization scheme flow chart

is implemented to improve the optimization results and will be discussed in the following chapter.

The idea behind reinitialization is to prevent premature convergence and mitigate the effects of choosing too small of a number of RBFs initially as well as the effects of a not so great initial guess. By successively incrementing the number of RBFs, the local support decreases allowing more fine tuning as the optimizer iterates closer to the desired boundary(ies). This will be discussed further in the following chapters.

The previous sections have detailed the inner workings of the optimization methodology used to perform topology optimization with the LSM. The following chapters will illustrate the results of the optimization scheme as well as discuss the main features and take-aways. Conclusions and proposed future work follow.

# Chapter 5

# Results and discussion

The following figures and tables detail the results of the optimization procedure explained in the previous chapter. This chapter is setup to show successive optimization results for single body shape optimization, multiple body shape optimization, and finally topology optimization. The distinction between the several categories is that shape optimization is the deformation of an arbitrary number of bodies throughout the optimization process but the number of bodies remains fixed, while topology optimization changes the number of bodies and deforms their boundaries simultaneously. Furthermore, an example of the effect of initial guess on the final result is shown in addition to the results before and after reinitializing the design parameters. These results are followed by a discussion of computational time scaling as a vision for future applications in CFD.

Each example will have two tables summarizing the optimization parameters and results, with accompanying figures to provide visual evidence. The optimization setup has a number of parameters that influence the initial LSF for the problem. The final quantities are presented to provide a means for comparison of the accuracy, efficiency, and robustness of the algorithm. The parameters documented in each section are the

- number of initial and final RBFs;
- support radius w.r.t. RBF spacing;

- initial and final support radii;
- the fraction of the support radius that the RBFs are offset from initial curve;
- number of rays used in objective calculations
- initial guess type
- number of points used to generate the initial boundary (before RBF distribution);
- total number of iterations;
- total number of reinitializations;
- total number of function evaluations;
- total time for the optimization;
- the initial and final objection function values.

## 5.1 Shape optimization results

The examples shown here are the results of optimizing the LSF to match the x-rays of a known number of arbitrarily shaped bodies. The test cases include a circle, an arbitrary body, a turbine blade cross-section, two side-by-side vertical ellipses, two diagonally-oriented ellipses, and ellipses of varying aspect ratios aligned vertically. The arbitrary body test case was chosen to explain the setup and initialization procedure. Furthermore, results of using different initial guesses are presented to show their effect on the efficiency, accuracy, and robustness of the optimization scheme. Similar tables and plots presented are documented for all test cases in Appendix A.

### 5.1.1 An arbitrary body

An arbitrary body was used to define the desired x-rays for this test case. The geometry used is shown in Fig. 5.1. The key feature about this test case is the various convexities and concavities that the boundary contains, making it a complicated shape that could expose issues with the optimization scheme. The setup can be seen visually in Fig. 5.2 and is discussed here. The initial guess was generated using the bounding box approach and the number of grid points along each edge was 25, see Fig. 5.2c. The initial number of RBFs was set at $N_{RBF} = 20$, 10 inside and 10 outside. Fig. 5.2d shows the distribution of the midpoints. The support radius is based on the maximum distance between neighboring RBFs, which in this case was 0.384. This distance, multiplied by the prescribed number of RBFs that the support radius should encompass ($N_{SR} = 2.5$ for this case), results in an initial support radius of $SR = 0.96$. The distance along the normal vectors that the inside and outside RBFs were placed was $F_{SR} = 0.25SR$ in both directions, separating them by a total distance of $0.5SR$. Fig. 5.2e displays the resulting distribution of RBFs, with blue indicating inside the body and red indicating outside. The initial level-set representation with the cubic spline fit can be seen in Fig. 5.2f. The number of rays used to generate the x-rays for both horizontal and vertical projections was $N_{rays} = 101$ and the initial objective function value was $9.49\times10^{-2}$. Tables 5.1 and 5.2 document each of these parameters at the beginning and end of the optimization procedure.

Fig. 5.3 shows the optimization results including objective function values and comparisons of the horizontal and vertical x-rays. Each subfigure consists of four plots – (from left to right, top to bottom) the vertical x-rays, the LSF, the cubic spline boundary representation of the zero level-set curve(s), and the horizontal x-rays. In each x-ray plot, the level-set

Table 5.1: Initial parameters for arbitrary body case

| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **30** | Number of RBFs |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **1.06** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Rectangular** | Type of initial guess used for optimization |
| $N_{grid}$ | **60** | Number of in cells in a single row of initial grid guess |
| $N_{rays}$ | **101** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | $\mathbf{1.30}\times10^{+0}$ | Initial objective function value |

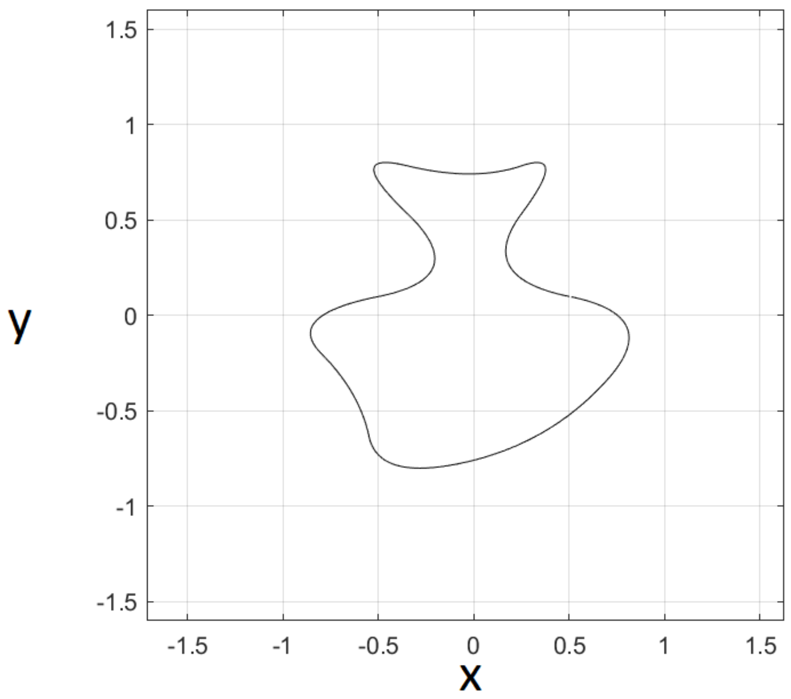
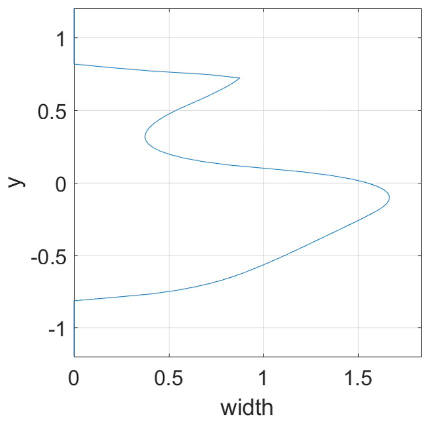
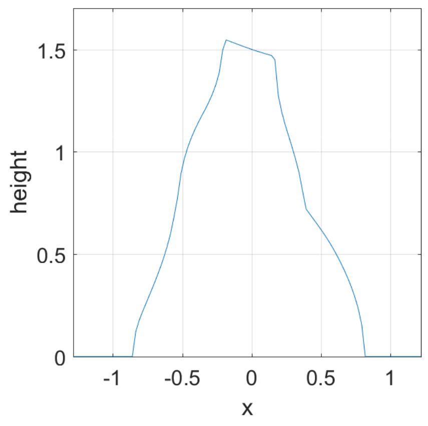


Figure 5.1: Arbitrary body used to generate the desired x-rays

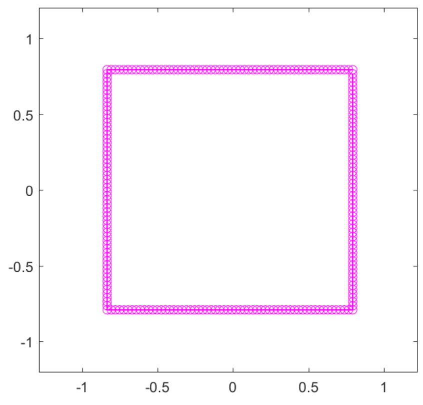
x-rays are shown as the blue solid line and the desired x-rays are shown as the red dashed line. Fig. 5.3a displays the initial level-set representation generated from the initialization procedure using a bounding box approach. Fig. 5.3b displays the final objective function after one optimization run. While this result reduces the objective value by two orders of magnitude to $1.04{\times}10^{-2}$, it can be seen in Fig. 5.3b that various features in both desired x-rays are missed by the current level-set representation. Specifically, the flat top in the desired vertical x-ray is pointed in the level-set x-ray and the pointed portion of the desired horizontal x-ray is rounded in the level-set x-ray. The reason for this inaccurate final result is the fact that the motion and change in the RBF locations and coefficients has driven the optimizer into a local minimum where no change in any direction of the gradient will make an improvement in the objective function.In fact, the optimization scheme exits once a change in the objective function is less than $1.0{\times}10^{-10}$. Additionally, it can be observed that the RBFs are in very different final locations which indicates that the initial positions were not very close to optimal.



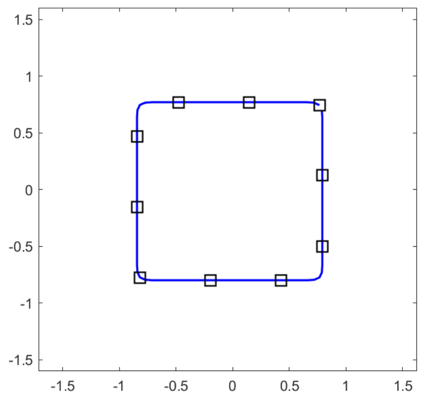
(a) Desired horizontal x-rays



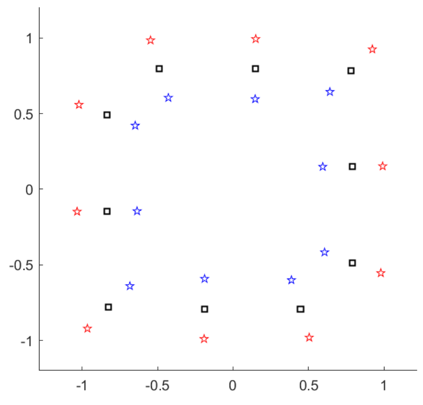
(b) Desired vertical x-rays



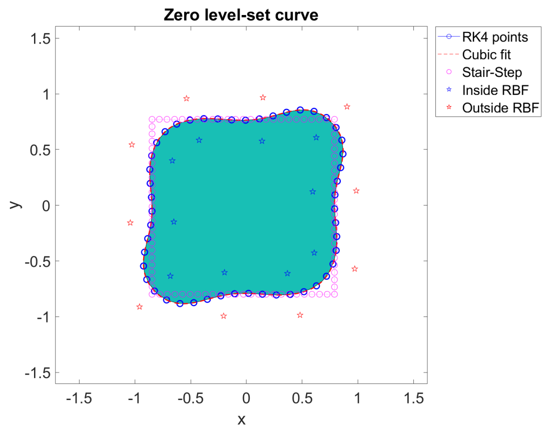
(c) Initial boundary representation



(d) Linearly smoothed curve with RBF midpoints

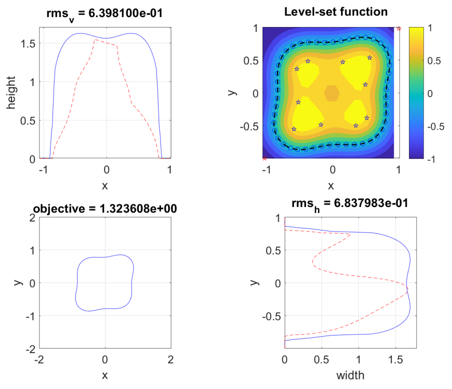


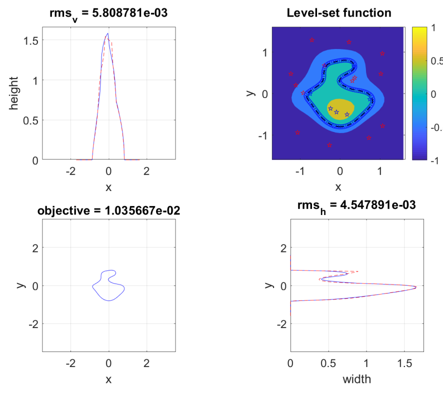
(e) RBF distribution. red - outside, blue - inside



(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure 5.2: Setup for arbitrary body case


(a) Initial objective function evaluation


(b) First optimum objective function evaluation

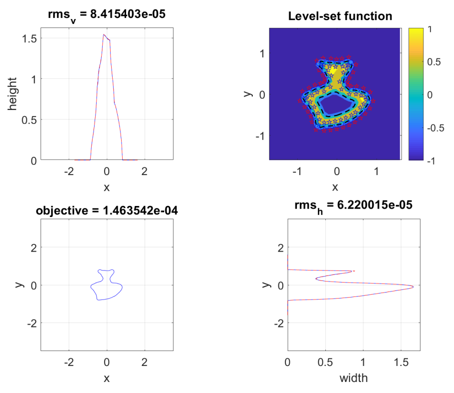

(c) Final optimum objective function evaluation

Figure 5.3: Optimization test case: arbitrary body

As mentioned in the methodology section, to improve this result a reinitialization step is employed in the optimizer that uses the optimal curve from the previous optimization run and redistributes the RBFs along this curve and re-solves for their coefficients using the same approach as before. The optimizer is then restarted and the process repeats. If the current optimal value is greater than a previous run, then the number of RBFs is increased by 2 (one inside and one outside). The reinitializations continue until the objective function falls below $1.0\times10^{-6}$ or the number of reinitializations reaches 99. Moreover, the total number of iterations per optimization run is 9999, which gives the optimizer 989,901 attempts to optimize the design; however, none of the examples shown here have reached this maximum number of iterations. In this particular case, the total number of iterations was $N_{iter} = 2042$ with $N_{reinit} = 76$ reinitializations, $N_{eval} = 528462$ function evaluations, and ran for roughly $T_{opt} = 64.5$ hours. The final result is shown in Fig. 5.3c with an optimum objective value of

$2.88{\times}10^{-4}$. The final parameters are summarized in Table 5.2.

Reinitializing the design parameters has the effect of improving the optimization results (in accuracy and robustness) while increasing the necessary computational resources. For this test case the result improved by nearly two orders of magnitude from the first optimum value of $1.21{\times}10^{-2}$ to the final optimum value of $2.88{\times}10^{-4}$. The number of RBFs influenced by each RBF ($N_{SR}$) was held constant throughout the optimization. This allows the local support radius, $SR$, to change each reinitialization because the maximum distance between neighboring RBFs may have changed. Table 5.2 shows that the number of RBFs has increased from $N_{RBF} = 30$ to $N_{RBF} = 150$, which means that the number of RBFs was increased 60 times. In other words, there were 60 optimization runs where the previous optimum value was less than the current. This implies that the initial guessed number of RBFs was insufficient for accurately matching the desired x-rays. Reinitialization enables the optimization scheme to adapt to the current boundary in several ways; first, by redistributing the RBFs about a more accurate curve, second, by evaluating whether the optimum objective has improved, and third, by choosing to increase the number of design parameters to better match the x-rays. Using these results as evidence of the benefits of reinitialization, the same process is used for each subsequent optimization test case. Next is a discussion about the curvature of the boundary output by the optimization scheme followed by an investigation into various initial guesses.

Table 5.2: Final parameters for Arbitrary Body case

| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **150** | Number of RBFs |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.20** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **2042** | Total number of iterations |
| $N_{reinit}$ | **76** | Total number of reinitializations |
| $N_{eval}$ | **528462** | Total number of function evaluations |
| $T_{opt}$ | **64.5** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | $\mathbf{2.88}{\times}10^{-4}$ | Optimum value for test case |

### 5.1.2 Curvature continuous boundary representation

With the optimization complete, it can also be shown that the output representation of the design is curvature continuous. The final boundary and a plot of its curvature can be found in Fig. 5.4. In Fig. 5.4a the red squares represent the cubic spline segment end points and the blue curve is the finely sampled curvature continuous representation of the arbitrary body. Evidence that the curve is continuous in the 2nd derivative is provided in Fig. 5.4b, where the red squares are the same cubic spline segment end points and the blue curve represents the curvature versus the parametric variable $t$. Since the boundary is a parametric curve, calculating the curvature is trivial because the first and second derivatives are readily available for $x(t)$ and $y(t)$. The equation for curvature using parametric coordinates is given by Eq. 5.1 as,

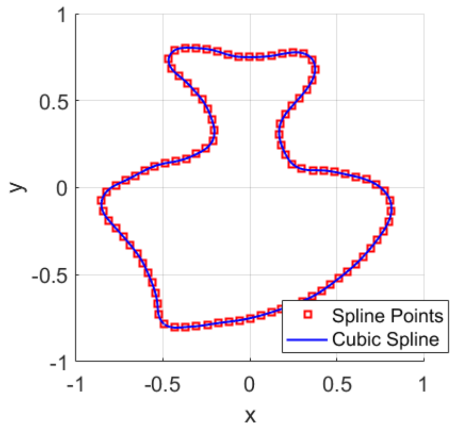
$$\kappa = \frac{x'y'' - y'x''}{(x'^2 + y'^2)^{\frac{3}{2}}} \tag{5.1}$$

where the primes denote derivatives with respect to parametric variable $t$. The fact that the curve in Fig. 5.4b is point continuous illustrates that the curve representing the arbitrary body is curvature continuous. Thus, the optimization method has produced a $C^2$ design.

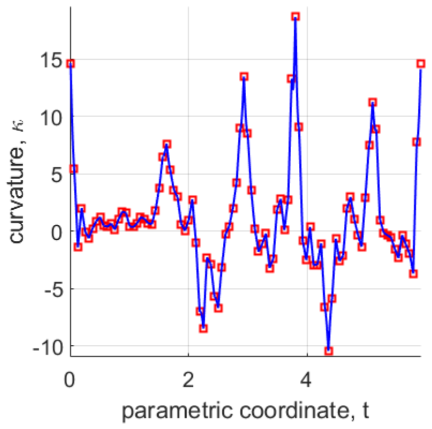
Here an initial guess was used by generating a bounding box using the desired x-ray information provided. As seen in Fig. 5.3a, this initial guess was poor and may have limited the success or efficiency of the optimization scheme. In the following section three different initial guesses are used to better understand how an initial guess may affect the outcome of the optimization scheme: a bounding box (as seen here), a bounding ellipse, and a stair-stepped representation.

### 5.1.3 The effect of different initial guesses

In Ch. 2 it was discussed that the initial guess for topology optimization problems can have an effect on the optimization results. Here the effects are investigated using three different initial guesses and studying how the optimization scheme changes with respect to the



(a) Final boundary representation



(b) Final curvature of the design

Figure 5.4: Optimization test case: arbitrary body

different starting designs. In particular, the final number of RBFs, final support radius, total number of iterations, total number of reinitializations, total number of function evaluations, total optimization time, and the accuracy of the optimization scheme are documented and compared. The initial parameters for each case are shown in Table 5.3 and the results can be found in in Table 5.4. The initial representation of the boundary generated from the x-rays is shown in Fig. 5.5. The initial and final level-set representations compared to the desired x-rays are shown in Figs. 5.6.

The initial parameters are similar in all aspects except support radius, where the rectangular case has a larger support radius by roughly a factor of 1.20 compared to the other two cases. While the initial setups may be similar, the initial objective value for each case is different, with the worst being the rectangular case and the best being the stair-stepped. This is expected because as the initial changes from rectangular to elliptical to stair-stepped the boundary of the geometry that generated the desired x-rays is more closely recovered. Therefore, the resulting level-set and subsequent cubic spline representation more closely matches the desired geometry and its x-rays. Visual evidence of this is seen in Fig. 5.6.

The optimization results in Table 5.4 show that the rectangular case requires many more reinitializations than the other two and this results in a much larger number of RBFs

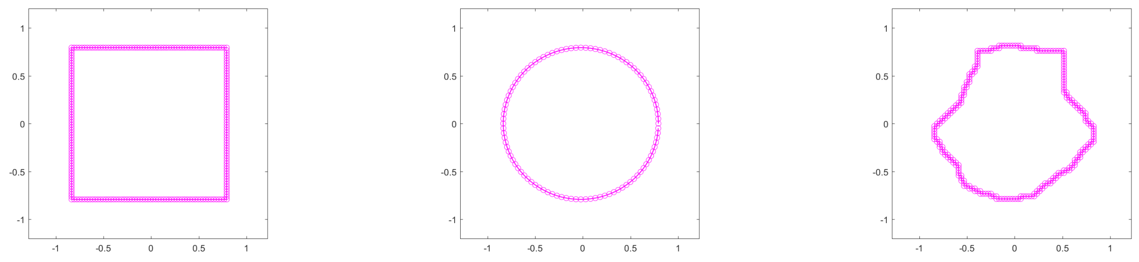

(a) Bounding box guess (b) Bounding ellipse guess (c) Stair-stepped guess

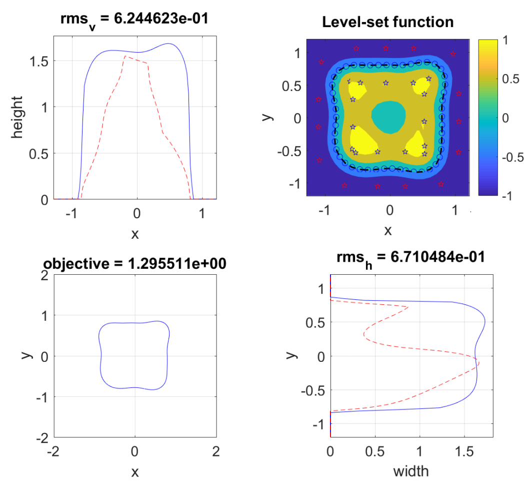
Figure 5.5: Example of initial guesses for RBF midpoint locations for the arbitrary body case

at the end of the optimization. The large number of design parameters contributes to the computational resources necessary to perform the optimization. Even though the rectangular case evaluates the objective function fewer times, the total time for the optimization was longer because more design variables were needed to be used in derivative and objective evaluations. According to Table 5.4, a lower number of RBFs throughout the optimization procedure can result in a faster optimization run. Furthermore, the initial guess has an effect on the final result. The difference between objective function values is at most 3 times worse than the best result. This is promising and may indicate that while a good initial guess will save computational resources and improve the efficiency of the algorithm, the result will be similar with a worse guess. More research is necessary to truly make the claim that the initial guess does not effect the final result and is beyond the scope of this work.

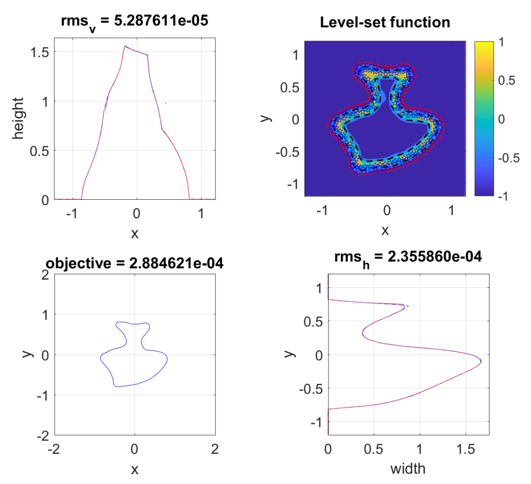
The arbitrary body served as an example for the complete optimization procedure and analysis of how reinitialization and initial guesses affect the overall scheme. The following

Table 5.3: Initial parameters for arbitrary body cases with different initial guesses

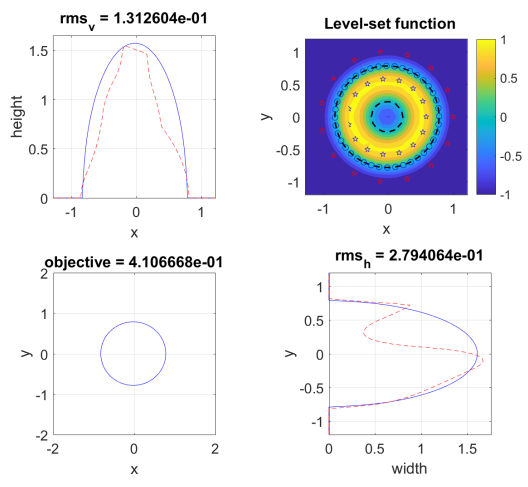
| *Parameter* | **Rectangular** | **Elliptical** | **Stair-stepped** |
|---|---|---|---|
| $N_{RBF}$ | **30** | **30** | **30** |
| $N_{SR}$ | **2.5** | **2.5** | **2.5** |
| $SR$ | **1.06** | **0.84** | **0.86** |
| $F_{SR}$ | **0.25** | **0.25** | **0.25** |
| $N_{grid}$ | **60** | **60** | **60** |
| $O(\vec{p}_0)$ | $\mathbf{1.30}\times 10^{+0}$ | $\mathbf{4.11}\times 10^{-1}$ | $\mathbf{2.06}\times 10^{-1}$ |



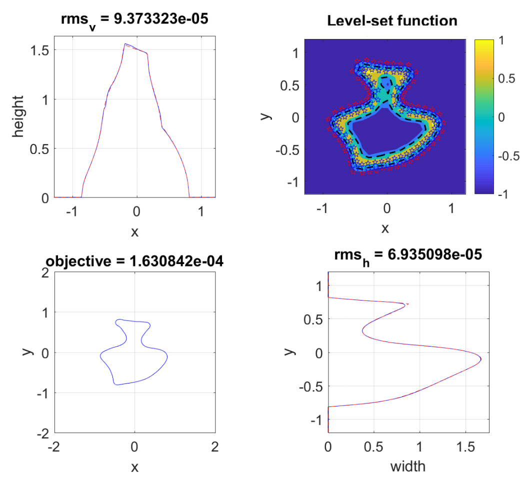
(a) Rectangular initial objective value



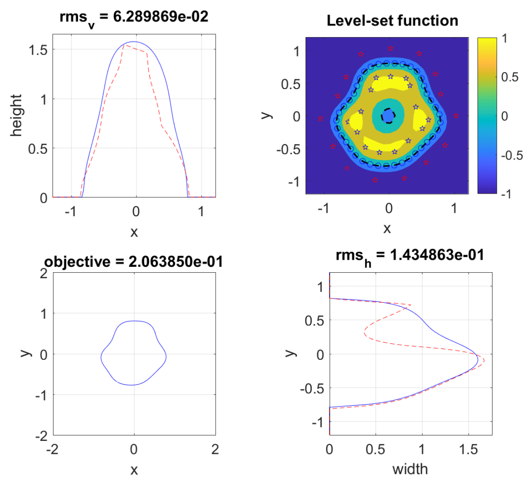
(b) Rectangular final objective value



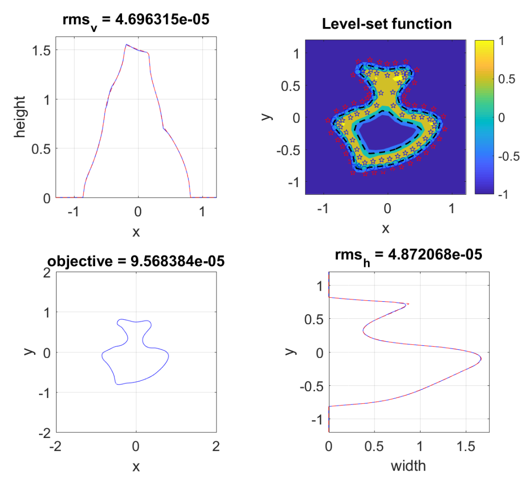
(c) Elliptical initial objective value



(d) Elliptical final objective value



(e) Stair-stepped initial objective value



(f) Stair-stepped final objective value

Figure 5.6: Optimization results for a various initial guesses

Table 5.4: Final parameters for Arbitrary Body cases with different initial guesses

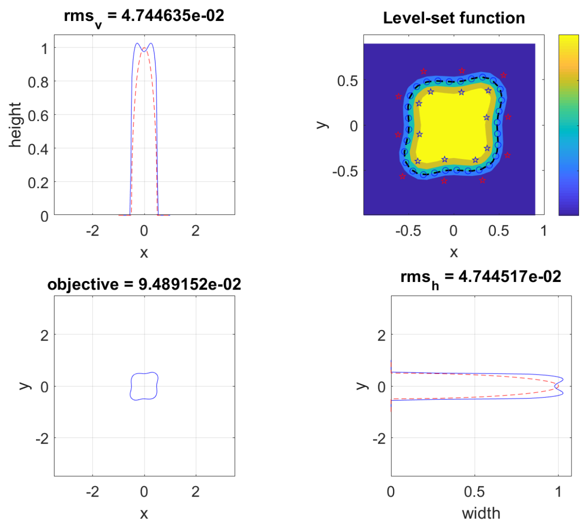
| *Parameter* | **Rectangular** | **Elliptical** | **Stair-Stepped** |
|---|---|---|---|
| $N_{RBF}$ | **150** | **112** | **86** |
| $N_{SR}$ | **2.5** | **2.5** | **2.5** |
| $SR$ | **0.20** | **0.27** | **0.35** |
| $N_{iter}$ | **2042** | **3054** | **2489** |
| $N_{reinit}$ | **76** | **56** | **35** |
| $N_{eval}$ | **528462** | **681570** | **619686** |
| $T_{opt}(hrs)$ | **64.5** | **60.9** | **56.1** |
| $O(\vec{p}_{opt})$ | $\mathbf{2.88}\times10^{-4}$ | $\mathbf{1.63}\times10^{-4}$ | $\mathbf{9.57}\times10^{-5}$ |

section shows more examples of the optimization algorithm applied to a variety of designs with single or multiple bodies.

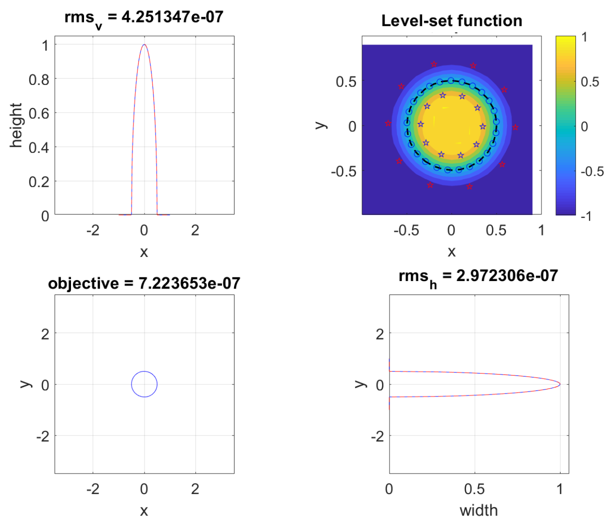
### 5.1.4 Optimization results for all test cases

In this section the results for matching the x-rays generated by a circle, turbine blade, two vertical ellipses, and three vertically aligned ellipses with varying aspect ratios are presented. The initial guesses for each case was generated with knowledge of the number of bodies in the design domain, so each case has the correct number of bodies at the start of the optimization. This section illustrates the ability of the optimization scheme to perform shape optimization on single and multiple body design problems. The setup and initialization figures and tables similar to those used to explain the arbitrary body case in the previous section can be found in Appendix A.

Figs. 5.7 - 5.10 display the initial and optimized results for the various test cases. A summary of the design parameters and the optimization analysis for each case can be found in Table 5.5. The circle case performs the best of the four cases, being able to reduce the objective function to less than $1.0e^{-6}$. This result was expected because the RBF footprint is a circle and therefore this case is well-suited for this level-set parameterization. The initial guess for the circle case was rectangular, which did not hinder the accuracy or efficiency of the optimization procedure, seeing as the objective has been sufficiently minimized and the



(a) Initial objective function evaluation



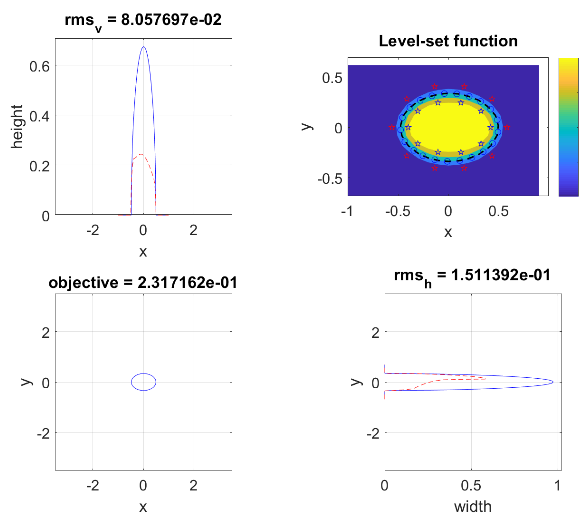
(b) Final objective function evaluation

Figure 5.7: Optimization results for a circle with rectangular initial guess

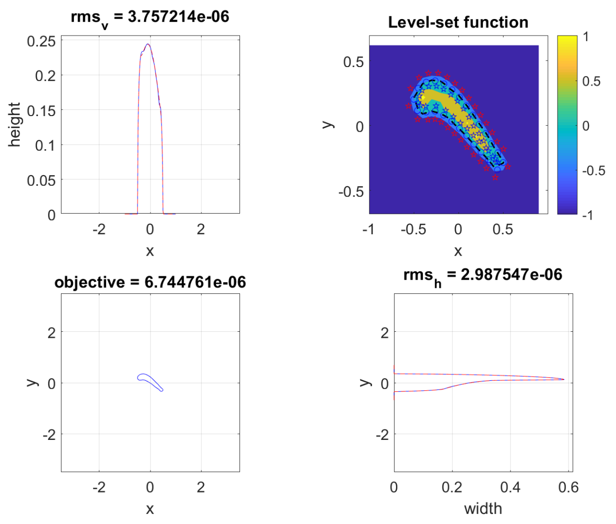
number of reinitializations is minimal. The next example with a single body is the turbine blade, which is a more complicated shape than the circle, having concavities and convexities, while also being somewhat slender (having a large aspect ratio).

The turbine blade case is an example of how well the optimization scheme can handle aerodynamic bodies. The LSF was initialized using an elliptical guess because of the desired x-rays' more complex shape. The key take-away from this test case is the ability of the optimization algorithm to accurately match the x-rays of an aerodynamic body. In view of using this tool for aerodynamic design optimization, producing the results shown in Fig. 5.8 are promising and present opportunities for future work. In particular, being able to optimize a collection of these blade-like designs may have implications in the design of turning vanes in a flow bend. The following examples aid in the understanding of how this tool works with multiple bodies.

Multiple bodies can be handled by the optimization algorithm, as shown by both Figs. 5.9 and 5.10. In both cases the objective function is of the order of magnitude $10^{-5}$ starting from values close to $10^{-1}$. Each case was initialized differently, however, with the vertical ellipse case using a rectangular initial guess and the three ellipse case using a stair-stepped

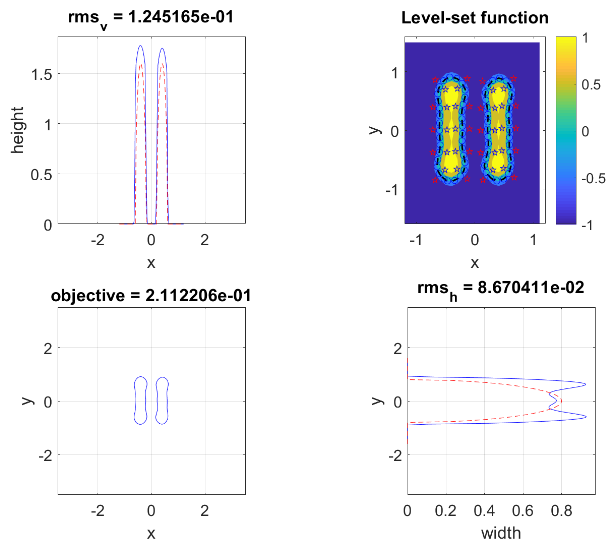


(a) Initial objective function evaluation

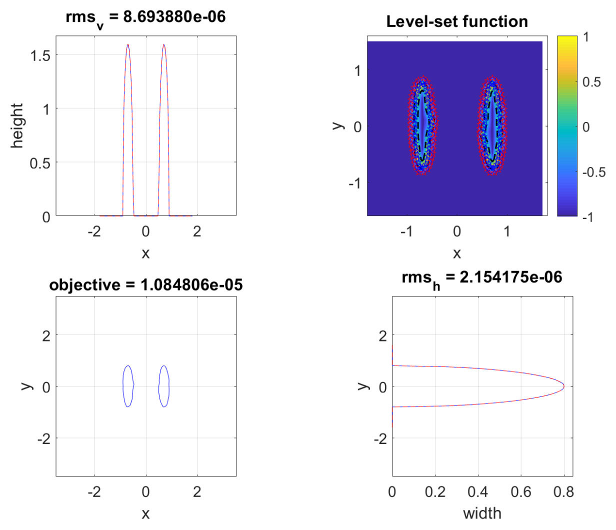


(b) Final objective function evaluation

Figure 5.8: Optimization results for a turbine blade with elliptical initial guess



(a) Initial objective function evaluation

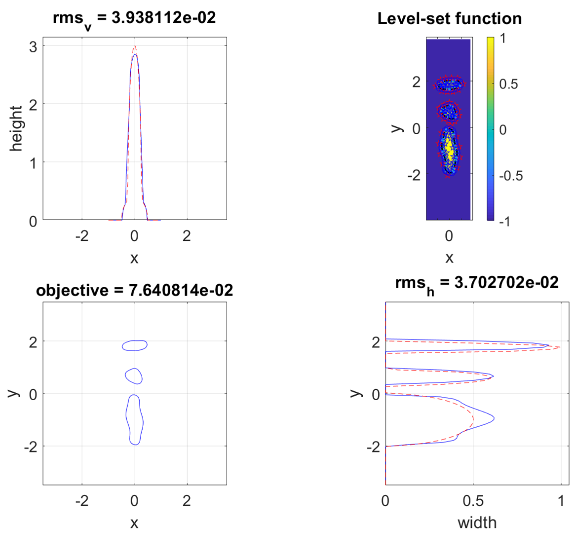


(b) Final objective function evaluation

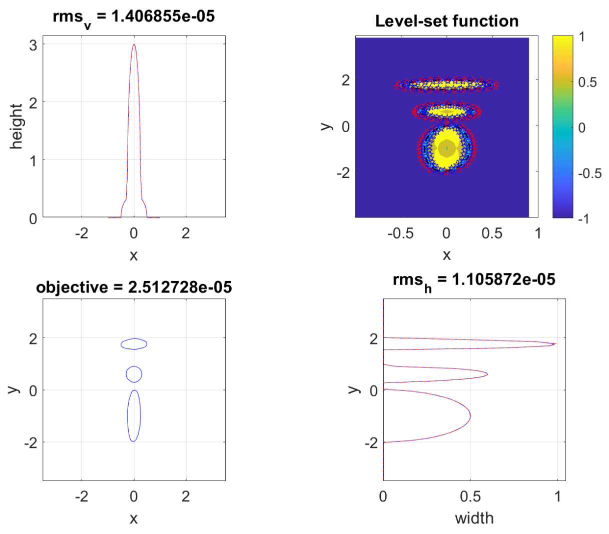
Figure 5.9: Optimization results for two vertical ellipses with rectangular initial guess

guess. However, both cases perform similarly with respect to accuracy of the final result. On the other hand, Table 5.5 shows that the two ellipse case out-performs the three ellipse case, even though the number of RBFs at the end of the optimization is greater.

The three ellipse case required more iterations and function evaluations, implying that the added complexity of three bodies versus two bodies results in larger computational cost.



(a) Initial objective function evaluation



(b) Final objective function evaluation

Figure 5.10: Optimization results for vertically aligned ellipses with varying aspect ratios with stair-stepped initial guess

A similar result is seen when comparing the single body cases to the two body ellipse case, where the total time for the optimizer to complete is at least one-fourth the time of the multiple body case. The relationship to design complexity and computational cost is also illustrated by the increasing number of function evaluations from case to case.

These test cases, while some were complicated, were made easier by the knowledge of how many bodies were in the design domain. The following section and examples examine what happens if the number of bodies is unknown or the initial guess incorrectly generates the number of bodies in the domain. As such, the examples will show topology optimization

Table 5.5: Initial/final parameters for various shape optimization test cases

| *Parameter* | **Circle** | **Turbine Blade** | **Vert. Ellipses** | **Three Ellipses** |
|---|---|---|---|---|
| $N_{RBF}$ | **20/20** | **20/56** | **40/196** | **60/156** |
| $N_{SR}$ | **2.5** | **2.5** | **2.5** | **2.5** |
| $SR$ | **0.96/0.77** | **0.65/0.23** | **0.97/0.175** | **1.06/0.41** |
| $N_{iter}$ | **2525** | **304** | **667** | **2108** |
| $N_{reinit}$ | **1** | **20** | **57** | **30** |
| $N_{eval}$ | **4177** | **51807** | **244584** | **909515** |
| $T_{opt}(hrs)$ | **0.13** | **4.1** | **29.22** | **51.83** |

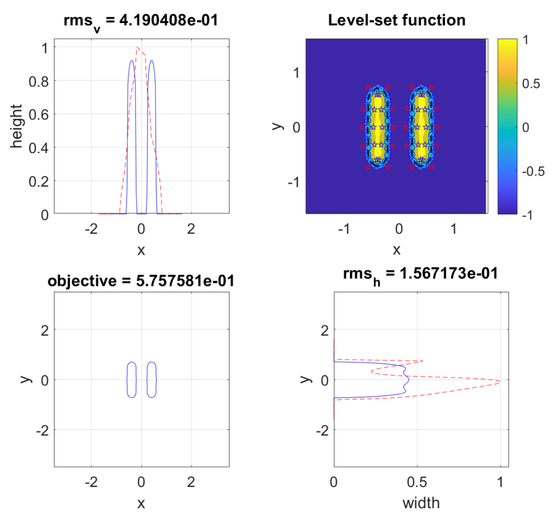
where the initial number of bodies is different than the final through topological changes including merging of bodies and bodies tearing apart.

## 5.2 Topology optimization results

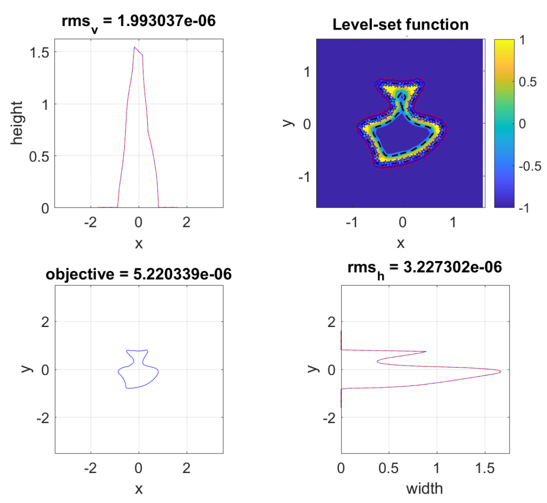
Topology optimization, by definition, is the process by which the topology of a design can change throughout the optimization procedure to obtain the best design possible for a given objective function. In the previous examples the topologies (number of bodies) were the same at the beginning and end of the optimization scheme, so in essence, these were shape optimization problems. However, if the initial guess cannot parse the various bodies apart or generates more bodies than necessary, then the optimization scheme can start with a different number of bodies than it ends with. Herein, topological changes such as merging bodies and tearing bodies apart are demonstrated by the optimization scheme. Merging two bodies into an arbitrary body is shown followed by three examples of topological tearing which demonstrate not only the tearing of bodies but also the disappearance of extraneous bodies. The following Figs. 5.11 – 5.16 detail the results of the various optimization test cases. Additionally, the overall results of the optimization algorithm are summarized in Table 5.6. An example of each type of topological change (merging and tearing) is also shown in the following subsections and supplemental results can be found in appendix B.

### 5.2.1 Merging bodies together

For the topologically merging test case, two ellipses are initialized using the rectangular initial guess generator. The goal of this optimization run was to match the x-rays of an arbitrary body given the x-rays of a topologically different design, hence the two ellipses. The initial guess and final result are shown in Fig. 5.11. In this case, the optimizer reduces the objective function by five orders of magnitude and represents the arbitrary body well with $RMS$ values for both the vertical and horizontal x-rays less than $5.0\times10^{-6}$. The number



(a) Initial objective function evaluation



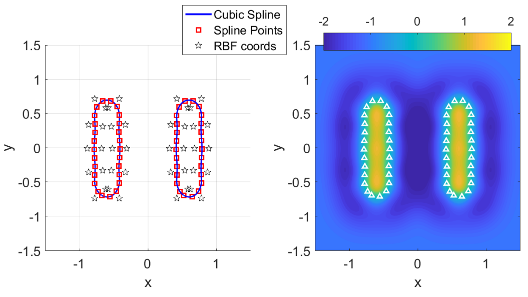
(b) Final objective function evaluation

Figure 5.11: Optimization results from two bodies to one with rectangular initial guess

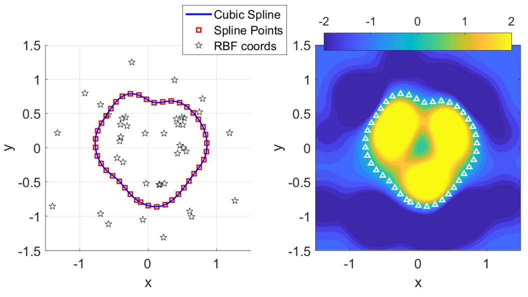
of RBFs began at 40 and once the merging occurred reduced to 20 because there was only one boundary. The reason for this reduction is that optimizer prescribes the number of RBFs to distribute per boundary. From 20 RBFs, the optimizer iteratively reinitialized and incremented the number to 102 where the final solution was found.

While Fig. 5.11 shows the initial and final result, it can be informative to see what happens in between to understand how and when the merging takes place. Fig. 5.12 shows that the merging occurs within the first iteration during the golden-section search algorithm. As can be seen in this figure, the RBFs have moved both toward and away from each other but the resulting arrangement is somewhat elliptical. The movement and change in coefficient causes the LSF to grow in the center of the design domain resulting in the merging phenomena.

The optimization scheme has been shown to handle the merging of boundaries. This type of topological change is the most common in current work, as discussed in Ch. 2, and is easier to induce. However, topological tearing, often referred to as hole generation, is more challenging. The next subsection shows the results of initializing the optimization scheme with too few bodies with the hope that tearing will occur. The goal is to better understand when, how, and why this type of topological event occurs to help apply the method to other design problems.



(a) Initial



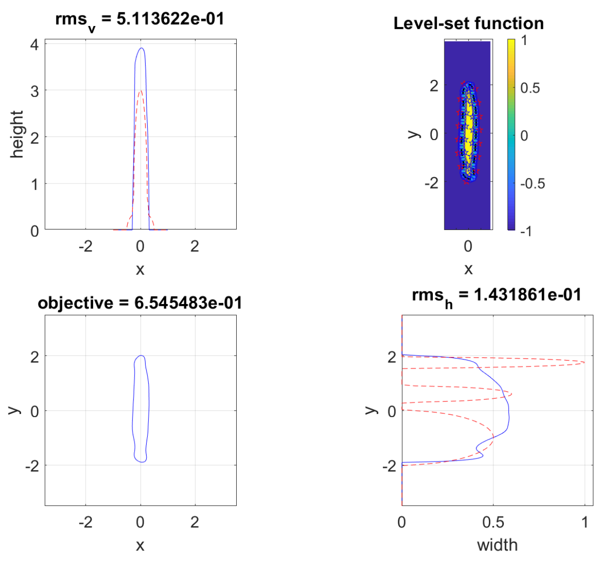
(b) Iteration 1

Figure 5.12: Merging example: cubic fit and RBF locations (left) and the LSF (right) are shown in each subfigure.

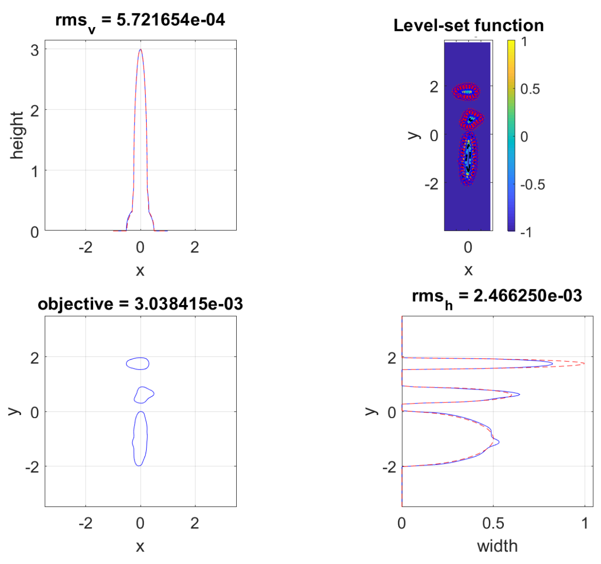
### 5.2.2 Tearing bodies apart

An example of three ellipses being initialized by one ellipse is used to illustrate topological tearing. Additionally, this example shows merging as well as the disappearance of bodies. The initial and final results are shown in Fig. 5.13 and several iterations are shown in Fig. 5.14. The case was initialized using the stair-stepped representation of a bounding ellipse that had a horizontal axis of 0.5 and a vertical axis of 4. The result of the optimization scheme suggests that, for this objective function, the optimizer is susceptible to local minima and that the change in one x-ray out weighs the change in the other. In this case, the vertical x-ray is dominating the optimization and a small change in the horizontal can produce a large change in the vertical. The result is seen in Fig. 5.13b where the bottom horizontal x-ray for the level-set representation has a defect or bump. At this location, moving that curve in would result in a larger negative change in the objective because the vertical x-ray would be perturbed. Clearly, this is an indication that the optimization method has room for improvement, which will be discussed further in the next chapter. The optimizer succeeded in morphing the LSF and tearing the design into three bodies from one, even though it struggled to minimize the objective function.

This test case finds the appropriate number of bodies within the first 5 iterations of the optimization. The progression from initial guess to final number of bodies is seen in



(a) Initial objective function evaluation

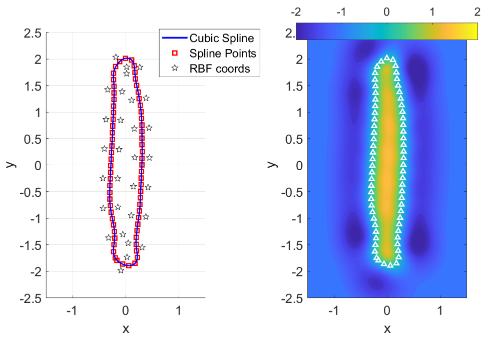


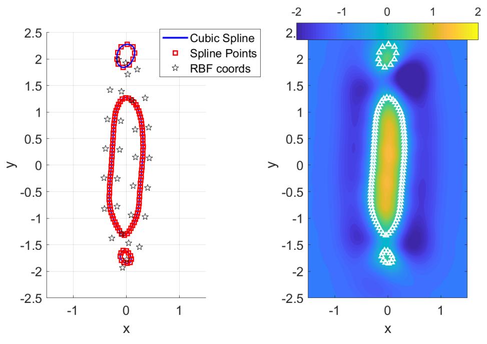
(b) Final objective function evaluation

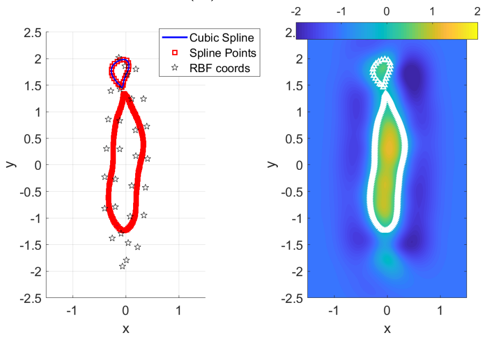
Figure 5.13: Optimization results from one body to three with stair-stepped initial guess

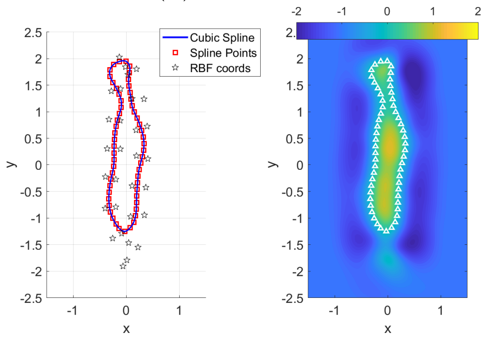
Fig. 5.14. Notice that the RBFs remain seemingly stationary throughout these few iterations, indicating that the driving force for the topology and shape changes is mainly due to changing the coefficient of the RBFs. Recall that the merging case displayed more apparent motion of the RBFs than seen here. This information is insufficient to make any claims regarding whether the motion of the RBFs or the change in their coefficients is more influential in generating topological changes. However, it is clear that both can contribute to the tearing and merging of boundaries. Additionally, this test case illustrates that the topology can change multiple times throughout a single optimization run. Appendix B shows examples where both the coefficient and motion of the RBFs contribute to the tearing of a design.

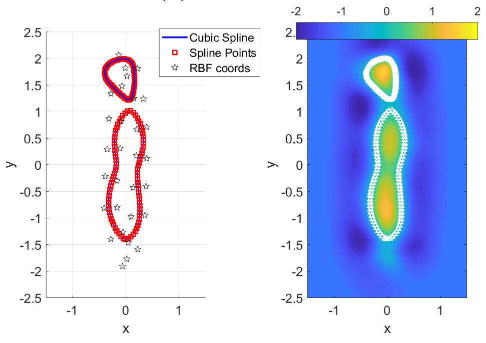
As can be seen in Fig. 5.14, this example shows that the optimization scheme can handle a variety of topological changes throughout the optimization process. Initially, the design splits into three bodies (Fig. 5.14b), then absorbs the lower body, leaving two (Fig. 5.14c). The two remaining bodies merge (Fig. 5.14d) and then tear, again (Fig. 5.14e). Finally, the third body pinches off from the larger one (Fig. 5.14g) and the desired topology is obtained. These changes are able to be captured because a scan of the domain for level-set crossings (described in Sec. 4.4.1) is performed multiple times during each execution of the golden-

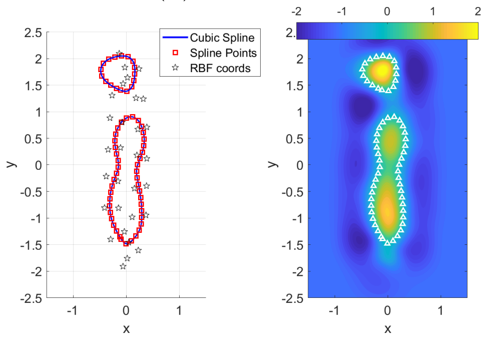

(a) Initial

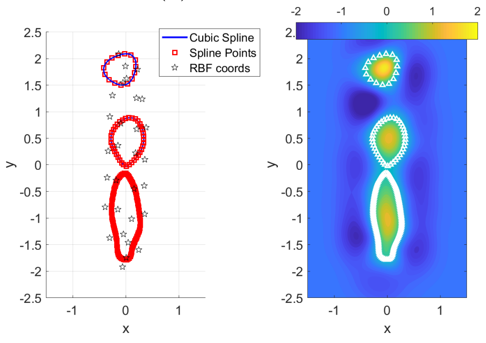

(b) Iteration 1


(c) Iteration 2


(d) Iteration 3


(e) Iteration 5


(f) Iteration 6


(g) Iteration 7

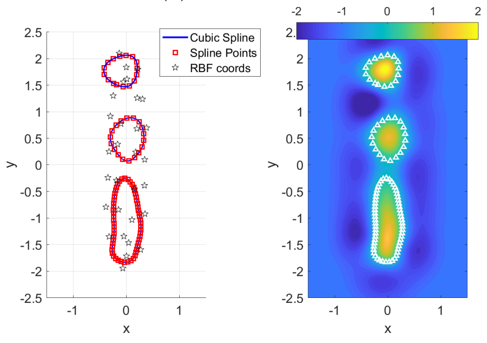
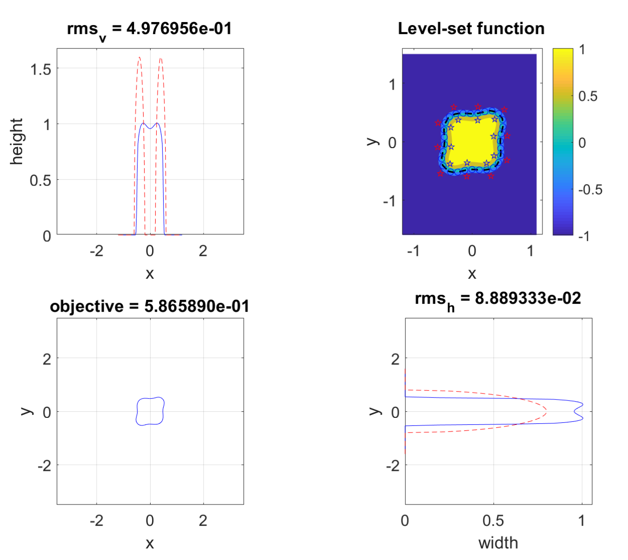
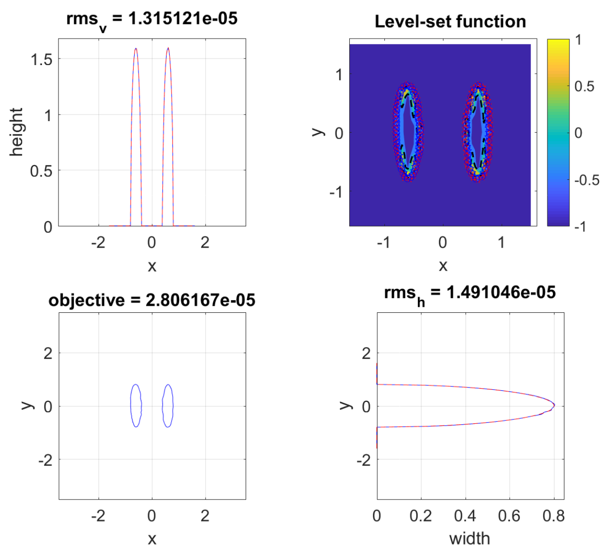

(h) Iteration 8

Figure 5.14: Tearing example: cubic fit and RBF locations (left) and the LSF (right) are shown in each subfigure.



(a) Initial objective function evaluation

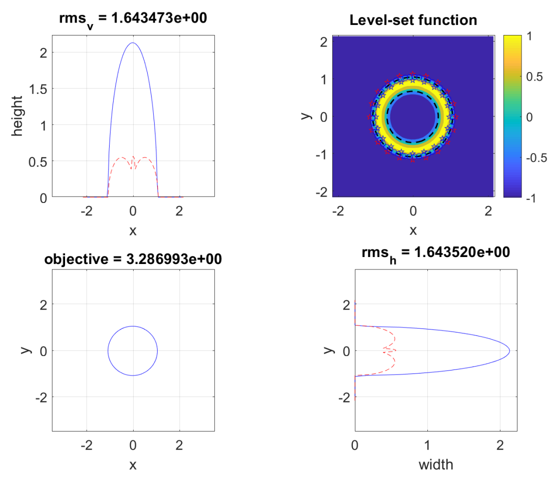


(b) Final objective function evaluation

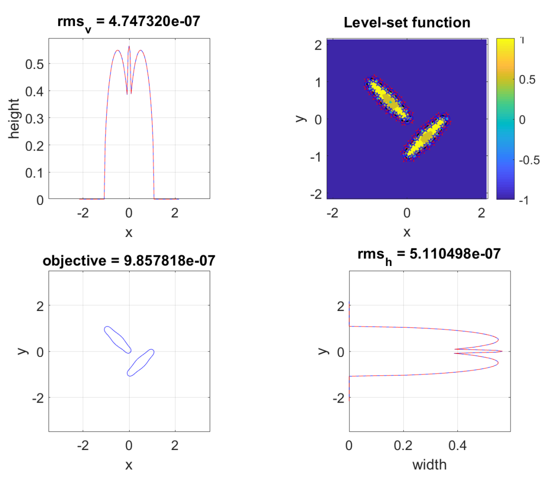
Figure 5.15: Optimization results from one body to two with rectangular initial guess

section search algorithm. By performing a scan each time the design variables are changed, the optimizer mitigates the effects of point-tracking that can cause curve tangling or point overlap and allows the algorithm to take advantage of the implicit nature of the LSF. Several other examples of topological tearing can be found in Figs. 5.15 and 5.16. While Fig. 5.15 shows basic tearing, Figs. 5.16 shows tearing without information that indicates the design should tear.

In the example of diagonal ellipses, the x-rays generated by the desired configuration do not have a clear break in them. Inspection of the previous examples shows that the desired x-rays for each case inform the topology or number of bodies in the result. The gap in the x-rays, where the desired x-ray is equal to zero, indicates a gap in the design. So, for the three ellipse case and the two ellipse case (Figs. 5.13 and Figs. 5.15) the optimizer had information that was beneficial to create a gap in particular places. For this example, however, there are no gaps because the geometries are overlapping. Instead, there exist stark discontinuities that indicate an abrupt change in the width or height of the design. Similar discontinuities can be seen in the arbitrary body x-rays, Figs. 5.2a and 5.2b, so they do not solely imply a topological change is required to satisfy the x-rays. A topological change may occur if it is



(a) Initial objective function evaluation



(b) Final objective function evaluation

Figure 5.16: Optimization results for two diagonal ellipses with elliptical initial guess

more beneficial to satisfy the discontinuities than the overall x-ray. Overlapping geometries pose a problem for this type of objective function because the orientation of the geometries is ambiguous to the optimizer, so the result can have one body or multiple.

The optimization scheme has been proven to handle and generate topological changes such as merging and tearing of boundaries. A summary of the optimization results is shown in Table 5.6. The merging case ran for the longest time, but also had the second smallest objective value, while the diagonal ellipse case ran for the shortest time and had the best objective value. Furthermore, as the number of total iterations increase, the total time for the optimization scheme increases. Considering these results and the results for shape optimization using this technique, a brief discussion of its scalability is warranted.

Table 5.6: Initial/final parameters for various topology optimization test cases

| *Parameter* | **Merge** | **Tear-1** | **Tear-2** | **Tear-3** |
|---|---|---|---|---|
| $N_{RBF}$ | **40/102** | **20/176** | **30/168** | **40/80** |
| $N_{SR}$ | **2.5** | **2.0** | **2.0** | **2.0** |
| $SR$ | **0.84/0.29** | **0.59/0.16** | **1.10/0.31** | **0.66/0.34** |
| $N_{iter}$ | **1830** | **876** | **286** | **196** |
| $N_{reinit}$ | **58** | **50** | **25** | **6** |
| $N_{eval}$ | **460615** | **283535** | **113302** | **55064** |
| $T_{opt}(hrs)$ | **75.69** | **24.98** | **11.81** | **2.94** |

### 5.2.3 Brief discussion on computational time

The aim of this work is to extend the developed technique to CFD applications which require a large amount of computational resources. The time to run CFD simulations can vary based on applications. For this discussion, it is assumed the simulations required to analyze the designs produced by the LSM require time on the order of hours versus days to complete. Complete in this sense means one function evaluation. Consider the first example of the arbitrary body. The total time for the optimization to complete was 64.5 hours with 528,462 function evaluations. If each function evaluation takes $N$ number of hours to complete, this problem is prohibitively expensive. However, there are several improvements that can be made to the technique that would increase the computational speed and reduce the time for each iteration.

The algorithm was implemented using MATLAB with considerable effort made throughout its development to limit vectorized operations so the code could be easily converted to C. MATLAB is considerably slower than C and with proper conversion, the total time can be reduced by a factor of at least 10. Conversion to C would have the effect of improving the speed at which functions were called and calculations were executed. Additional improvements in speed could be made by implementing the adjoint method for derivative calculations. Currently, the tangent linear method is used which requires an additional function evaluation for each design variable. Alternatively, the adjoint method would only require one additional function evaluation since there is only one objective function. This would greatly increase the speed of the optimization procedure. Furthermore, changing the optimization technique from conjugate-gradient to the Levenberg-Marquardt[127,128] algorithm could potentially improve the computational efficiency of this technique by reducing the number of times the function needs to be evaluated. The reduction would be the result of removing the need for a one-dimensional search algorithm and replacing it with the trust-region technique[129] used in the Levenberg-Marquardt algorithm. These improvements would assist in extending this topology optimization technique to CFD applications.

The optimization scheme has been described and demonstrated to show that it is capable of shape and topology optimization. The next chapter contains conclusions that can be drawn from these results, as well as direction for future research involving the ideas shared here.

# Chapter 6

# Conclusions and future research suggestions

Topology optimization is a powerful tool that can improve designs in many engineering fields. It has been widely accepted in the structural dynamics community and the fluid dynamics community for low to moderate Reynolds number flow applications. These methods are continually improving and allow new and inconceivable designs to be produced because they do not need intuition and experience to solve a design problem. Intuition and experience can hinder a design optimization procedure by choosing the wrong initial configuration. As shown in Ch. 5, if the wrong configuration is chosen, these types of algorithms can find the correct one.

## 6.1 Conclusions

Herein, an optimization methodology was presented using the parameterized LSM, combined with mathematical programming techniques, to optimize the locations and coefficients of RBFs given secondary data produced by a known topology. The examples presented illustrate the strengths and weaknesses of the optimization methodology and several conclusions can be drawn. First and foremost, the algorithm can, in fact, perform topology optimization. That is, given an initial guess topology the output optimal topology can be different. Moreover, the optimization scheme can perform shape optimization for single and multiple bodies, as well. The scheme also generates a curvature continuous representation of the designs at each iteration.

Reinitialization was shown to improve the overall optimization results by redefining the locations and coefficients of the RBFs in better locations based on the current design. The design at the end of the optimization run more closely matches the optimal design and the optimal x-rays, thus using this design as a new initial guess produces a better approximation of the desired geometry. The redistribution of the RBFs also can decrease or increase the support radius, depending on whether the RBFs are spaced closer or farther apart. Again, this allows the optimizer to change the initial guess parameters and improve the overall results of the scheme. Furthermore, evidence that reinitialization improves the optimization results also indicates that better approximations for an initial guess will improve the efficiency and accuracy of the optimization scheme.

The effects of various initial guesses were investigated. It was shown that an initial guess that more closely approximates the desired geometry can decrease the total time the optimization algorithm takes to complete and can improve the minimization of the objective function. This suggests that it is useful to explore low-fidelity method for generating initial guesses for topology optimization problems. In particular, for fluid dynamics applications an algorithm that applies potential flow theory can be useful for generating an initial guess that can be used in a high-fidelity viscous flow solver and can be a direction of future research. Since this optimization technique is new, there are many outlets for investigation and exploration.

## 6.2 Suggested future research

This work presents an outlet for a variety of further developments and research opportunities. In particular, there is a lack of research in topology optimization for aerodynamic design. Improving upon the current algorithm and investigating various effects of the level-set parameterization can prove informative to the fluid dynamics and design optimization community. Furthermore, extension to 3D could be interesting for the design of turbine blades, grid fins, and HVAC baffles for fluid flow delivery systems. The LSM ideas and theory extend easily into 3D, however, the method of boundary generation would be more complex. More ideas for future work involving this optimization technique are detailed here.

### 6.2.1 Algorithm development

The algorithm can be greatly improved by optimizing the various operations and converting the MATLAB code into C or another open-source language like Python. By doing so, others can use the algorithm and adapt it for their own purposes and potentially find new applications. Furthermore, conversion to C would allow for the algorithm to plug into open-source CFD solvers like SU2, which is discussed in a subsequent section. Code conversion is not quite a research direction, but more of a research task. However, there are other aspects of algorithm development that would require investigation such as determining a better reinitialization technique, improving the level-set boundary detection method, testing a variety of RBFs and different support radius definitions (i.e., elliptical or super-elliptical).

The current reinitialization approach is to add an RBF inside and outside the boundary when the optimizer does not improve the objective function from one complete run to the next. While proven to be effective, this is more of a brute-force initial attempt than an elegant method. Possible approaches may include using the total boundary length to determine the number of RBFs, for instance, prescribing the distance between neighboring RBFs along the curve instead of the number of RBFs and decreasing the distance if the optimizer does not improve. Alternatively, clustering RBFs in high curvature regions to provide more control over the boundary could be useful. The benefits of a more thoughtful reinitialization scheme could result in decreased computational time for the optimizer as well as improved accuracy of the overall scheme.

The zero level-set curve method can be improved and is currently limited to only find bodies of a certain length scale and can potentially miss a curve. The method performs multiple ray-casting sweeps of the domain to locate the zero-crossings. As an alternative, the RBFs maybe able to act like sensors that can be used to determine if there is a zero-crossing. Simply check if the LSF is positive at any RBF, then find the closest RBF with a negative level-set value and use a bisection algorithm to locate the zero. One issue with this method is that it is not guaranteed that the zero level-set curve separates RBFs with positive and negative values. It is possible that the zero curve could exist without any RBFs within it. In any case, efforts to improve the detection method could result in improved robustness by being able to find all bodies in the domain, no matter the length scale.

Additionally, investigation of the effects of changing the support radius and using different shapes from a circular footprint may provide insights into how curves of varying continuity can be represented. Using an ellipse as the support radius with the axis aligned with the local tangent could improve the boundary smoothness and reduce undesirable variations. Moreover, use of a super-ellipse could have similar results while also giving added degrees of freedom to the optimization problem. Another possible direction for support radius investigations could be allowing the support radius to vary throughout the optimization and use it as a design variable.

### 6.2.2 Boundary variation control

The designs generated by the optimization algorithm are guaranteed to be curvature continuous because a cubic spline is fit through the points found along the zero level-set curve. However, the variation along the curve is not controlled which means that while the curve is $C^2$ there may be spurious oscillations introduced along the design boundaries during the optimization procedure, which may or may not be undesirable for high Reynolds number flow applications. The oscillations can be seen in several of the optimization results in the previous chapter as well as Appendix A and B. These variations in the boundary can cause designs in high Reynolds fluid flow regimes to perform poorly. First, an initial study of the how the algorithm works using a CFD solver for objective function evaluations should be conducted because the variations may not appear, as they could be detrimental to the performance of the design and have the effect of increasing the objective function instead of decreasing it.

Regularization or control may be necessary to ensure the variation along the boundary is minimized throughout the optimization. Implementation of a relaxation or artificial viscosity-type term as a penalty function may be useful. Developing methods to identify these variations and minimize them throughout the optimization could prove to be advantageous if this optimization technique is to be used for aerodynamic design optimization. Additionally, manipulation of the support radius may also have the effect of reducing the boundary variations.

The support radius has the effect of smoothing the boundary, but choosing an appropriate value for it is difficult because the locations and coefficients of the RBFs change throughout the optimization procedure. Choosing a support radius based on the spacing of the RBFs along the curve ensures that the boundary varies smoothly from RBF to RBF at initialization, but does not maintain this minimal variation throughout the optimization. An investigation of letting the support radius vary throughout the optimization procedure, or defining the support radius differently, for instance, as an ellipse aligned with the local boundary tangent may improve the smoothness. Efforts to control the boundary variation can aid in the robustness of this optimization scheme when coupled with a CFD solver; however, additional steps are required to extend this scheme to fluid dynamics problems and are discussed in the next section.

### 6.2.3 Investigation of fluid flow applications

The progression for developing the technique presented here into one that can perform topology optimization for aerodynamic designs is to start from grid-free aerodynamics (potential flow) and build on top of the algorithm. Each step is detailed here as a suggested schedule for future work.

#### Potential flow applications

The goal of the presented work was to match desired x-rays, where the x-rays acted as a surrogate for the wake behind an object in cross-flow. Potential flow problems provide a sandbox for testing and developing an appropriate objective function and possibly penalty functions to try to match the wake behind a given object with a level-set representation. Instead of using the cubic spline points to produce the x-rays, in the framework of a potential flow problem the spline points can be used to define source panels. With the source panels defining the boundary of the design, the potential flow solution can be obtained and the wake behind the design can be measured and compared to the desired wake. This project would indicate whether the optimizer can match the wake behind an object or if modifications are necessary to obtain a good result. From this, extension to high-fidelity would require implementing a grid generator as well as a mesh deformer.

#### ***delaundo*: 2D grid generation and deformation**

The grid is the connection between the design topology, the boundaries, and the fluid flow. A grid needs to be created so the flow physics is captured accurately and efficiently, in that, grid points are clustered in regions with high gradients and spaced out in regions of low gradients. This minimizes the computational cost of the flow solver. Grid generation can be performed using a structured or unstructured generator. Structured grids are much more difficult to generate automatically and require more computer power and time. For the initial investigation, unstructured grids will be used for the analysis. While there are numerous grid generation tools available, *delaundo* can be chosen as a starting point.

*delaundo* is a 2D unstructured grid generation tool developed by J.-D. Müller requiring minimal user input[130]. The Frontal Delaunay Method is used to connect the points on the internal boundaries to the outer boundaries while creating an internal mesh between the two. *delaundo* has been extensively tested and is shown to be robust for a variety of topologies. Furthermore, the software is open-source which makes it desirable to use for building the topology optimization tool that was presented here. Furthermore, a mesh deformer would need to be implemented for successive design updates. Being able to deform the mesh results in fewer grid regenerations and reduces the computational resources required for the optimization run. Additionally, a mesh deformer can be easily differentiated using algorithmic differentiation techniques so the sensitivities of the mesh deformer with respect to the surface points of the design can be found. Examples of existing mesh deformation techniques include RBF deformation, linear springs, and mean-value-coordinates. These methods can be used and the results compared to determine which is best for the optimization technique. With a mesh generator and deformation technique in place, a CFD solver can be used to evaluate the level-set design and output the objective function and sensitivities of the flow variables with respect to the mesh points.

### SU2: the high-fidelity model

SU2 is a well-known computational fluid dynamics flow solver that was created for aerodynamic design optimization purposes[131]. SU2 can be used to model viscous flow over a topology generated by the LSM. The software outputs a variety of data that includes fluid flow field quantities, such as velocities and pressure, as well as their distributions along the design surfaces. The calculation of these other quantities, such as pressure loss across the domain, can be added to the software and set as an objective function. Having the ability to choose an objective function is important because that function is the basis for adjoint-based optimization. The key attribute that makes SU2 desirable is the fact that it can calculate the adjoint of a user-defined objective function using algorithmic differentiation. With the adjoint produced from SU2, the optimization scheme presented here can calculate the derivatives of the objective function with respect to the locations and coefficients of the RBFs. This can be accomplished by modifying the algorithms to call SU2 as the flow solver and adjoint calculator to obtain the gradients of the objective function with respect to the fluid flow variables. These gradients can then be used to relate the objective function to the level-set design parameters.

Normally, for an adjoint-based optimization scheme, the entire code needs to be differentiated to relate the objective function to the design parameters. Instead SU2 provides the sensitivities of the objective function to the design surfaces, so only the sensitivities between the LSF and the grid generation tool need to be calculated externally. One issue arises when attempting to calculate the sensitivities of the field mesh to the surface mesh and then the surface mesh to the design variables. If a new grid were to be generated during every optimization iteration, the process that *delaundo* uses to create the grid would need to be differentiated with respect to surface point location. This raises the question, can the *delaundo* code be differentiated? Instead of using a code differentiation tool to solve this problem, another method can be used for grid augmentation and deformation.

RBF interpolation has been developed for just this purpose[132,133]. The idea is to develop a deformation function using RBFs. The RBFs are assigned displacement values at the surface that is being deformed and these displacement values are then interpolated to every point in the field. The advantage of using this method for mesh deformation is that the calculation is linear so the derivatives are easily obtained. Therefore, RBF interpolation provides a computationally efficient method for calculating design sensitivities of the field mesh and the surface mesh. Furthermore, since a cubic spline is the output of the optimization scheme, the points can be distributed along the boundary at the discretion of the end-user. So, while the boundary and the spline points change throughout the optimization process, the points can be distributed in a way that mesh deformation techniques can propagate the surface changes throughout the mesh.

Once the grid sensitivities are calculated, the sensitivities of the surface grid to the level-set parameters is required. Recall, the level-set sensitivities were calculated in Ch. 4 analytically. The sensitivities are given to the optimizer and the design is changed accordingly and new sensitivities are subsequently calculated. This process is repeated until an optimal design is found.

The advantages of using an analytical approach are improved accuracy of the sensitivities as well as reduced computational cost than finite-differences[134]. A disadvantage (or difficulty) is that the procedure for creating the geometry must be known. Using finite-differences is advantageous when the process for making the geometry is unknown, but the geometry needs to be perturbed and remodeled to calculate the sensitivities. This procedure can be prohibitive due to the large computational cost. Fortunately, the geometry is generated from a LSM, so the analytical approach can be used.

The work presented herein is an initial development and exploration of a new topology optimization technique that maintains curvature continuous designs throughout the optimization procedures. The algorithms and ideas can be used to extend the technique to fluid flow applications and some recommendations for future work have been made. The hope of the author is that these ideas spark intrigue and curiosity so further investigation and development of these types of methods in the aerodynamics community continues.

# Appendix A

# Supplemental test case examples

Within this appendix the various test cases are presented in the same for as in the document. First the initial parameters are presented followed by a figure illustrating the initial setup. Then the results of the optimization scheme are presented along with the final design parameters summarized in a table.

## A.1 Circle

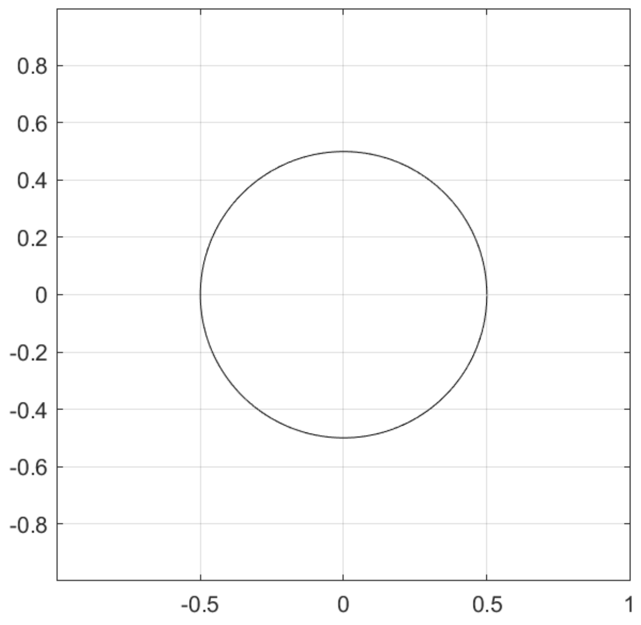
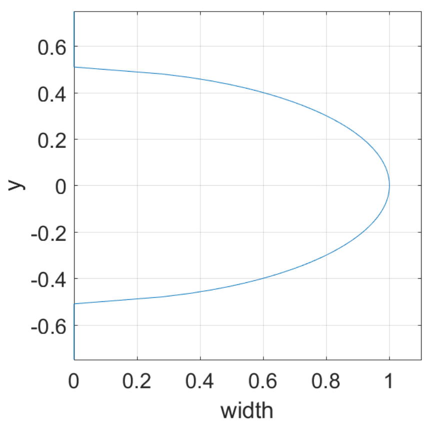


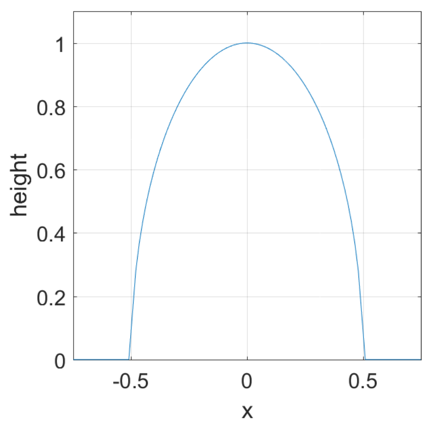
Figure A.1: Geometry used for x-ray generation

Table A.1: Initial parameters for circle case

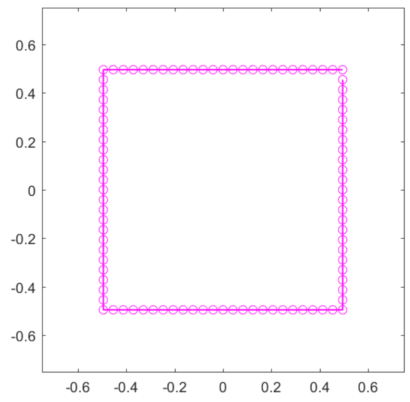
| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **20** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.96** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Rectangular** | Type of initial guess used for optimization |
| $N_{grid}$ | **25** | Number of points distributed around each side of the initial guess |
| $N_{rays}$ | **101** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | $\mathbf{9.49}\times 10^{-2}$ | Initial objective function value |



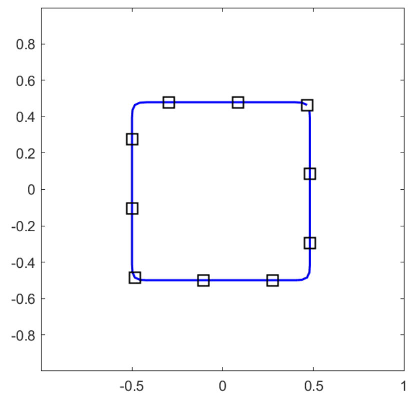
(a) Desired horizontal x-rays



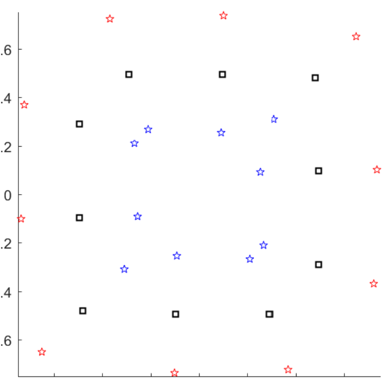
(b) Desired vertical x-rays



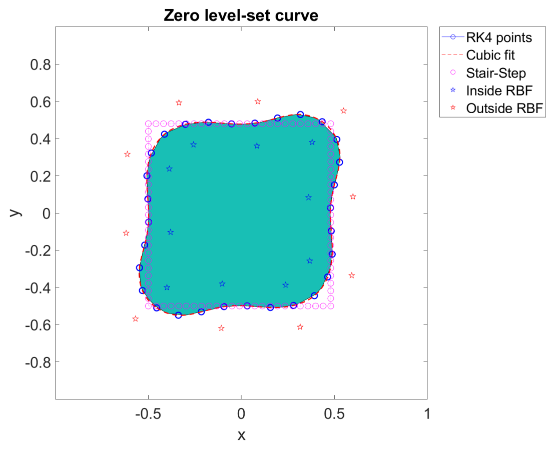
(c) Initial boundary representation



(d) Linearly smoothed curve with RBF midpoints

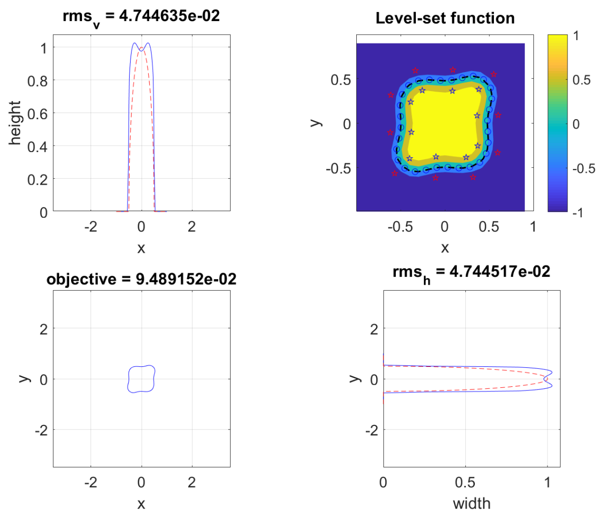


(e) RBF distribution. red - outside, blue - inside

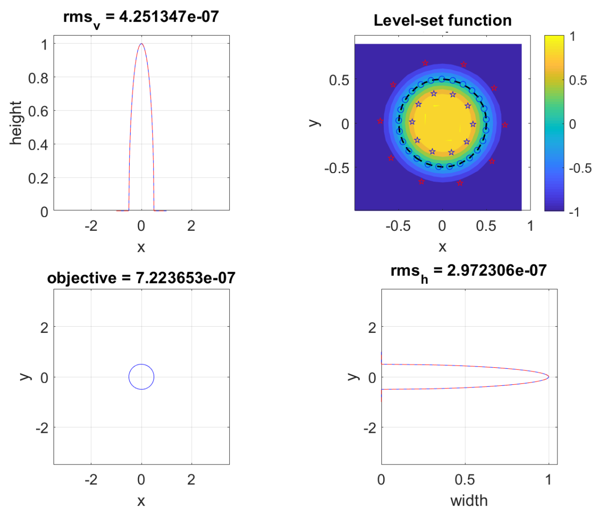


(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure A.2: Setup for circle case



(a) Initial objective function evaluation



(b) Final optimum objective function evaluation

Figure A.3: Optimization test case: circle

Table A.2: Final parameters for circle case

| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **20** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.77** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **28** | Total number of iterations |
| $N_{reinit}$ | **1** | Total number of reinitializations |
| $N_{eval}$ | **4177** | Total number of function evaluations |
| $T_{opt}$ | **0.13** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | **7.22**$\times 10^{-7}$ | Optimum value for test case |

## A.2 Turbine blade

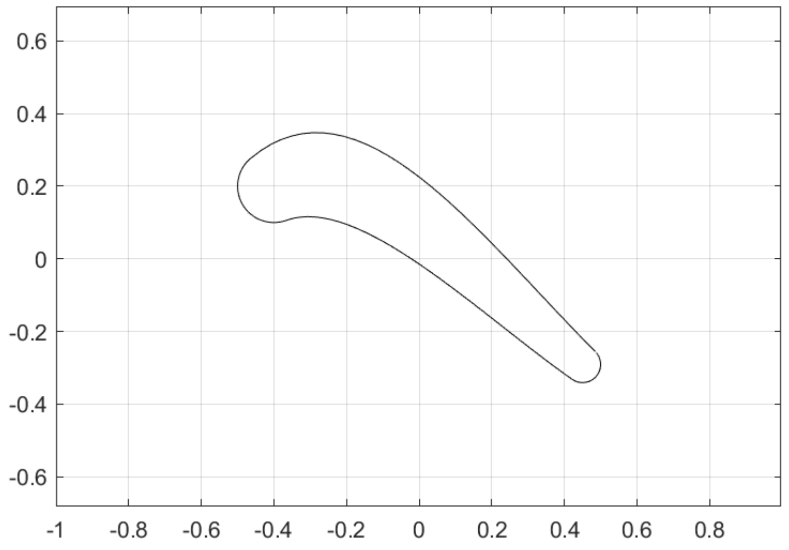
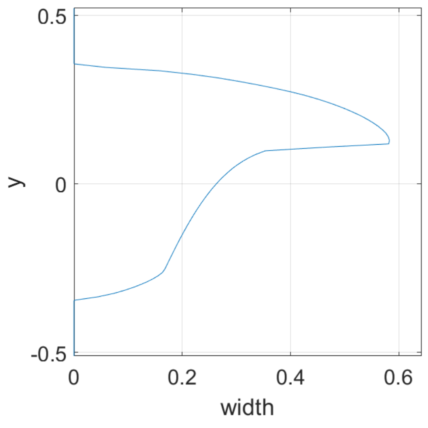


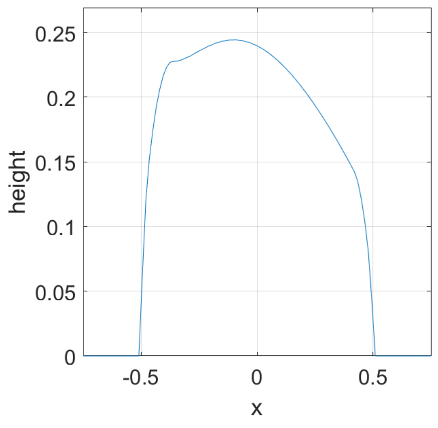
Figure A.4: Geometry used for x-ray generation

Table A.3: Initial parameters for turbine blade case

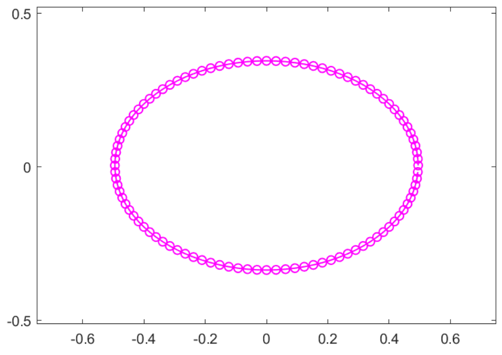
| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **20** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.65** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Elliptical** | Type of initial guess used for optimization |
| $N_{grid}$ | **18** | Number of points distributed around each side of the initial guess |
| $N_{rays}$ | **101** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | $\mathbf{2.32}\times10^{-1}$ | Initial objective function value |

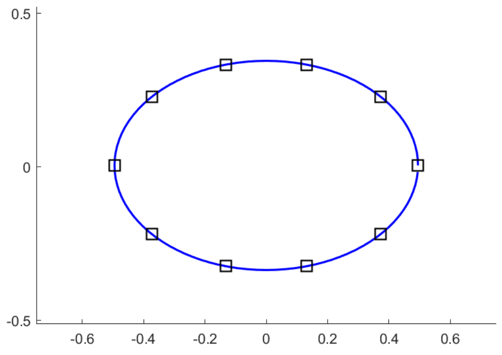


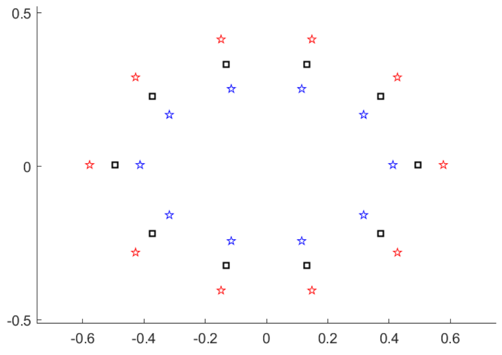
(a) Desired horizontal x-rays

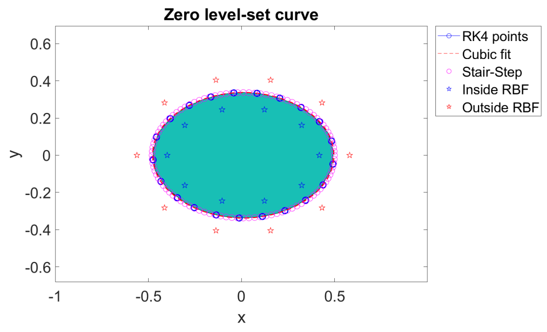


(b) Desired vertical x-rays


(c) Initial boundary representation

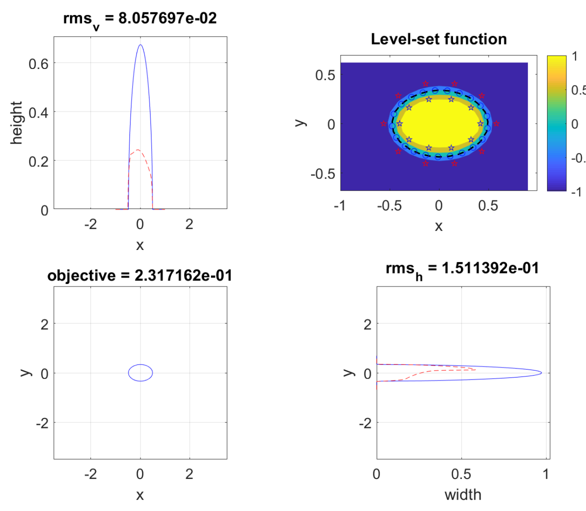

(d) Linearly smoothed curve with RBF midpoints


(e) RBF distribution. red - outside, blue - inside

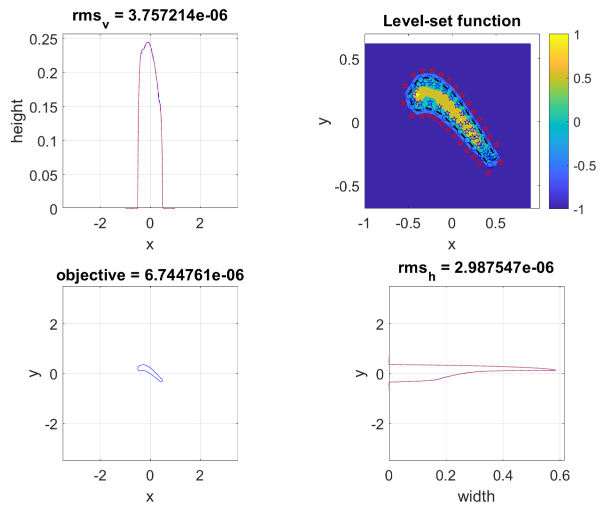


(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure A.5: Setup for turbine blade case



(a) Initial objective function evaluation



(b) Final optimum objective function evaluation

Figure A.6: Optimization test case: turbine blade

Table A.4: Final parameters for turbine blade case

| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **56** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.23** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **304** | Total number of iterations |
| $N_{reinit}$ | **20** | Total number of reinitializations |
| $N_{eval}$ | **51807** | Total number of function evaluations |
| $T_{opt}$ | **4.07** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | $\mathbf{6.74}\times10^{-6}$ | Optimum value for test case |

## A.3 Vertical Ellipses

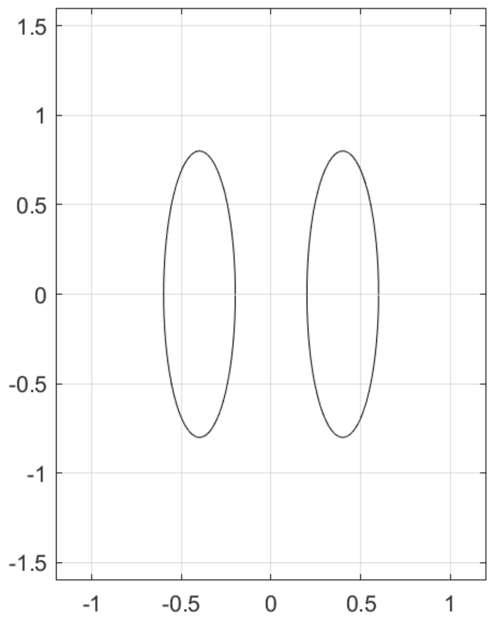
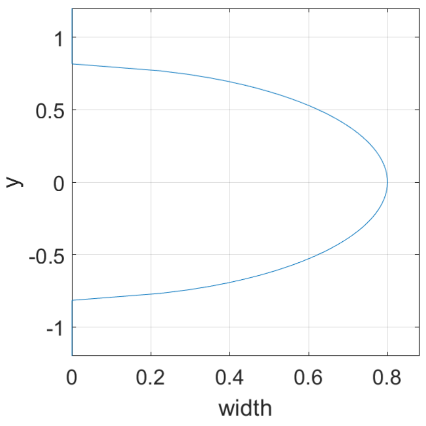


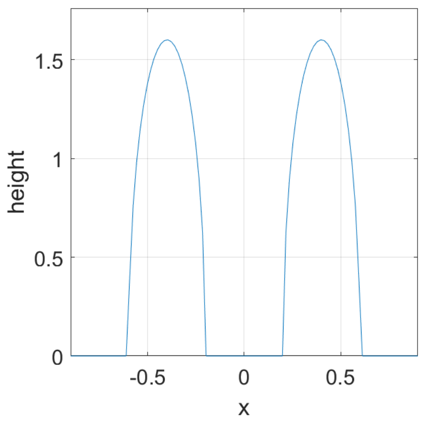
Figure A.7: Geometry used for x-ray generation

Table A.5: Initial parameters for vertical ellipses case

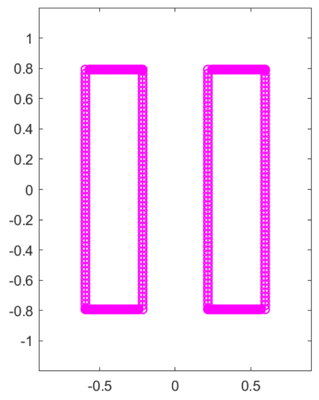
| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **40** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.97** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Rectangular** | Type of initial guess used for optimization |
| $N_{grid}$ | **50** | Number of points distributed around each side of the initial guess |
| $N_{rays}$ | **101** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | **2.11**$\times 10^{-1}$ | Initial objective function value |



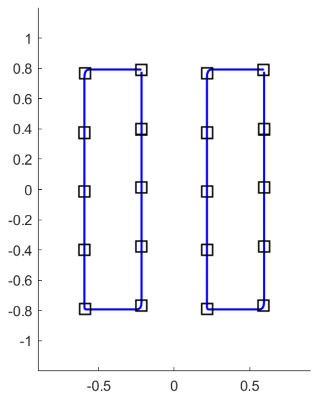
(a) Desired horizontal x-rays



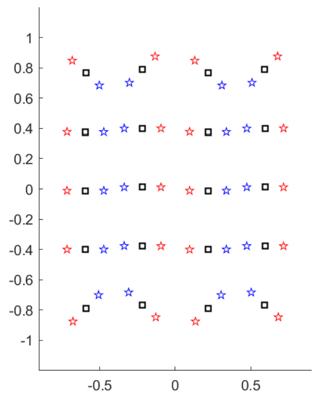
(b) Desired vertical x-rays



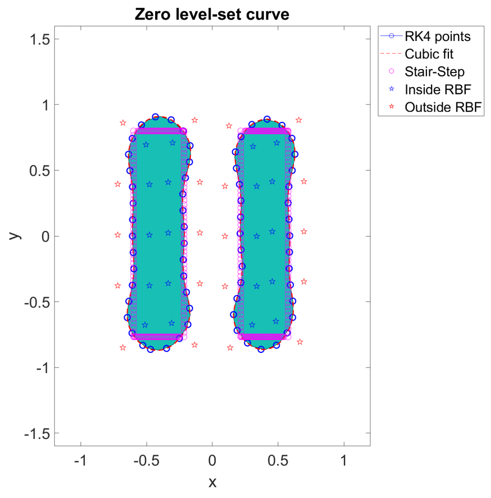
(c) Initial boundary representation



(d) Linearly smoothed curve with RBF midpoints

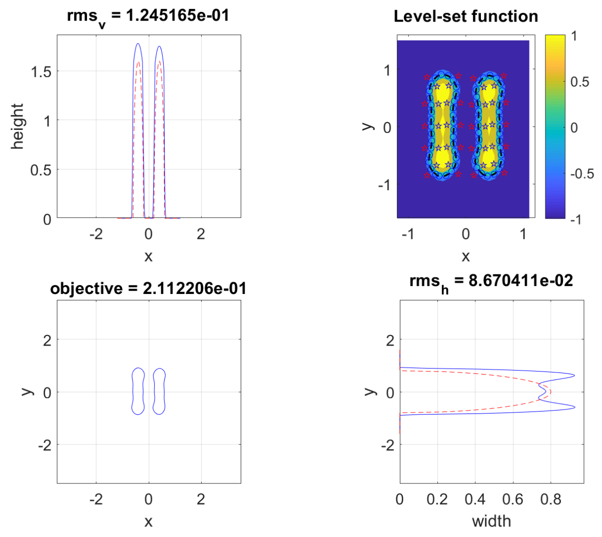


(e) RBF distribution. red - outside, blue - inside

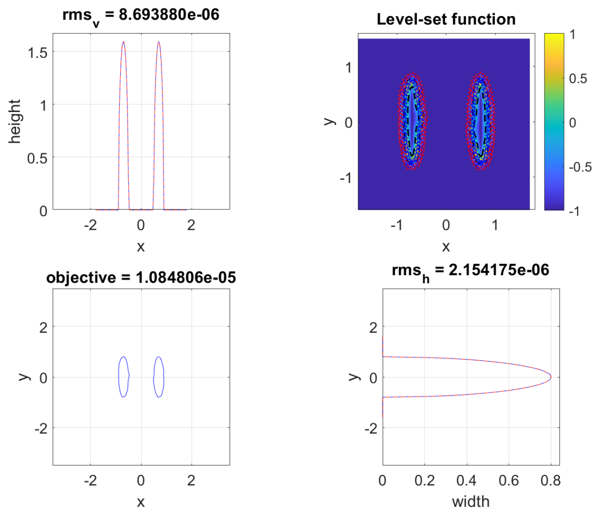


(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure A.8: Setup for vertical ellipses case



(a) Initial objective function evaluation



(b) Final optimum objective function evaluation

Figure A.9: Optimization test case: vertical ellipses

Table A.6: Final parameters for vertical ellipses case

| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **196** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.18** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **667** | Total number of iterations |
| $N_{reinit}$ | **57** | Total number of reinitializations |
| $N_{eval}$ | **244584** | Total number of function evaluations |
| $T_{opt}$ | **29.22** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | $\mathbf{1.08{\times}10^{-5}}$ | Optimum value for test case |

## A.4 Cascade of Ellipses

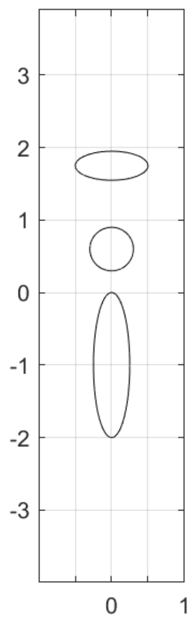
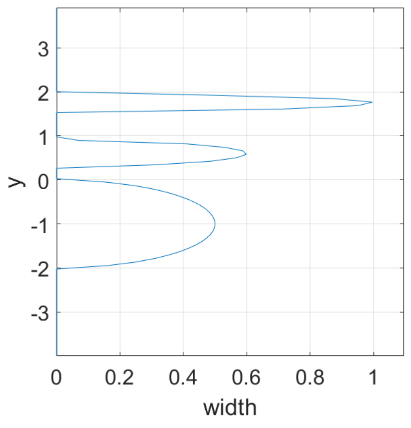


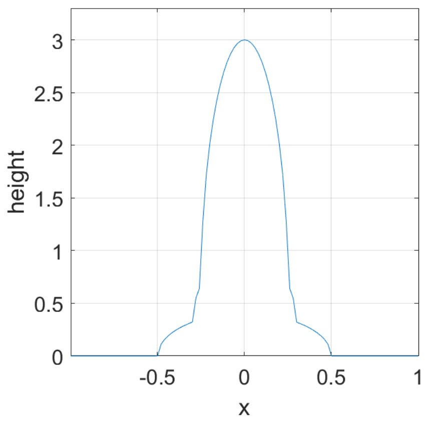
Figure A.10: Geometry used for x-ray generation

Table A.7: Initial parameters for cascade of ellipses case

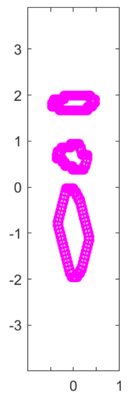
| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **60** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **1.06** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Stair-stepped** | Type of initial guess used for optimization |
| $N_{grid}$ | **50** | Number of points distributed around each side of the initial guess |
| $N_{rays}$ | **101** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | **7.64**$\times 10^{-2}$ | Initial objective function value |



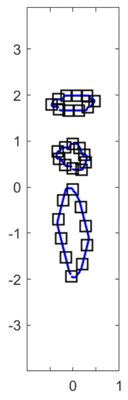
(a) Desired horizontal x-rays



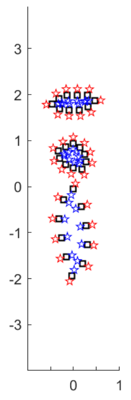
(b) Desired vertical x-rays



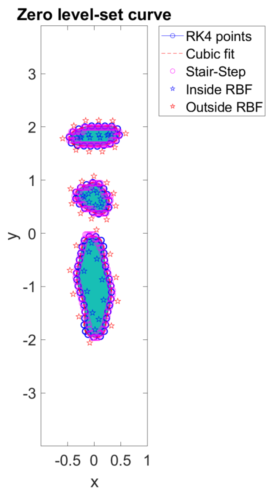
(c) Initial boundary representation



(d) Linearly smoothed curve with RBF midpoints

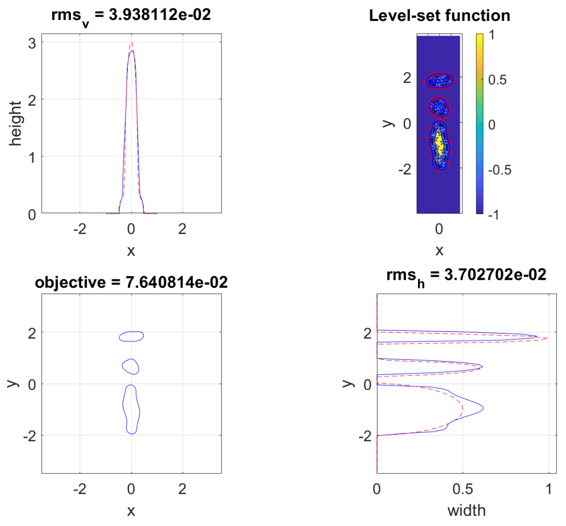


(e) RBF distribution. red - outside, blue - inside

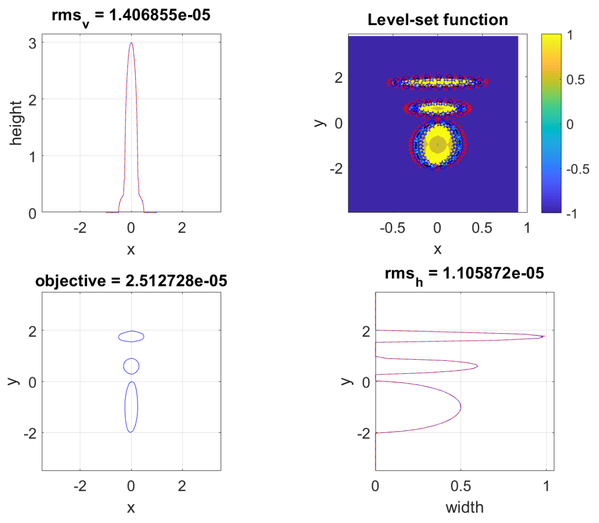


(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure A.11: Setup for cascade of ellipses case



(a) Initial objective function evaluation



(b) Final optimum objective function evaluation

Figure A.12: Optimization test case: cascade of ellipses

Table A.8: Final parameters for cascade of ellipses case

| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **156** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.41** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **2108** | Total number of iterations |
| $N_{reinit}$ | **30** | Total number of reinitializations |
| $N_{eval}$ | **909515** | Total number of function evaluations |
| $T_{opt}$ | **51.83** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | $\mathbf{2.51}\times10^{-5}$ | Optimum value for test case |

## A.5 Vertical Ellipses to an Arbitrary Body

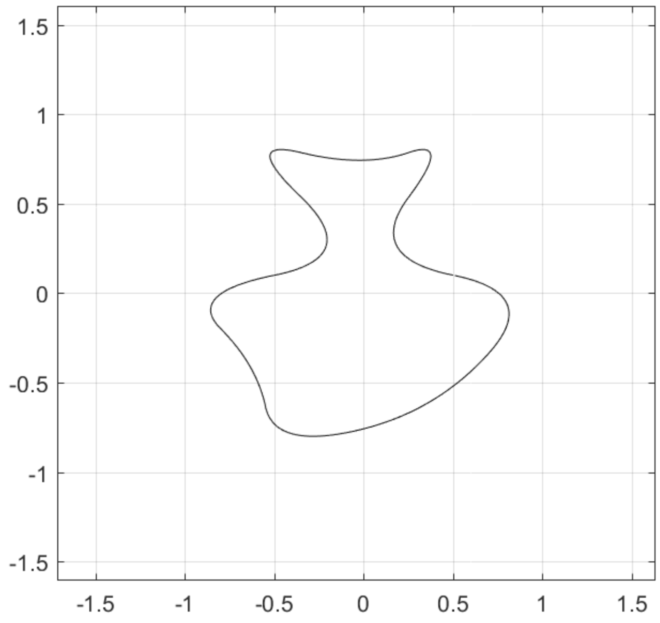
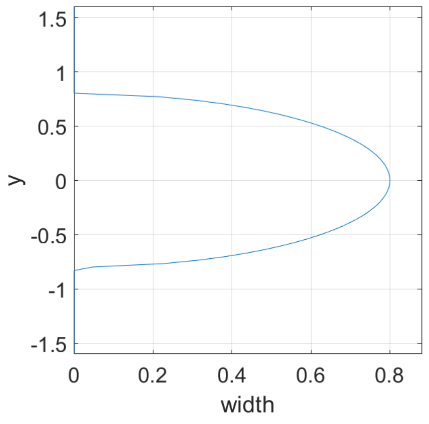


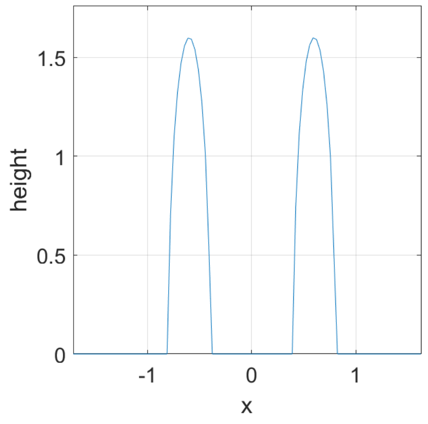
Figure A.13: Geometry used for x-ray generation

Table A.9: Initial parameters for vertical ellipses to an arbitrary body case

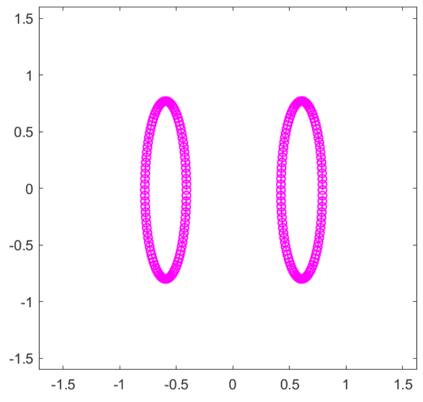
| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **40** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.84** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Elliptical** | Type of initial guess used for optimization |
| $N_{grid}$ | **50** | Number of points distributed around each side of the initial guess |
| $N_{rays}$ | **101** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | $\mathbf{5.76}\times10^{-1}$ | Initial objective function value |



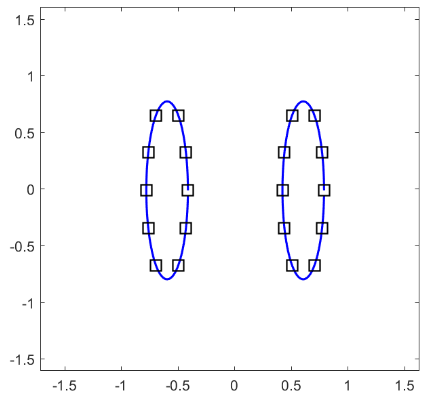
(a) Desired horizontal x-rays



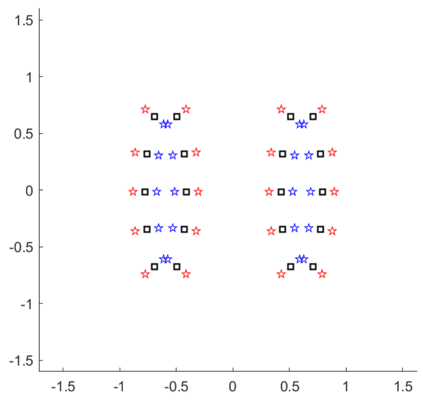
(b) Desired vertical x-rays



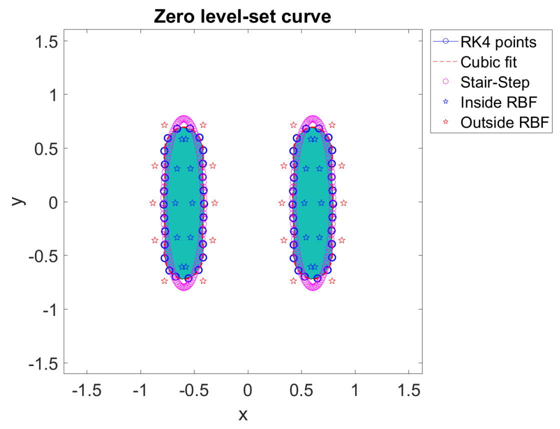
(c) Initial boundary representation



(d) Linearly smoothed curve with RBF midpoints

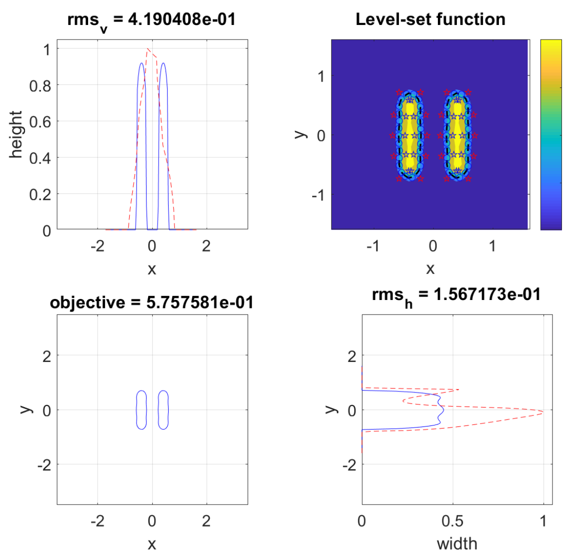


(e) RBF distribution. red - outside, blue - inside

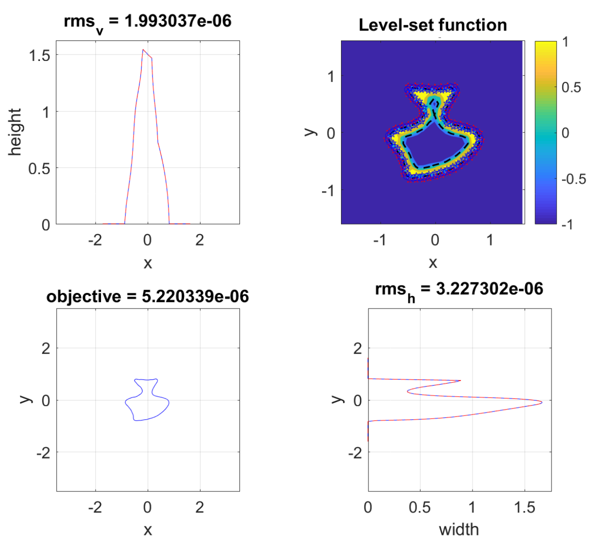


(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure A.14: Setup for vertical ellipses to an arbitrary body case



(a) Initial objective function evaluation



(b) Final optimum objective function evaluation

Figure A.15: Optimization test case: vertical ellipses to an arbitrary bodys

Table A.10: Final parameters for vertical ellipses to an arbitrary body case

| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **102** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.29** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **1830** | Total number of iterations |
| $N_{reinit}$ | **58** | Total number of reinitializations |
| $N_{eval}$ | **460615** | Total number of function evaluations |
| $T_{opt}$ | **75.69** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | $\mathbf{5.22}\times 10^{-6}$ | Optimum value for test case |

## A.6 Vertical ellipse to Cascade Ellipses

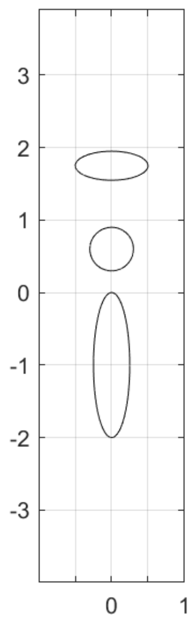
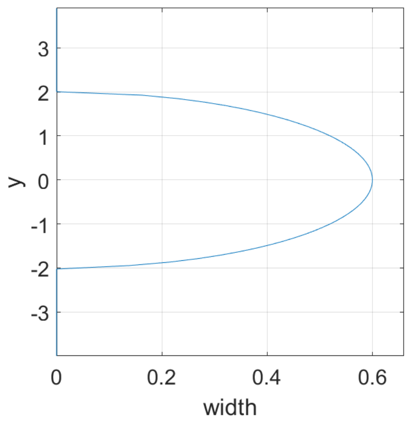


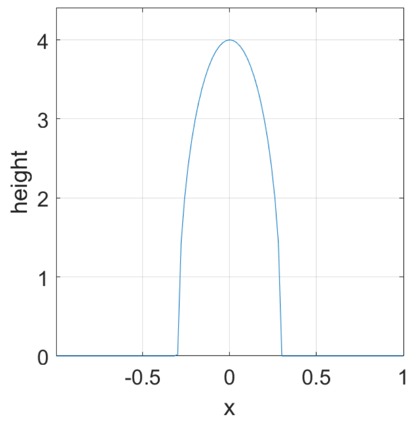
Figure A.16: Geometry used for x-ray generation

Table A.11: Initial parameters for cascade of ellipses case

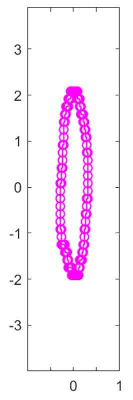
| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **30** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **1.10** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Stair-Stepped** | Type of initial guess used for optimization |
| $N_{grid}$ | **25** | Number of points distributed around each side of the initial guess |
| $N_{rays}$ | **201** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | **6.55**$\times 10^{-1}$ | Initial objective function value |



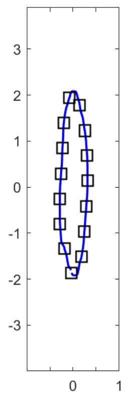
(a) Desired horizontal x-rays



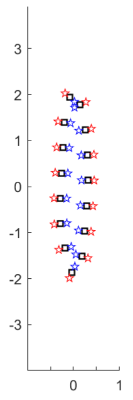
(b) Desired vertical x-rays



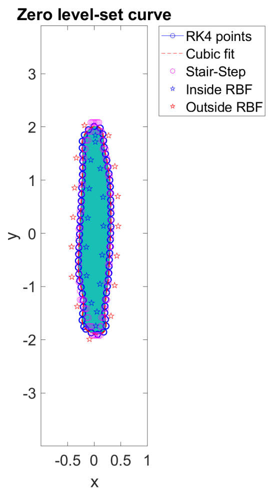
(c) Initial boundary representation



(d) Linearly smoothed curve with RBF midpoints

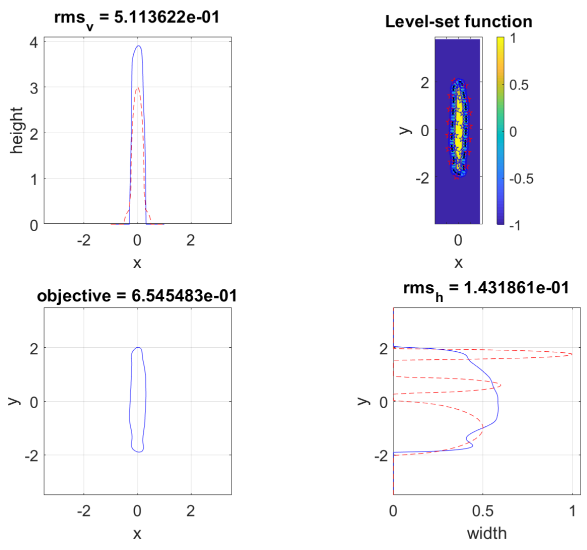


(e) RBF distribution. red - outside, blue - inside

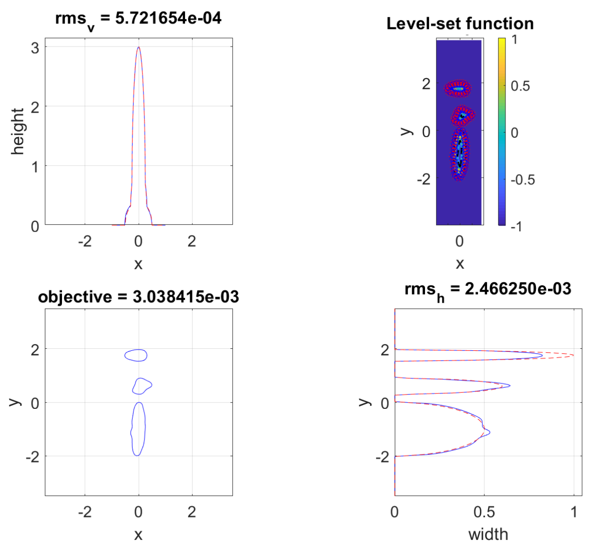


(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure A.17: Setup for vertical ellipse to cascade of ellipses case



(a) Initial objective function evaluation



(b) Final optimum objective function evaluation

Figure A.18: Optimization test case: cascade of ellipses

Table A.12: Final parameters for cascade of ellipses case

| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **56** | Number of radial basis functions |
| $N_{SR}$ | **2.5** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.31** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **286** | Total number of iterations |
| $N_{reinit}$ | **25** | Total number of reinitializations |
| $N_{eval}$ | **113302** | Total number of function evaluations |
| $T_{opt}$ | **11.18** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | $\mathbf{3.04}\times10^{-3}$ | Optimum value for test case |

## A.7 Single Body to Two Vertical Ellipses

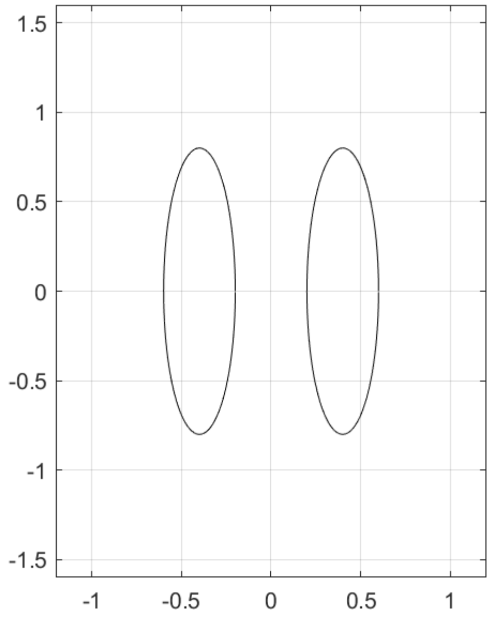
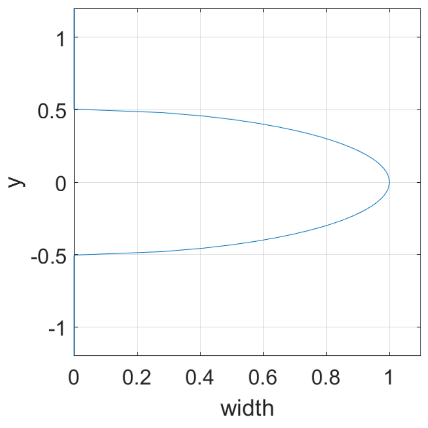


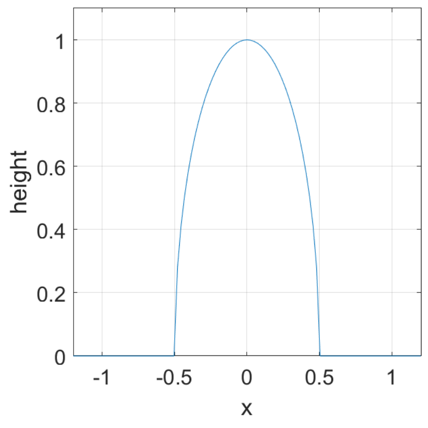
Figure A.19: Geometry used for x-ray generation

Table A.13: Initial parameters for single body to two vertical ellipses case

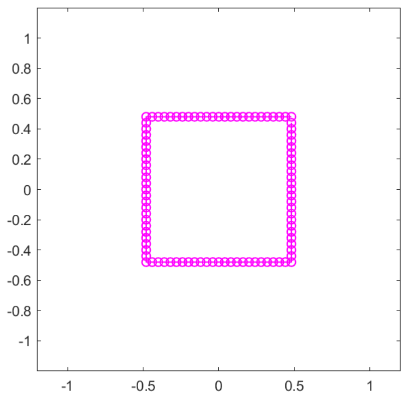
| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **20** | Number of radial basis functions |
| $N_{SR}$ | **2.0** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.94** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Rectangular** | Type of initial guess used for optimization |
| $N_{grid}$ | **25** | Number of points distributed around each side of the initial guess |
| $N_{rays}$ | **101** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | $\mathbf{1.13}\times10^{+0}$ | Initial objective function value |



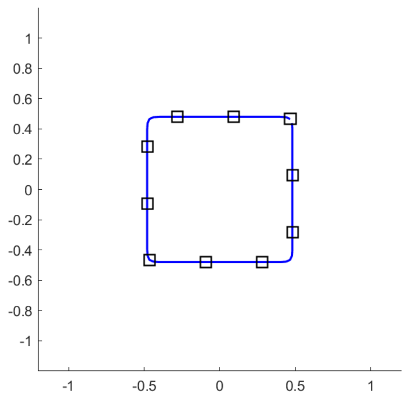
(a) Desired horizontal x-rays



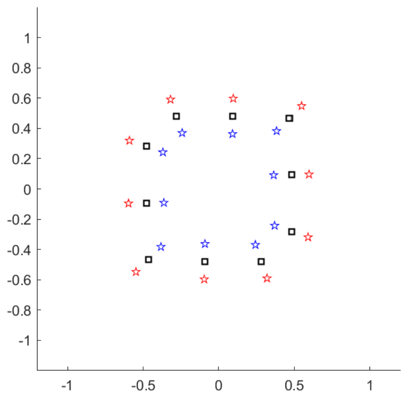
(b) Desired vertical x-rays



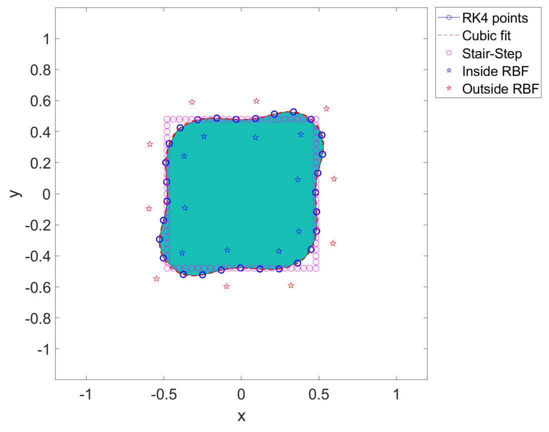
(c) Initial boundary representation



(d) Linearly smoothed curve with RBF midpoints

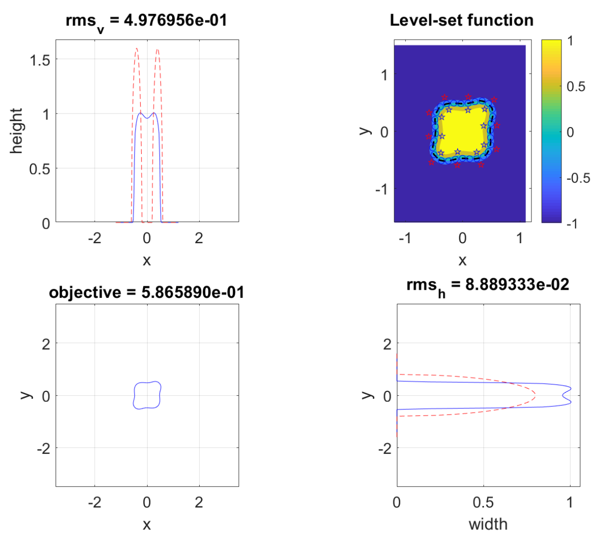


(e) RBF distribution. red - outside, blue - inside

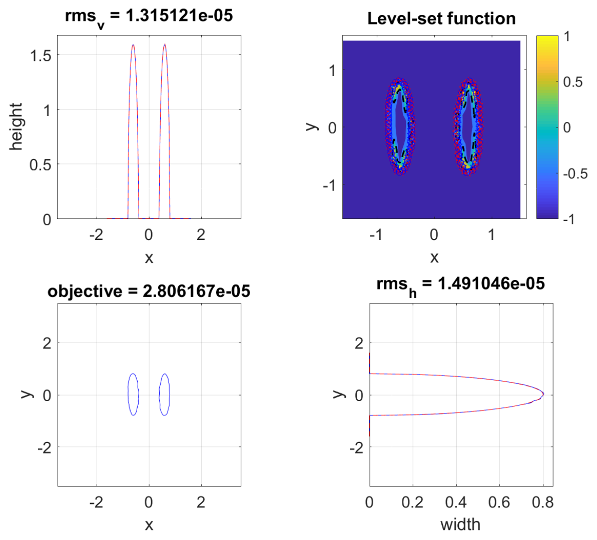


(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure A.20: Setup for single body to two vertical ellipses case



(a) Initial objective function evaluation



(b) Final optimum objective function evaluation

Figure A.21: Optimization test case: single body to two vertical ellipses

Table A.14: Final parameters for single body to two vertical ellipses case

| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **176** | Number of radial basis functions |
| $N_{SR}$ | **2.0** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.16** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **876** | Total number of iterations |
| $N_{reinit}$ | **50** | Total number of reinitializations |
| $N_{eval}$ | **283535** | Total number of function evaluations |
| $T_{opt}$ | **24.98** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | $\mathbf{2.81}\times10^{-5}$ | Optimum value for test case |

## A.8 Single Body to Two Diagonal Ellipses

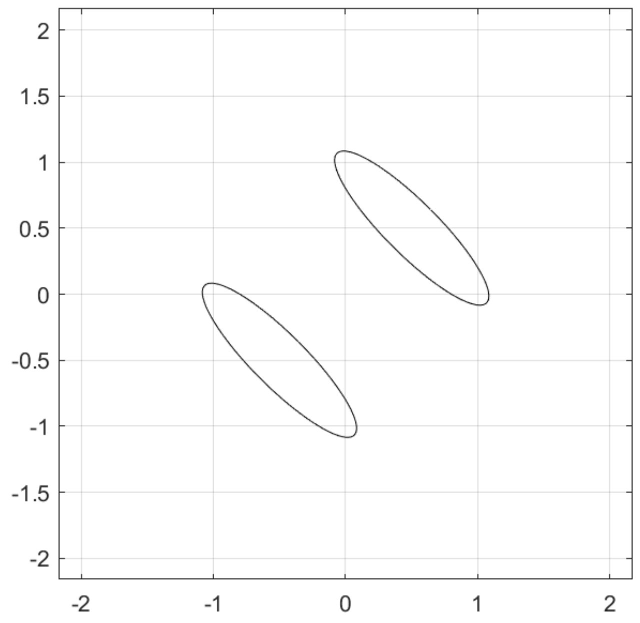
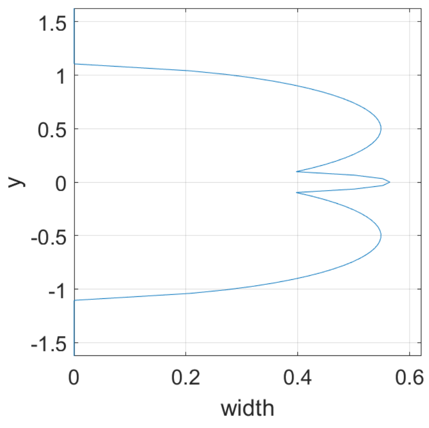


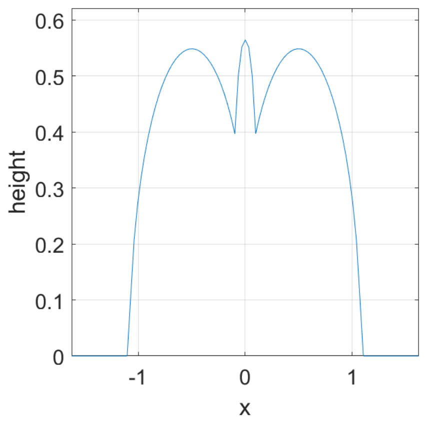
Figure A.22: Geometry used for x-ray generation

Table A.15: Initial parameters for single body to diagonal ellipses case

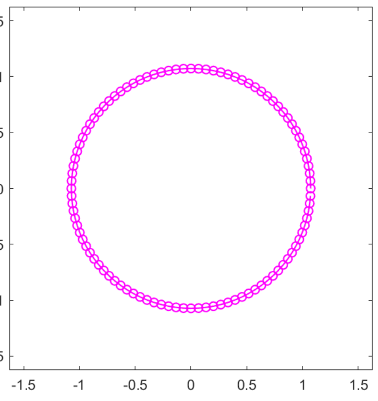
| *Parameter* | **Value/Type** | Description |
|---|---|---|
| $N_{RBF}$ | **40** | Number of radial basis functions |
| $N_{SR}$ | **2.0** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.66** | Support radius in w.r.t. to domain axes |
| $F_{SR}$ | **0.25** | Fraction of the support radius that the RBFs are offset from initial curve |
| Initial guess | **Rectangular** | Type of initial guess used for optimization |
| $N_{grid}$ | **50** | Number of points distributed around each side of the initial guess |
| $N_{rays}$ | **101** | Number of rays used for objective calculation |
| $O(\vec{p}_0)$ | $\mathbf{3.29}\times10^{+0}$ | Initial objective function value |

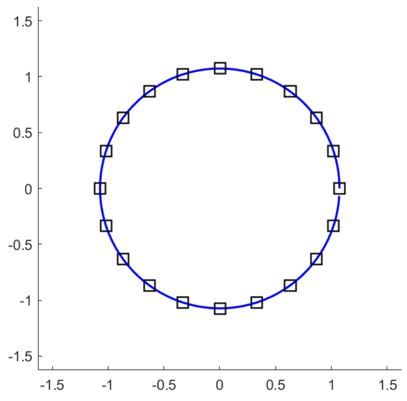


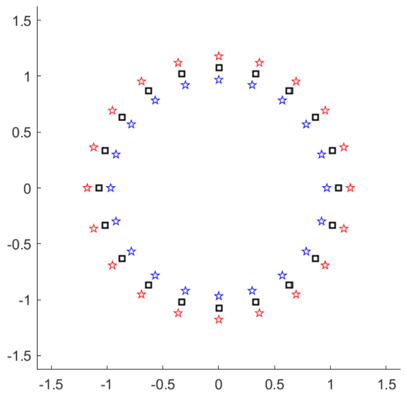
(a) Desired horizontal x-rays

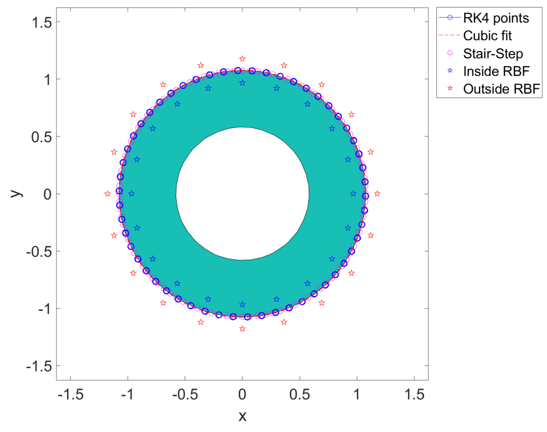


(b) Desired vertical x-rays


(c) Initial boundary representation

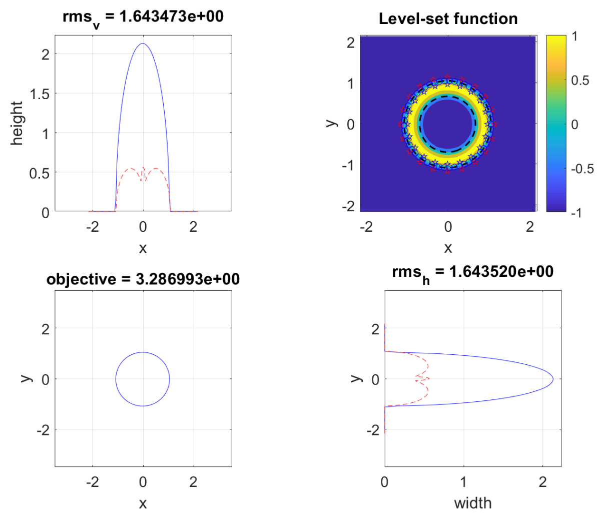

(d) Linearly smoothed curve with RBF midpoints


(e) RBF distribution. red - outside, blue - inside

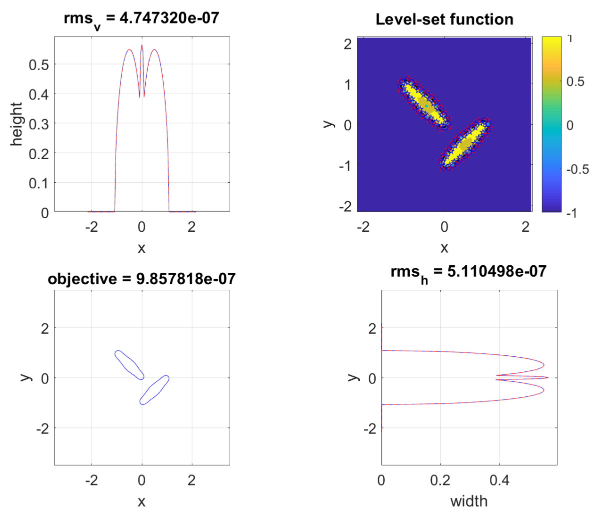


(f) Level-set representation with cubic fit superimposed on initial boundary guess

Figure A.23: Setup for single body to diagonal ellipses case



(a) Initial objective function evaluation



(b) Final optimum objective function evaluation

Figure A.24: Optimization test case: single body to diagonal ellipses

Table A.16: Final parameters for single body to diagonal ellipses case

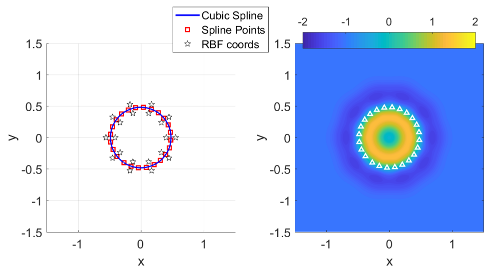
| *Parameter* | **Value** | Description |
|---|---|---|
| $N_{RBF}$ | **40** | Number of radial basis functions |
| $N_{SR}$ | **2.0** | Support radius in w.r.t. RBF spacing |
| $SR$ | **0.34** | Support radius in w.r.t. to domain axes |
| $N_{iter}$ | **196** | Total number of iterations |
| $N_{reinit}$ | **6** | Total number of reinitializations |
| $N_{eval}$ | **55064** | Total number of function evaluations |
| $T_{opt}$ | **2.94** | Total time for optimization (hours) |
| $O(\vec{p}_{opt})$ | **9.86**$\times 10^{-7}$ | Optimum value for test case |

# Appendix B

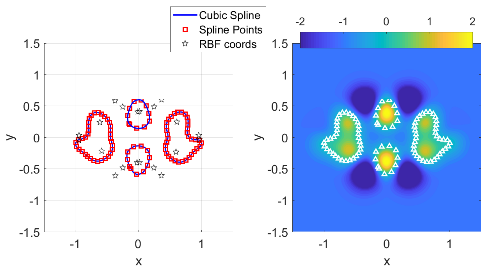
# Supplemental topology change examples

This appendix shows two separate test cases where the initial topology is different from the final. The setup and result for each case can be found in Appendix A. Herein, the topological changes throughout the optimization process are shown. Each figure shows eight iterations of the optimization process for each test case. Each subfigure displays the RBF locations, the spline points and curve, and the level-set representation at the current iteration.

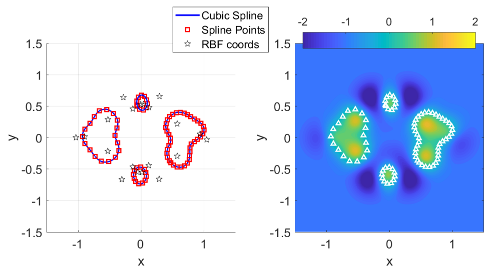
## B.1 Circle to two vertical ellipses



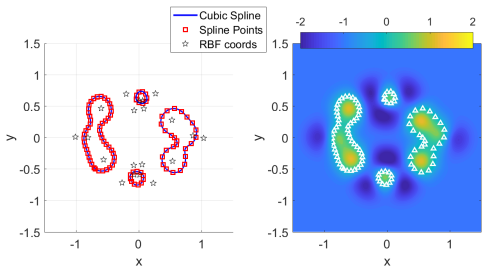
(a) Initial



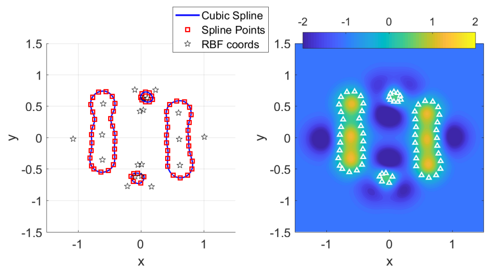
(b) Iteration 1



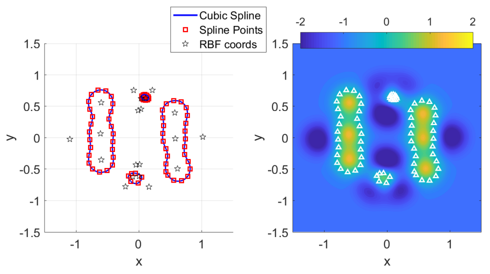
(c) Iteration 5



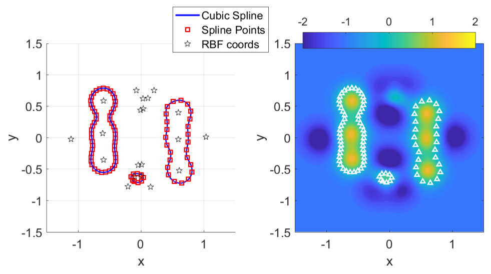
(d) Iteration 8



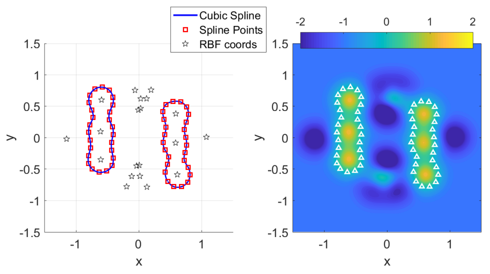
(e) Iteration 12



(f) Iteration 15



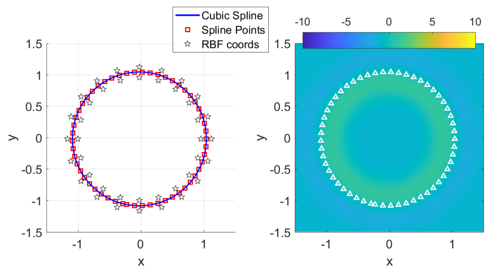
(g) Iteration 18



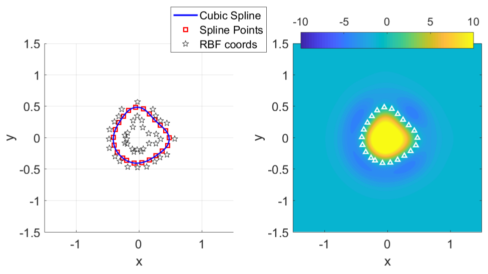
(h) Iteration 25

Figure B.1: Tearing example: cubic fit and RBF locations (left) and the level-set function (right) are shown in each subfigure.

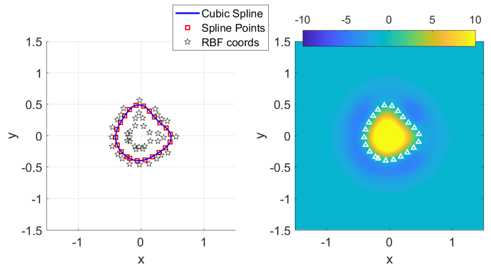
## B.2 Circle to two diagonal ellipses



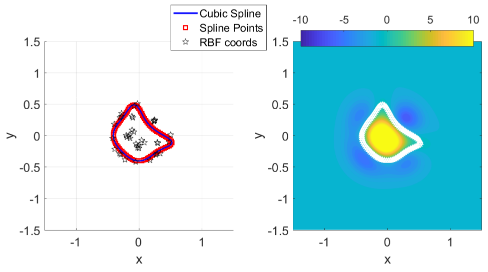
(a) Initial



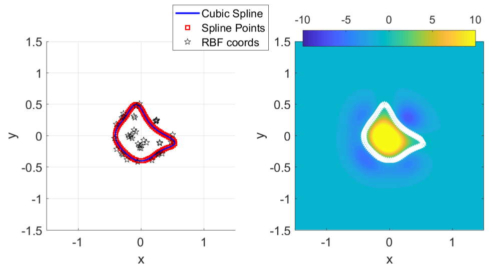
(b) Iteration 1



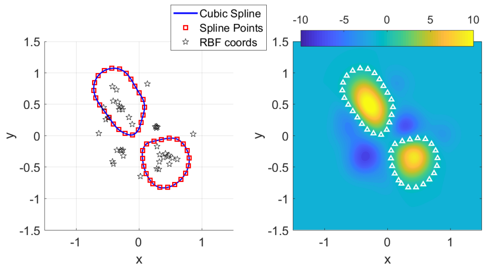
(c) Iteration 5



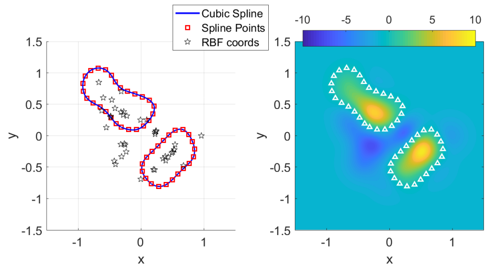
(d) Iteration 12



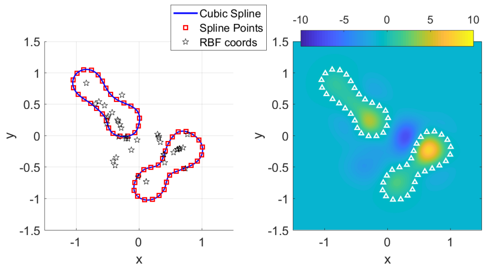
(e) Iteration 14



(f) Iteration 15



(g) Iteration 21



(h) Iteration 41

Figure B.2: Tearing example: cubic fit and RBF locations (left) and the level-set function (right) are shown in each subfigure.

# Appendix C

# Code Listing

```
function [] = levelset_topology_optimization( )
    % function levelset_topology_optimization() serves as the main
    % code for the topology optimization scheme developed by Jack Rossetti
    % under mentorship of Dr. Dannenhoffer and Dr. Green.
    %
    % Written by Jack Rossetti
    % Annotated by Jack Rossetti__________________________________02/24/20
    close all
    %
    % Case parameters:
    %
    case_num =   3;        % Case number
    icase    =   6;        % Initial geom
    jgeom    =   6;        % Desired geom
    rect_elli=   2;        % Initial guess generation,
                           % 0 - rectangle(s)
                           % 1 - ellipse(s)
                           % 2 - stair-stepper
    %
    % RBF parameters:
    %
    nRBF     =   15;        % Number of RBFs divided by two
    SR       =    0.50;     % Support radius
    nSR      =    2.5;      % Number of RBFs affected by each
    fSR      =    0.25;    % Fraction of offset from initial curve of
                            % inside/outside RBFs
                            % w.r.t. SR
    OFFSET   =    1.0;      % Offset to obtain zero-curve
    inout    =    1;        % 0 for only inside RBFs,
                            % 1 for inside and outside RBFs
    %
    % Boundary fitting parameters:
    %
    poly_fit =    2;        % Fit type for points generator:
                            % 1 - linear spline,
                            % 2 - cubic polygon (cubic fit with linear
                            %     segments)
                            % 3 - cubic spline
    %
    % Objective parameters:
    %
    nrays    =  101;        % Number of rays used in objective
    NRML     =    0;        % 0 unscaled objective,
                            % 1 scaled    objective
    %
    % Sensitivity parameters:
    %
    sens_calc=    4;        % type of derivative calculation:
                            % 1 - finite difference,
                            % 2 - tangent linear    ,
                            % 3 - adjoint mode      ,
                            % 4 - complex step
    comp_calc=    1;        % 1 for all variables,
                            % 2 for location      ,
                            % 3 for alfa
                            % 4 for x-location
    %
    % Boundary fitting parameters:
    %
    hstep    = 1.25e-1;     % initial step size for surface point gen.
    tspan    = [0.0, 200]; % tspan for the surface point gen.
    eta_tol  = 1e-3;        % tolerance for distance from zero-curve
    dxg      = 0.1;         % grid resolution to find boundaries
    dyg      = 0.1;
    x_tol    = 10^-10;      % tolerance for bisection
    intol    = dxg/4;       % tolerance for inpolygon
    mpts     = 10;          % number of points along each spline segment
    %
    % Optimization parmeters:
    %
    sd_cg    =    1;        % 0 for steepest,
                            % 1 for conj
    MAX_ITER =  9999;       % Maximum number of iterations
    ctol     = 1e-10;       % Tolerance on the norm of gradients
    dftol    = 1e-10;       % Tolerance on the change in objective
    ftol     = 1e-6;        % Tolerance on the value of objective
    %
    % Golden section search parameters:
    %
    delta    = 10^-3;       % Delta parameter for golden section search
    I_tol    = 10^-8;       % Interval of uncertainty tolerance
    %
    % Debugging parameter:
    %
    DEBUG    =     0;       % Debugging parameter
                            % 0 - Run as normal
                            % 1 - Output debugging
    %
    % The various test cases;
    %
    % nShape = 1 : Circle
    % nShape = 2 : Two ellipses side-by-side
    % nShape = 3 : Two ellipses diagonal side-by-side
    % nShape = 4 : NACA4420
    % nShape = 5 : Three NACA4420
    % nShape = 6 : Potato with varying concavities
    %
    % Define an array containing the test case numbers;
    %
    test_suite =[  1, ...
                   2, ...
                   3, ...
                   4, ...
                   5, ...
                   6, ...
                   7, ...
                   8, ...
                   9  ];
    %
    % Define an array containing the square grid dimension for the
    % stair-stepping algorithm;
```

```
%
igrid       =[ 25, ...
               60, ...
               25, ...
               25, ...
               25, ...
               60, ...
               25, ...
               25, ...
               50 ];
%
% Define an array of seed variables so the stair-stepped results are
% repeatable for debugging and analysis;
%
iseed       =[ 48919 , ... 48919
               111559, ... 111559
               100385, ... 100385
               73566 , ... 73566
               98918 , ... 98918
               75446 , ... 75446
               48919 , ... 48919
               48919 , ... 48919
               48919   ];
init_geo    = test_suite(icase);
des_geo     = test_suite(jgeom);
%
% Set the seed and grid variables for the current test case;
%
nseed = iseed(icase);
ngrid = igrid(icase);
%
% Set the number of tries for the stair-stepper and number of smoothing
% passes for the linear smoother;
%
NTRY  = 1;
NPASS = 2;
%
% Store the current directory for moving into the case directory and
% back into the current directory at the end of the simulation;
%
currd       = cd;
%% Set up solution directory
%
% Create the directories for the case files and data:
%
[ case_dir , ...
  opt_dir  , ...
  gss_dir  , ...
  int_dir  , ...
  fin_dir    ...
          ] = SetupSolutionDirectory( icase    , ...
                                      jgeom    , ...
                                      poly_fit , ...
                                      case_num );
if(DEBUG == 0 || DEBUG == 1)
    [ SUCCESS, ...
      MESSAGE, ...
      MSSGEID  ...
            ] = mkdir(case_dir);

    if(~isempty(MESSAGE))
        fprintf('%s\n', MESSAGE);
        rm_dir = input(' To remove directory and continue enter 1.\n Otherwise enter 0.\
            n');
        if(rm_dir == 0)
            return
        elseif(rm_dir == 1)
            [ SUCCESS, ...
```

```
                MESSAGE,  ...
                MSSGEID   ...
                        ] = rmdir(case_dir, 's');
        elseif(rm_dir ~= 1 || rm_dir ~= 0)
            error('Invalid input.');
        end %if
    end %if

    mkdir(sprintf('./%s/', opt_dir));
    mkdir(sprintf('./%s/', gss_dir));
    mkdir(sprintf('./%s/', int_dir));
    mkdir(sprintf('./%s/', fin_dir));
end %if
%% Generate the desired geometry and x-rays
%
% Define the desired body/bodies for the generation of the desired
% x-rays for matching;
%
[ xyPoly_desired, ...
  x_geo         , ...
  y_geo         , ...
  xyBOX          ] = desired_geometry( des_geo );
%
% Define the domain for the x-ray sweeps. Use twice the bounding box of
% the desired bodies to ensure the level-set body is captured;
%
hmax          = xyBOX(1) * 1.50;
hmin          = xyBOX(2) * 1.50;
vmax          = xyBOX(3) * 1.50;
vmin          = xyBOX(4) * 1.50;
%
% Generate the desired x-rays;
%
xrays_desired = GetXray( xyPoly_desired  , ...
                         []              , ...
                         []              , ...
                         1               , ... % Using linear interp
                         nrays           , ...
                         hmin            , ...
                         hmax            , ...
                         vmin            , ...
                         vmax             );

if(DEBUG == 1 || DEBUG == 0)
    figure;
    plot(x_geo, y_geo, 'k-')
    grid on
    axis image
    axis([hmin hmax vmin vmax])
    saveas(gcf, sprintf('./%s/desired_geometry.png', int_dir))
%
% Evaluate the objective function using the desired x-rays as a check
% that the objective is being calculated properly;
%
    EvaluateObjective(xyPoly_desired, ...
                      []            , ...
                      []            , ...
                      1             , ...
                      xrays_desired , ...
                      NRML          , ...
                      nrays         , ...
                      hmin          , ...
                      hmax          , ...
                      vmin          , ...
                      vmax          , ...
                      DEBUG          );
end %if DEBUG
%% Initial geometry generation
```

```
%
% Obtain an initial guess for the body/bodies using either rectangles
% or ellipses.
%
% Define the desired body/bodies for the generation of the desired
% x-rays for matching;
%
[ xyPoly_initial, ...
  x_geo         , ...
  y_geo         , ...
  ~              ] = desired_geometry( init_geo );
int_fig = [];
if(DEBUG == 1 || DEBUG == 0)
    figure(2392);
    plot(x_geo, y_geo, 'k-')
    grid on
    axis image
    axis([hmin hmax vmin vmax])
    saveas(gcf, sprintf('./%s/initial_geometry.png', int_dir))
%
% Evaluate the objective function using the desired x-rays as a check
% that the objective is being calculated properly;
%
    EvaluateObjective(xyPoly_initial, ...
                      []            , ...
                      []            , ...
                      1             , ...
                      xrays_desired , ...
                      NRML          , ...
                      nrays         , ...
                      hmin          , ...
                      hmax          , ...
                      vmin          , ...
                      vmax          , ...
                      1              );
end %if DEBUG
xrays_initial = GetXray( xyPoly_initial  , ...
                         []              , ...
                         []              , ...
                         1               , ... % Using linear interp
                         nrays           , ...
                         hmin            , ...
                         hmax            , ...
                         vmin            , ...
                         vmax             );
hray = zeros(nrays,2);
vray = zeros(nrays,2);
for iray = 1 : 2*(nrays+1)
    if(iray <= nrays+1)
        hray(iray,1) = xrays_initial(2*iray-1);
        hray(iray,2) = xrays_initial(2*iray  );
    elseif(iray >= nrays+2)
        jray = iray - (nrays+1);
        vray(jray,1) = xrays_initial(2*iray-1);
        vray(jray,2) = xrays_initial(2*iray  );
    end %if
end %for iray

if(DEBUG == 0)
    figure(6191)
    plot(hray(:,2), hray(:,1))
    grid on
    axis([0 max(hray(:,2))*1.1 vmin vmax])
    axis square
    xlabel('width')
    ylabel('y')
    set(gca, 'FontSize', 15)
    figure(6192)
    plot(vray(:,1), vray(:,2))
    grid on
    axis([hmin hmax 0 max(vray(:,2))*1.1])
    axis square
    xlabel('x')
    ylabel('height')
    set(gca, 'FontSize', 15)
    saveas(figure(6191), sprintf('./%s/horizontal_xrays.png'          , int_dir))
    saveas(figure(6192), sprintf('./%s/vertical_xrays.png'            , int_dir))
end %if

if(rect_elli ~= 2)
%
% Separate the rays into vertical and horizontal
%
    hray = zeros(nrays,2);
    vray = zeros(nrays,2);
    for iray = 1 : 2*(nrays+1)
        if(iray <= nrays+1)
            hray(iray,1) = xrays_initial(2*iray-1);
            hray(iray,2) = xrays_initial(2*iray  );
        elseif(iray >= nrays+2)
            jray = iray - (nrays+1);
            vray(jray,1) = xrays_initial(2*iray-1);
            vray(jray,2) = xrays_initial(2*iray  );
        end %if
    end %for iray
%
% Determine how many bodies are in initial guess;
%
    hbdy = 0;
    vbdy = 0;
    for iray = 1 : nrays
        ip1ray = iray + 1;
        if((hray(iray,2) ~= 0) && (hray(iray,2)*hray(ip1ray,2) == 0))
            hbdy = hbdy + 1;
        end %if
        if((vray(iray,2) ~= 0) && (vray(iray,2)*vray(ip1ray,2) == 0))
            vbdy = vbdy + 1;
        end %if
    end %for iray
    if(hbdy ~= vbdy)
        if(hbdy > vbdy)
            nbdy = hbdy;
        elseif(hbdy < vbdy)
            nbdy = vbdy;
        end %if
    elseif(hbdy == vbdy)
        nbdy = hbdy;
    end %if
%
% Define arrays for major and minor axes to define the rectangle(s) or
% ellipse(s):
%
    h_ax = zeros(nbdy,1);
    v_ax = zeros(nbdy,1);
%
% Loop through both hrays and vrays to determine the major and minor
% axes for each body:
%
    icross = 0;
    htemp_ax= zeros(2*nbdy,1);
    for iray = 1 : nrays
        ip1ray = iray + 1;
        if((hray(ip1ray,2) ~= 0) && (hray(iray,2)*hray(ip1ray,2) == 0))
            icross          = icross + 1;
            htemp_ax(icross) = hray(ip1ray,1);
        elseif(hray(iray,2) ~= 0) && (hray(iray,2)*hray(ip1ray,2) == 0)
            icross          = icross + 1;
            htemp_ax(icross) = hray(iray,1);
        end %if
    end %for iray
    for ibdy = 1 : hbdy
        h_ax(ibdy) = htemp_ax(2*ibdy) - htemp_ax(2*ibdy-1);
    end %if
    if(hbdy < nbdy)
        for ibdy = hbdy+1 : nbdy
            h_ax(ibdy) = h_ax(1);
            htemp_ax(2*ibdy-1) = htemp_ax(1);
            htemp_ax(2*ibdy  ) = htemp_ax(2);
        end %for ibdy
    end %if
    icross = 0;
    vtemp_ax= zeros(2*nbdy,1);
    for iray = 1 : nrays
        ip1ray = iray + 1;
        if((vray(ip1ray,2) ~= 0) && (vray(iray,2)*vray(ip1ray,2) == 0))
            icross          = icross + 1;
            vtemp_ax(icross) = vray(ip1ray,1);
        elseif(vray(iray,2) ~= 0) && (vray(iray,2)*vray(ip1ray,2) == 0)
            icross          = icross + 1;
            vtemp_ax(icross) = vray(iray,1);
        end %if
    end %for iray
    for ibdy = 1 : vbdy
        v_ax(ibdy) = vtemp_ax(2*ibdy) - vtemp_ax(2*ibdy-1);
    end %if
    if(vbdy < nbdy)
        for ibdy = vbdy+1 : nbdy
            v_ax(ibdy) = v_ax(1);
            vtemp_ax(2*ibdy-1) = vtemp_ax(1);
            vtemp_ax(2*ibdy  ) = vtemp_ax(2);
        end %for ibdy
    end %if
%
% Generate the rectangle(s) or ellipse(s)
%
    if(rect_elli == 0) % rectangular initial guess
        dxg = zeros(nbdy,1);
        dyg = zeros(nbdy,1);
        x_rect = zeros((4*(ngrid-1)+1)*nbdy,1);
        y_rect = zeros((4*(ngrid-1)+1)*nbdy,1);
        for ibdy = 1 : nbdy
            dxg(ibdy) = h_ax(ibdy)/(ngrid-1);
            dyg(ibdy) = v_ax(ibdy)/(ngrid-1);
            ipnt      = 1 + (4*(ngrid-1)+1)*(ibdy-1);
            % Bottom
            x_rect(ipnt+0*(ngrid-1) : ipnt+1*(ngrid-1)-1) = ...
                              linspace(vtemp_ax(2*ibdy           ), ...
                                       vtemp_ax(2*ibdy-1)+dxg(ibdy), ...
                                       ngrid-1                     );
            y_rect(ipnt+0*(ngrid-1) : ipnt+1*(ngrid-1)-1) = ...
                              ones(ngrid-1,1)*htemp_ax(2*ibdy  );

            % Right
            x_rect(ipnt+1*(ngrid-1) : ipnt+2*(ngrid-1)-1) = ...
                              ones(ngrid-1,1)*vtemp_ax(2*ibdy-1);
            y_rect(ipnt+1*(ngrid-1) : ipnt+2*(ngrid-1)-1) = ...
                              linspace(htemp_ax(2*ibdy  )          , ...
                                       htemp_ax(2*ibdy-1)+dyg(ibdy), ...
                                       ngrid-1                     );

            % Top
            x_rect(ipnt+2*(ngrid-1) : ipnt+3*(ngrid-1)-1) = ...
                              linspace(vtemp_ax(2*ibdy-1)          , ...
                                vtemp_ax(2*ibdy   )-dxg(ibdy), ...
                                ngrid-1                     );
            y_rect(ipnt+2*(ngrid-1) : ipnt+3*(ngrid-1)-1) = ...
                          ones(ngrid-1,1)*htemp_ax(2*ibdy-1);

            % Left
            x_rect(ipnt+3*(ngrid-1) : ipnt+4*(ngrid-1)-1) = ...
                          ones(ngrid-1,1)*vtemp_ax(2*ibdy   );
            y_rect(ipnt+3*(ngrid-1) : ipnt+4*(ngrid-1)-1) = ...
                          linspace(htemp_ax(2*ibdy-1) , ...
                          htemp_ax(2*ibdy   )-dyg(ibdy), ...
                          ngrid-1                     );
            x_rect(ipnt+4*(ngrid-1)) = NaN;
            y_rect(ipnt+4*(ngrid-1)) = NaN;
        end %for ibdy
        xy_ord = [x_rect, y_rect];
    elseif(rect_elli == 1) % elliptical initial guess
        n_elli = 100;
        x_elli = zeros((n_elli+1)*nbdy,1);
        y_elli = zeros((n_elli+1)*nbdy,1);
        t_elli = linspace(0, 2*pi, n_elli+1);
        for ibdy = 1 : nbdy
            ipnt = 1 + (n_elli+1) * (ibdy-1);
            length(t_elli(1:n_elli))
            x_elli(ipnt : (n_elli+1)*ibdy-1) = 0.5*v_ax(ibdy)*cos(t_elli(1:n_elli)) +
                0.5*(vtemp_ax(2*ibdy-1) + vtemp_ax(2*ibdy));
            y_elli(ipnt : (n_elli+1)*ibdy-1) = 0.5*h_ax(ibdy)*sin(t_elli(1:n_elli)) +
                0.5*(htemp_ax(2*ibdy-1) + htemp_ax(2*ibdy));
            x_elli((n_elli+1)*ibdy) = NaN;
            y_elli((n_elli+1)*ibdy) = NaN;
        end %for ibdy
        xy_ord = [x_elli, y_elli];
    end %if
elseif(rect_elli == 2)
    [xy_ss, ...
      npnt , ...
      nbdy  ] = stair_stepped_representation( NTRY          , ...
                                              xyPoly_initial, ...
                                              ngrid         , ...
                                              nseed          );
    xy_ord = zeros(npnt,2);
    for ipnt = 1 : npnt
        xy_ord(ipnt,1) = xy_ss(2*ipnt-1);
        xy_ord(ipnt,2) = xy_ss(2*ipnt   );
    end %for ipnt
    saveas(figure(3234), sprintf('./%s/discrete_horizontal_xrays1.png', int_dir))
    saveas(figure(4234), sprintf('./%s/discrete_vertical_xrays1.png'  , int_dir))
    saveas(figure(5234), sprintf('./%s/discrete_horizontal_xrays2.png', int_dir))
    saveas(figure(6234), sprintf('./%s/discrete_vertical_xrays2.png'  , int_dir))
end %if

if(DEBUG == 1 || DEBUG == 0)
    figure(2392)
    clf;
    plot(xy_ord(:,1), xy_ord(:,2), 'mo-', 'LineWidth', 1.0)
    axis image
    axis([hmin hmax vmin vmax])
    saveas(gcf, sprintf('./%s/boundary_approx.png', int_dir))
    pause(0.1)
end %if
%% Smooth the initial guess
%
% Smooth the stair-step using linear smoothing technique using NPASS to
% determine the number of smoothing passes;
%
if(rect_elli == 0)
    npnt  = length(x_rect);
    xy_ss = zeros(2*npnt,1);
    for ipnt = 1 : npnt
        xy_ss(2*ipnt-1) = x_rect(ipnt);
        xy_ss(2*ipnt  ) = y_rect(ipnt);
    end %for ipnt
    xy     = linear_smoothing(  xy_ss, ...
                                npnt , ...
                                nbdy , ...
                                NPASS );
elseif(rect_elli == 1)
    npnt  = length(x_elli);
    xy = [x_elli, y_elli];
elseif(rect_elli == 2)
    xy     = linear_smoothing(  xy_ss, ...
                                npnt , ...
                                nbdy , ...
                                NPASS );
end %if
%% Optimization iterations including reinitialization:
old_dir  = case_dir;
for reinit = 1 : 100
%
% Check if a reinitialized directory needs to be created
%
    if(reinit > 1)
        case_dir = sprintf('%s_reinit%03d'    , old_dir, reinit-1);
        opt_dir  = sprintf('%s/opt_iterations', case_dir);
        gss_dir  = sprintf('%s/gss_iterations', case_dir);
        int_dir  = sprintf('%s/initial'       , case_dir);
        fin_dir  = sprintf('%s/final'         , case_dir);

        fprintf(1,'%s\n',case_dir);
        fprintf(1,'%s\n',opt_dir);
        fprintf(1,'%s\n',gss_dir);
        fprintf(1,'%s\n',int_dir);
        fprintf(1,'%s\n',fin_dir);
        if(DEBUG == 0 || DEBUG == 1)
            SUCCESS = [];
            MESSAGE = [];
            MSSGEID = [];
            [ SUCCESS, ...
              MESSAGE, ...
              MSSGEID  ...
                      ] = mkdir(case_dir);

            if(~isempty(MESSAGE))
                if(rm_dir == 0)
                    return
                elseif(rm_dir == 1)
                    [ SUCCESS, ...
                      MESSAGE, ...
                      ~    ...
                              ] = rmdir(case_dir, 's');
                end %if
            end %if

            mkdir(sprintf('./%s/', opt_dir));
            mkdir(sprintf('./%s/', gss_dir));
            mkdir(sprintf('./%s/', int_dir));
            mkdir(sprintf('./%s/', fin_dir));
        end %if
    end %if
%
% Define the midpoint of the inside and outside RBFs at equally spaced
% points along the smoothed curve;
%
    xyRBFm = RBF_distributor( xy     , ...
                              nRBF   , ...
                              npnt     , ...
```

```
                                nbdy    );

    if(DEBUG == 1 || DEBUG == 0)
        figure(2392)
        if(reinit > 1)
            clf;
        end %if
        hold on
        plot(xy(:,1), xy(:,2), 'b-', 'LineWidth', 1.5)
        hold off
        axis image
        axis([hmin hmax vmin vmax])
        saveas(gcf, sprintf('./%s/boundary_smooth.png', int_dir))
        pause(0.1)

        figure(2392)
        hold on
        plot(xyRBFm(:,1), xyRBFm(:,2), 'ks', 'MarkerSize', 10, 'LineWidth', 1.5)
        hold off
        axis image
        axis([hmin hmax vmin vmax])
        saveas(gcf, sprintf('./%s/RBF_midpoints.png', int_dir))
        pause(0.1)

        figure(23921)
        if(reinit > 1)
            clf;
        end %if
        hold on
        plot(xyRBFm(:,1), xyRBFm(:,2), 'ks', 'MarkerSize', 5, 'LineWidth', 1.5)
        hold off
        axis image
        axis([hmin hmax vmin vmax])
        saveas(gcf, sprintf('./%s/RBF_midpoints1.png', int_dir))
        pause(0.1)
    end %if

    xRBF_mid = xyRBFm(:,1);
    yRBF_mid = xyRBFm(:,2);
%
% In case the RBFs are not spaced equally, calculate the maximum
% distance between RBFs along a curve for use as the support radius if
% the nSR is a value other than 0. This defines the support radius as
% the largest distant between RBFs on all boundaries in the domain, so
% the largest body dictates the support radius.
%
    max_dRBF = -99999;
    ibeg     = 1;
    for iRBF = 1 : nRBF * nbdy
        iRBFp1 = iRBF + 1;
        if(mod(iRBF, nRBF) == 0)
            iRBFp1 = ibeg;
            ibeg   = iRBF + 1;
        end %if
        dist = sqrt((xRBF_mid(iRBFp1) - xRBF_mid(iRBF))^2 + ...
                    (yRBF_mid(iRBFp1) - yRBF_mid(iRBF))^2);
        if(max_dRBF < dist)
            max_dRBF = dist;
        end %if
    end %for iRBF
%
% Set the support radius if nSR is defined:
%
    if(nSR ~= 0)
        mSR= nSR;
        SR = mSR * max_dRBF;
    elseif(nSR == 0)
        mSR=  SR / max_dRBF;
    end %if
%
% Generate the README.txt file
%
    WriteREADME( case_dir , ...
                 case_num , ...
                 icase    , ...
                 jgeom    , ...
                 rect_elli, ...
                 nseed    , ...
                 ngrid    , ...
                 NTRY     , ...
                 NPASS    , ...
                 nRBF     , ...
                 SR       , ...
                 mSR      , ...
                 fSR      , ...
                 OFFSET   , ...
                 inout    , ...
                 dxg      , ...
                 dyg      , ...
                 hstep    , ...
                 tspan    , ...
                 eta_tol  , ...
                 x_tol    , ...
                 intol    , ...
                 mpts     , ...
                 nrays    , ...
                 NRML     , ...
                 poly_fit , ...
                 hmin     , ...
                 hmax     , ...
                 vmin     , ...
                 vmax     , ...
                 sens_calc, ...
                 comp_calc, ...
                 sd_cg    , ...
                 ctol     , ...
                 dftol    , ...
                 ftol     , ...
                 MAX_ITER , ...
                 delta    , ...
                 I_tol    , ...
                 DEBUG     );
%% Generate RBF locations and coefficients
%
% Distribute the RBFs and solve for coefficients based on OFFSET
%
    [ xRBF, ...
      yRBF, ...
      aRBF, ...
      mRBF ] = RBF_parameters( xRBF_mid, ...
                               yRBF_mid, ...
                               SR      , ...
                               fSR     , ...
                               nRBF    , ...
                               nbdy    , ...
                               OFFSET  , ...
                               inout   , ...
                               int_fig , ...
                               int_dir , ...
                               DEBUG    );

    LSF  = GenerateLSF(  xRBF  , ...
                         yRBF  , ...
                         hmax  , ...
                         hmin  , ...
                         vmax  , ...
```

```
                             vmin   , ...
                             aRBF   , ...
                             SR     , ...
                             OFFSET );
        delx = 0.1;
        dely = 0.1;
        xLSF = hmin : delx : hmax;
        yLSF = vmin : dely : vmax;
        [xmesh, ymesh] = meshgrid(xLSF, yLSF);

        if(DEBUG == 1 || DEBUG == 0)
            figure(2392)
            clf;
            set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
            contourf(xmesh, ymesh, LSF, [0 0])
            axis image
            axis([hmin hmax vmin vmax])
            set(gca, 'FontSize', 20)
%             title(sprintf('Zero level-set curve'))
            xlabel('x')
            ylabel('y')
            saveas(gcf, sprintf('./%s/LSF_representation.png', int_dir))
        end %if DEBUG

        tic

        if(DEBUG == 1)
            fprintf(1, 'Number of bodies: %2d\n', nbdy);
        end %if
        xg     = hmin : dxg : hmax;
        yg     = vmin : dyg : vmax;
    %
    % Solve for the spline fit for the boundary(ies)
    %
        xy_spln = [];
        xy_tpar = [];
        xy_curv = [];
        [ xy_spln, ...
          xy_tpar, ...
          xy_curv, ...
          xy_topo, ...
          nbdy      ...
                 ] = LevelSetSpline( xg       , ...
                                     yg       , ...
                                     xRBF     , ...
                                     yRBF     , ...
                                     aRBF     , ...
                                     SR       , ...
                                     OFFSET   , ...
                                     x_tol    , ...
                                     intol    , ...
                                     tspan    , ...
                                     hstep    , ...
                                     eta_tol  , ...
                                     poly_fit , ...
                                     mpts     , ...
                                     hmin     , ...
                                     hmax     , ...
                                     vmin     , ...
                                     vmax     , ...
                                     DEBUG     );

        toc
            if(poly_fit == 1 || poly_fit == 3)
                npts       = length(xy_spln(:,1));
                xySpln = zeros(2*(npts),1);
                for i = 1 : npts
                    xySpln(2*i-1) = xy_spln(i,1);
                xySpln(2*i   ) = xy_spln(i,2);
            end %for i
        elseif(poly_fit == 2)
            npts       = length(xy_spln(:,1));
            xySpln = zeros(2*(npts),1);
            for i = 1 : npts
                xySpln(2*i-1) = xy_spln(i,1);
                xySpln(2*i   ) = xy_spln(i,2);
            end %for i

            npts       = length(xy_topo(:,1));
            xyPoly = zeros(2*(npts),1);
            for i = 1 : npts
                xyPoly(2*i-1) = xy_topo(i,1);
                xyPoly(2*i   ) = xy_topo(i,2);
            end %for i
        end %if
        xyCurv = xy_curv;
        xyTpar = xy_tpar;
%
% Evaluate the objective function based on fit:
% -> poly_fit == 1 - a linear spline
% -> poly_fit == 2 - a linear spline approximation of the cubic
% -> poly_fit == 3 - a cubic spline (inaccurate calcs, currently)
%
        if(poly_fit == 1 || poly_fit == 3)

            ObjFunc  = EvaluateObjective(  xySpln    , ...
                                           xyCurv    , ...
                                           xyTpar    , ...
                                           poly_fit      , ...
                                           xrays_desired , ...
                                           NRML          , ...
                                           nrays         , ...
                                           hmin          , ...
                                           hmax          , ...
                                           vmin          , ...
                                           vmax          , ...
                                           1              );
        elseif(poly_fit == 2)

            ObjFunc  = EvaluateObjective(  xyPoly    , ...
                                           xyCurv    , ...
                                           xyTpar    , ...
                                           poly_fit      , ...
                                           xrays_desired , ...
                                           NRML          , ...
                                           nrays         , ...
                                           hmin          , ...
                                           hmax          , ...
                                           vmin          , ...
                                           vmax          , ...
                                           1             );
        end %if

    if(DEBUG == 1 || DEBUG == 0)
        if(DEBUG == 1)
            fprintf(1, '   \n');
            fprintf(1, '[ xySpln ]\n');
            for ipnt = 1 : length(xySpln)
            fprintf(1, '[ %+5.2f ]\n', xySpln(ipnt));
            end %for ipnt
            fprintf(1, '   \n');
            fprintf(1, '[ xyCurv ]\n');
            for ipnt = 1 : length(xyCurv)
            fprintf(1, '[ %+5.2f ]\n', xyCurv(ipnt));
            end %for ipnt
            fprintf(1, '   \n');
            fprintf(1, '[ xyTpar ]\n');
            for ipnt = 1 : length(xyTpar)
            fprintf(1, '[ %+5.2f ]\n', xyTpar(ipnt));
            end %for ipnt
        end %if

        LSF    = GenerateLSF(  xRBF, ...
                               yRBF, ...
                               hmax, ...
                               hmin, ...
                               vmax, ...
                               vmin, ...
                               aRBF, ...
                               SR   , ...
                               OFFSET    );

        figure(45322)
        subplot(2,2,2)
        contourf(xmesh, ymesh, LSF, 'LineStyle', 'none')
        hold on
        contour(xmesh, ymesh, LSF, [0 0], 'k--', 'LineWidth', 1.5)
        for iRBF = 1 : length(xRBF)

            LSFchck= EvaluateLSF(  xRBF(iRBF), ...
                                   yRBF(iRBF), ...
                                   xRBF      , ...
                                   yRBF      , ...
                                   aRBF      , ...
                                   SR            , ...
                                   OFFSET         );
            if(LSFchck > 0) %inside
                plot(xRBF(iRBF), yRBF(iRBF), 'bp')
            elseif(LSFchck < 0) %outside
                plot(xRBF(iRBF), yRBF(iRBF), 'rp')
            else
                plot(xRBF(iRBF), yRBF(iRBF), 'kp')
            end %if

        end %for iRBF

        plot(xy_spln(:,1), xy_spln(:,2), 'bo');
        hold off
        xlabel('x')
        ylabel('y')
        title( [{sprintf('Level-set function')}, ...
                {sprintf(' iteration: %4d', 0)}] )
        axis image
        axis([hmin hmax vmin vmax])
        colorbar
        caxis([-1 1])
        set(gca, 'FontSize', 15)
        pause(0.01)
        saveas(gcf, sprintf('./%s/iter_%05d.png', int_dir,    0));


        figure(23921)
        hold on
        for iRBF = 1 : length(xRBF)

            LSFchck= EvaluateLSF(  xRBF(iRBF), ...
                                   yRBF(iRBF), ...
                                   xRBF      , ...
                                   yRBF      , ...
                                   aRBF      , ...
                                   SR            , ...
                                   OFFSET         );
            if(LSFchck > 0) %inside
                plot(xRBF(iRBF), yRBF(iRBF), 'bp')
```

```
            elseif(LSFchck < 0) %outside
                plot(xRBF(iRBF), yRBF(iRBF), 'rp')
            else
                plot(xRBF(iRBF), yRBF(iRBF), 'kp')
            end %if

        end %for iRBF
        hold off

        saveas(gcf, sprintf('./%s/RBF_midpnt_dist.png', int_dir));
    end %if

    if(DEBUG == 1)
        fprintf('\n')
        fprintf('Objective function: %+8.6f\n', ObjFunc);
    end %if

    if(DEBUG == 1 || DEBUG == 0)
        x_c  = xy_topo(:,1);
        y_c  = xy_topo(:,2);

        kpts = length(xySpln)/2;
        x_k  = zeros(kpts,1);
        y_k  = zeros(kpts,1);
        for ik = 1 : kpts
            x_k(ik) = xySpln(2*ik-1);
            y_k(ik) = xySpln(2*ik  );
        end %for ik

        figure(718)
        clf;
        set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
        contourf(xmesh, ymesh, LSF, [0 0])
        axis image
        axis([hmin hmax vmin vmax])
        set(gca, 'FontSize', 20)
%           title(sprintf('Zero level-set curve'))
        xlabel('x')
        ylabel('y')

        gcf;
        hold on
        plot(x_k       , y_k       , 'bo' , 'MarkerSize',  8, 'LineWidth', 1.5)
        plot(x_c       , y_c       , 'r--',                   'LineWidth', 1.5)
        w1 = plot(NaN,NaN, 'bo-');
        w2 = plot(NaN,NaN, 'r--');
        hold off
        axis([-1 1 -1 1])

        gcf;
        hold on
        if(reinit == 1)
            plot(xy_ord(:,1), xy_ord(:,2), 'mo:', 'MarkerSize', 8)
        end %if
        for iRBF = 1 : length(xRBF)

            LSFchck= EvaluateLSF(  xRBF(iRBF), ...
                                   yRBF(iRBF), ...
                                   xRBF      , ...
                                   yRBF      , ...
                                   aRBF      , ...
                                   SR            , ...
                                   OFFSET           );
            if(LSFchck > 0) %inside
                plot(xRBF(iRBF), yRBF(iRBF), 'bp' , 'MarkerSize', 8)
            elseif(LSFchck < 0) %outside
                plot(xRBF(iRBF), yRBF(iRBF), 'rp' , 'MarkerSize', 8)
            else
```

```
                plot(xRBF(iRBF), yRBF(iRBF), 'kp' , 'MarkerSize', 8)
            end %if

        end %for iRBF
        w5 = plot(NaN, NaN, 'bp');
        w6 = plot(NaN, NaN, 'rp');
        hold off
        axis([hmin hmax vmin vmax])
        if(reinit == 1)
            hold on
            w4 = plot(NaN, NaN, 'mo');
            hold off
            legend([w1,w2, w4, w5, w6], 'RK4 points' , ...
                                         'Cubic fit'  , ...
                                         'Stair-Step' , ...
                                         'Inside RBF' , ...
                                         'Outside RBF', ...
                                         'location'   , 'NorthEastOutside')
        elseif(reinit > 1)
            legend([w1,w2, w5, w6], 'RK4 points' , ...
                                         'Cubic fit'  , ...
                                         'Inside RBF' , ...
                                         'Outside RBF', ...
                                         'location'   , 'NorthEastOutside')
        end %if
        saveas(gcf, sprintf('./%s/initial_rep.png', int_dir))
    end %if DEBUG
%
%======================== END OF INITIAL SETUP ========================%
%
    fclose all;
    return
%% Optimizer
    [ ObjFunc    , ...
      xySpln_opt ] = levelset_topology_optimizer( ObjFunc   , ...
                                                  xrays_desired, ...
                                                  xRBF      , ...
                                                  yRBF      , ...
                                                  aRBF      , ...
                                                  mRBF      , ...
                                                  nbdy      , ...
                                                  SR        , ...
                                                  OFFSET    , ...
                                                  poly_fit  , ...
                                                  mpts      , ...
                                                  xg        , ...
                                                  yg        , ...
                                                  x_tol     , ...
                                                  intol     , ...
                                                  tspan     , ...
                                                  hstep     , ...
                                                  eta_tol   , ...
                                                  xySpln    , ...
                                                  xyCurv    , ...
                                                  xyTpar    , ...
                                                  nrays     , ...
                                                  NRML      , ...
                                                  hmax      , ...
                                                  hmin      , ...
                                                  vmax      , ...
                                                  vmin      , ...
                                                  sens_calc , ...
                                                  comp_calc , ...
                                                  sd_cg     , ...
                                                  ctol      , ...
                                                  dftol     , ...
                                                  ftol      , ...
                                                  MAX_ITER  , ...
```

```
                                                                  delta     , ...
                                                                  I_tol     , ...
                                                                  xmesh     , ...
                                                                  ymesh     , ...
                                                                  case_dir  , ...
                                                                  opt_dir   , ...
                                                                  gss_dir   , ...
                                                                  DEBUG      );

        cd(currd);
    %
    % Exit if objective is less than ftol
    %
        if(ObjFunc < ftol)
            npnt  = length(xySpln_opt)/2;
            nbdy  = 0;
            xy    = zeros(npnt,2);
            for ipnt = 1 : npnt
                if(isnan(xySpln_opt(2*ipnt-1)))
                    nbdy = nbdy + 1;
                end %if
                xy(ipnt,1) = xySpln_opt(2*ipnt-1);
                xy(ipnt,2) = xySpln_opt(2*ipnt  );
            end %for ipnt
            break;
        end %if
    %
    % Determine the reinitialization step required based on new objective
    % value
    %
        if(reinit == 1)
            f_old = ObjFunc;
            npnt  = length(xySpln_opt)/2;
            nbdy  = 0;
            xy    = zeros(npnt,2);
            for ipnt = 1 : npnt
                if(isnan(xySpln_opt(2*ipnt-1)))
                    nbdy = nbdy + 1;
                end %if
                xy(ipnt,1) = xySpln_opt(2*ipnt-1);
                xy(ipnt,2) = xySpln_opt(2*ipnt  );
            end %for ipnt
        elseif(reinit > 1)
            if(ObjFunc > f_old)
                temp = nRBF+1;
                nRBF = temp;
                nbdy  = 0;
                for ipnt = 1 : length(xy(:,1))
                    if(isnan(xy(ipnt,1)))
                        nbdy = nbdy + 1;
                    end %if
                end %for ipnt
            else
                f_old = ObjFunc;
                npnt  = length(xySpln_opt)/2;
                nbdy  = 0;
                xy    = zeros(npnt,2);
                for ipnt = 1 : npnt
                    if(isnan(xySpln_opt(2*ipnt-1)))
                        nbdy = nbdy + 1;
                    end %if
                    xy(ipnt,1) = xySpln_opt(2*ipnt-1);
                    xy(ipnt,2) = xySpln_opt(2*ipnt  );
                end %for ipnt
            end %if
        end %if
    end %for reinit
end
```

```
%function levelset_topology_optimizationMar07
%-------------------------------------------------------------------
```

```
function [ObjFunc    , ...
          xySpln_opt ] = levelset_topology_optimizer( ObjFunc   , ...
                                                       xrays_des , ...
                                                       xRBF      , ...
                                                       yRBF      , ...
                                                       aRBF      , ...
                                                       mRBF      , ...
                                                       nbdy      , ...
                                                       SR        , ...
                                                       OFFSET    , ...
                                                       poly_fit  , ...
                                                       mpts      , ...
                                                       xg        , ...
                                                       yg        , ...
                                                       x_tol     , ...
                                                       intol     , ...
                                                       tspan     , ...
                                                       hstep     , ...
                                                       eta_tol   , ...
                                                       xySpln    , ...
                                                       xyCurv    , ...
                                                       xyTpar    , ...
                                                       nrays     , ...
                                                       NRML      , ...
                                                       hmax      , ...
                                                       hmin      , ...
                                                       vmax      , ...
                                                       vmin      , ...
                                                       sens_calc , ...
                                                       comp_calc , ...
                                                       sd_cg     , ...
                                                       ctol      , ...
                                                       dftol     , ...
                                                       ftol      , ...
                                                       MAX_ITER  , ...
                                                       delta     , ...
                                                       I_tol     , ...
                                                       xmesh     , ...
                                                       ymesh     , ...
                                                       case_dir  , ...
                                                       opt_dir   , ...
                                                       gss_dir   , ...
                                                       DEBUG      )
    % function levelset_topology_optimizer( ) drives the optimization
    % portion of the scheme using either conjugate gradient or steepest
    % desecnt techniques based on the input into the function.
    %
    % Inputs:
    %                  ObjFunc     -> Initial objective function
    %                  xrays_des   -> Desired x-rays for objective calcs
    %                  xRBF        -> Inital x-coord.    of RBFs
    %                  yRBF        -> Inital y-coord.    of RBFs
    %                  aRBF        -> Inital coefficient of RBFs
    %                  mRBF        -> Total number of RBFs
    %                  nbdy        -> Total number of bodies
    %                  SR          -> Support radius
    %                  OFFSET      -> Offset for LSF calcs
    %                  poly_fit    -> Fit type
    %                  mpts        -> number of points along spline
    %                  xg          -> Array of x-values for zero-point
    %                                 identification on one or more
    %                                 level-set curves
    %                  yg          -> Array of y-values for zero-point
    %                                 identification on one or more
    %                                 level-set curves
    %                  x_tol       -> Tolerance for the difference between
    %                                 free variables in the zero-point
    %                                 identification algorithm
%                   intol       -> Tolerance for whether a point is
%                                  inside/outside an existing boundary
%                                  in the zero-point identification
%                                  algorithm
%                   tspan       -> Range of parametric coordinate for
%                                  the level-set RK4 algorithm
%                   hstep       -> Initial step size for the level-set
%                                  RK4 algorithm
%                   eta_tol     -> Tolerance used to test whether a
%                                  step taken by the RK4 algorithm is
%                                  within an acceptable distance
%                   xySpln      -> Initial spline xy-coordinates
%                   xyCurv      -> Initial curvature data
%                   xyTpar      -> Initial parametric coordinate data
%                   nrays       -> Number of rays used for objective
%                                  calculations
%                   NRML        -> Whether the objective function
%                                  should be normalized or not
%                   hmax        -> Maximum y-coordinate of the
%                                  horizontal x-rays
%                   hmin        -> Minimum y-coordinate of the
%                                  horizontal x-rays
%                   vmax        -> Maximum x-coordinate of the
%                                  vertical x-rays
%                   vmin        -> Minimum x-coordinate of the
%                                  vertical x-rays
%                   sens_calc   -> Indicator for type of derivative
%                                  calculations:
%                                  1 - finite difference;
%                                  2 - tangent linear;
%                                  3 - complex step;
%                                  4 - adjoint mode
%                   comp_calc   -> Indicator for number of design
%                                  variables:
%                                  1 - RBF locations and coefficients;
%                                  2 - RBF locations;
%                                  3 - RBF coefficients
%                   sd_cg       -> Indicator for steepest descent or
%                                  conjugate gradient
%                   ctol        -> Tolerance for difference between
%                                  gradients from one optimization
%                                  iteration to the next
%                   dftol       -> Tolerance for difference between
%                                  objective function values from one
%                   ftol        -> Tolerance for value of the objective
%                                  function
%                                  optimization iteration to the next
%                   MAX_ITER    -> Maximum optimization iterations
%                   delta       -> Value used for golden section search
%                                  algorithm to initialize 1D search
%                                  parameter, alpha
%                   I_tol       -> Tolerance for interval of
%                                  uncertainty in golden section search
%                                  algorithm
%                   xmesh       -> x-coordinates for LSF plotting
%                   ymesh       -> y-coordinates for LSF plotting
%                   case_dir    -> case directory for saving files
%                   opt_dir     -> optimization directory for saving
%                                  files at each optimization iteration
%                   gss_dir     -> golden section search directory for
%                                  saving files at each search
%                                  iteration
%                   DEBUG       -> Indicator for debugging the code:
%                                  0 - run as normal;
%                                  1 - debugging output to screen;
%
% Outputs:
%                   ObjFunc     -> Optimized objective function at the
%                                  end of the optimization run
%                   xySpln_opt  -> Optimized spline coordinates at the
%                                  end of the optimization run
%
% Written by Jack Rossetti
% Annontated by Jack Rossetti_________________________________02/24/20
%
%==================== OPTIMIZATION SETUP AND RUN ====================%
%
% Calculate the sensitivities
%
if(DEBUG == 1)
    fprintf(1, 'Calculate sensitivities:\n');
end %if DEBUG
if(sens_calc == 1)
    if(DEBUG == 1)
        fprintf(1, ' ==> finite differences <==\n');
    end %if DEBUG
    FDsens=finite_diff_sensitivities(  xRBF      , ...
                                       yRBF      , ...
                                       aRBF      , ...
                                       SR        , ...
                                       nbdy      , ...
                                       poly_fit  , ...
                                       xySpln    , ...
                                       xyCurv    , ...
                                       xyTpar    , ...
                                       xrays_des , ...
                                       nrays     , ...
                                       NRML      , ...
                                       hmin      , ...
                                       hmax      , ...
                                       vmin      , ...
                                       vmax       );
    if(DEBUG == 1)
        for igrad = 1 : 3*mRBF
            fprintf(1, ' FD[%4d] = %+f\n', igrad, FDsens(igrad));
        end %for igrad
        fprintf(1, '\n');
    end %if DEBUG
    dOdp = FDsens;
elseif(sens_calc == 2)
    if(DEBUG == 1)
        fprintf(1, ' ==> tangent linear <==\n');
    end %if DEBUG
    TLsens=tangent_sensitivities( xRBF          , ...
                                  yRBF          , ...
                                  aRBF          , ...
                                  SR            , ...
                                  nbdy          , ...
                                  poly_fit      , ...
                                  xySpln        , ...
                                  xyCurv        , ...
                                  xyTpar        , ...
                                  xrays_des , ...
                                  nrays         , ...
                                  NRML          , ...
                                  hmin          , ...
                                  hmax          , ...
                                  vmin          , ...
                                  vmax           );
    if(DEBUG == 1)
        for igrad = 1 : 3*mRBF
            fprintf(1, ' TL[%4d] = %+f\n', igrad, TLsens(igrad) );
        end %for igrad
        fprintf(1, '\n');
    end %if DEBUG
    dOdp = TLsens;
    elseif(sens_calc == 3)
        if(DEBUG == 1)
            fprintf(1, ' ==> adjoint method <==\n');
        end %if DEBUG
        AMsens= adjoint_sensitivities(  xRBF          , ...
                                        yRBF          , ...
                                        aRBF          , ...
                                        SR            , ...
                                        nbdy          , ...
                                        poly_fit      , ...
                                        xySpln        , ...
                                        xyCurv        , ...
                                        xyTpar        , ...
                                        xrays_des , ...
                                        nrays         , ...
                                        NRML          , ...
                                        hmin          , ...
                                        hmax          , ...
                                        vmin          , ...
                                        vmax           );
        if(DEBUG == 1)
            for igrad = 1 : 3*mRBF
                fprintf(1, ' AM[%4d] = %+f\n', igrad, AMsens(igrad));
            end %for igrad
            fprintf(1, '\n');
        end %if DEBUG
        dOdp = AMsens;
    elseif(sens_calc == 4)
        if(DEBUG == 1)
            fprintf(1, ' ==>     complex step     <==\n');
        end %if DEBUG
        CSsens = complexstep_sensitivities( xRBF          , ...
                                            yRBF          , ...
                                            aRBF          , ...
                                            SR            , ...
                                            nbdy          , ...
                                            poly_fit      , ...
                                            mpts          , ...
                                            xySpln        , ...
                                            xrays_des , ...
                                            nrays         , ...
                                            NRML          , ...
                                            hmin          , ...
                                            hmax          , ...
                                            vmin          , ...
                                            vmax          , ...
                                            comp_calc      );
        if(DEBUG == 1)
            for igrad = 1 : 3*mRBF
                fprintf(1, ' CS[%4d] = %+f\n', igrad, CSsens(igrad));
            end %for igrad
            fprintf(1, '\n');
        end %if DEBUG
        dOdp = CSsens;
    end %if
    %
    % Define initial optimization variables
    %
    cold = dOdp;
    dold =-cold;

    xRBF_old      = xRBF;
    yRBF_old      = yRBF;
    aRBF_old      = aRBF;
%
    % Write iteration data:
    %
        npts = length(xySpln)/2;
```

```
if(DEBUG == 0)
    nvar = 3;
    dim  = 2;

    desparam = zeros(3*mRBF,1);
    for i = 1 : mRBF
        desparam(3*i-2) = xRBF_old(i);
        desparam(3*i-1) = yRBF_old(i);
        desparam(3*i-0) = aRBF_old(i);
    end %for i

    if(comp_calc == 1)
        gradient = zeros(3*mRBF,1);
        for i = 1 : mRBF
            gradient(3*i-2) = cold(i + 0*mRBF);
            gradient(3*i-1) = cold(i + 1*mRBF);
            gradient(3*i-0) = cold(i + 2*mRBF);
        end %for i

        conj_gradient = zeros(3*mRBF,1);
        for i = 1 : mRBF
            conj_gradient(3*i-2) = dold(i + 0*mRBF);
            conj_gradient(3*i-1) = dold(i + 1*mRBF);
            conj_gradient(3*i-0) = dold(i + 2*mRBF);
        end %for i
    elseif(comp_calc == 2)
        gradient = zeros(2*mRBF,1);
        for i = 1 : mRBF
            gradient(2*i-1) = cold(i + 0*mRBF);
            gradient(2*i  ) = cold(i + 1*mRBF);
        end %for i

        conj_gradient = zeros(2*mRBF,1);
        for i = 1 : mRBF
            conj_gradient(2*i-1) = dold(i + 0*mRBF);
            conj_gradient(2*i  ) = dold(i + 1*mRBF);
        end %for i
    elseif(comp_calc == 3)
        gradient = zeros(mRBF,1);
        for i = 1 : mRBF
            gradient(i) = cold(i);
        end %for i

        conj_gradient = zeros(mRBF,1);
        for i = 1 : mRBF
            conj_gradient(i) = dold(i);
        end %for i
    elseif(comp_calc == 4)
        gradient = zeros(mRBF,1);
        for i = 1 : mRBF
            gradient(i) = cold(i);
        end %for i

        conj_gradient = zeros(mRBF,1);
        for i = 1 : mRBF
            conj_gradient(i) = dold(i);
        end %for i
    end %if
    directory = sprintf('./%s/%s/', case_dir, 'initial');

    write_data(  mRBF          , ...
                 nvar          , ...
                 desparam      , ...
                 npts          , ...
                 dim           , ...
                 xySpln        , ...
                 comp_calc     , ...
                 gradient      , ...
```

```
                     conj_gradient , ...
                     ObjFunc       , ...
                     []            , ...
                     []            , ...
                     []            , ...
                     []            , ...
                     directory      );
    end %if DEBUG

ipng = 0;

timing = toc;
fpo    = 1;
fprintf(fpo,'%-5s %-7s %-7s %-10s %-10s %-10s\n', ...
                'iter', 'time(s)', 'dObjFunc', 'ObjFunc', 'beta', 'norm(dOdp)');
fprintf(fpo,'==================================================================\n');
fprintf(fpo,'%4d, %6.2f, %6.1e, %6.2e, %+9.4f, %+9.4f\n', ...
                0, timing, NaN, min(ObjFunc, 9999), NaN, NaN);
%
% Write output file for optimization iterations:
%
fname = sprintf('./%s/optimization_history.txt', case_dir);
fpo   = fopen(fname, 'w');
fprintf(fpo,'%-5s %-7s %-7s %-10s %-10s %-10s\n', ...
                'iter', 'time(s)', 'dObjFunc', 'ObjFunc', 'beta', 'norm(dOdp)');
fprintf(fpo,'==================================================================\n');
fprintf(fpo,'%4d, %6.2f, %6.1e, %6.2e, %+9.4f, %+9.4f\n', ...
                0, timing, NaN, min(ObjFunc, 9999), NaN, NaN);
fclose(fpo);
conv_hist    = zeros(MAX_ITER,1);
conv_hist(1) = ObjFunc;
%
% Run the optimizer
%
for opt_iter = 1 : MAX_ITER
    if(DEBUG == 1)
        if(comp_calc == 1)
            for igrad = 1 : 3*mRBF
                fprintf(1, ' dOdp[%4d] = %+f\n', igrad, dOdp(igrad));
            end %for igrad
            fprintf(1, '\n');
        elseif(comp_calc == 2)
            for igrad = 1 : 2*mRBF
                fprintf(1, ' dOdp[%4d] = %+f\n', igrad, dOdp(igrad));
            end %for igrad
            fprintf(1, '\n');
        elseif(comp_calc == 3)
            for igrad = 1 : 1*mRBF
                fprintf(1, ' dOdp[%4d] = %+f\n', igrad, dOdp(igrad));
            end %for igrad
            fprintf(1, '\n');
        elseif(comp_calc == 4)
            for igrad = 1 : 1*mRBF
                fprintf(1, ' dOdp[%4d] = %+f\n', igrad, dOdp(igrad));
            end %for igrad
            fprintf(1, '\n');
        end %if
    end %if DEBUG
% ==> Take a step in the conjugate gradient or steepest descent dir
%---- Begin conjugate gradient optimization ----%
% From Arora's Introduction to Optimum Design:
% -> Step 1: Estimate a starting design as x0. Set iteration counter
%            k = 0. Select the convergence parameter tol. Set search
%            direction as the negative gradient.
%            Check the convergence criteria.
    if(norm(cold) < ctol)
        xyRBF_opt = zeros(2*length(xRBF_new),1);
        for i = 1 : length(xRBF_new)
```

```
            xyRBF_opt(2*i-1) = xRBF_new(i);
            xyRBF_opt(2*i  ) = yRBF_new(i);
        end %for i
        aRBF_opt = aRBF_new;

        fpo   = 1;
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> norm(cold) criterion reached.\n');

        fname = sprintf('./%s/optimization_history.txt', case_dir);
        fpo   = fopen(fname, 'a');
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> norm(cold) criterion reached.\n');
        fclose(fpo);
        break;
    end %if
%
% -> Step 2: Compute the gradient of the cost function.
%
    cnew = dOdp;
%
% -> Step 3: Compute the norm of the gradient and check convergence.
%
    if(norm(cnew) < ctol)
        xyRBF_opt = zeros(2*length(xRBF_old),1);
        for i = 1 : length(xRBF_old)
            xyRBF_opt(2*i-1) = xRBF_old(i);
            xyRBF_opt(2*i  ) = yRBF_old(i);
        end %for i
        aRBF_opt = aRBF_old;

        fpo   = 1;
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> norm(cnew) criterion reached.\n');

        fname = sprintf('./%s/optimization_history.txt', case_dir);
        fpo   = fopen(fname, 'a');
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> norm(cnew) criterion reached.\n');
        fclose(fpo);
        break;
    end %if
%
% -> Step 4: Calculate the new conjugate direction as
%
    if(sd_cg == 1)
        ycnj    = cnew - cold;
        beta_fr = (cnew' * cnew)/(cold' * cold); % Fletcher-Reeves
        beta_pr = (cnew' * ycnj)/(cold' * cold); % Polak-Ribiere
        if(0 <= beta_pr && beta_pr <= beta_fr)
            beta = beta_pr;
        elseif(beta_pr > beta_fr)
            beta = beta_fr;
        elseif(beta_pr < 0)
            beta = 0;
        end %if
    end %if

    if(opt_iter > 1 && sd_cg == 1)
        dnew =-cnew + beta*dold;
    elseif(mod(opt_iter,mRBF) == 0 || opt_iter == 1 || sd_cg == 0)
        beta = 0.0;
        dnew =-cnew;
    end %if
%
% -> Step 5: Compute the step size alpha using GOLDEN SEARCH.
%
    [ alpha   , ...
```

```
      GSS_brak, ...
      GSS_iter ] = GoldenSectionSearch(   delta        , ...
                                          I_tol        , ...
                                          dnew         , ...
                                          xRBF_old     , ...
                                          yRBF_old     , ...
                                          aRBF_old     , ...
                                          SR           , ...
                                          OFFSET       , ...
                                          hmax         , ...
                                          hmin         , ...
                                          vmax         , ...
                                          vmin         , ...
                                          xmesh        , ...
                                          ymesh        , ...
                                          xg           , ...
                                          yg           , ...
                                          x_tol        , ...
                                          intol        , ...
                                          tspan        , ...
                                          hstep        , ...
                                          eta_tol      , ...
                                          poly_fit     , ...
                                          mpts         , ...
                                          xrays_des    , ...
                                          NRML         , ...
                                          nrays        , ...
                                          comp_calc    , ...
                                          ipng         , ...
                                          gss_dir      , ...
                                          DEBUG         );
    if(DEBUG == 1)
        fprintf(1, '\n');
        fprintf(1, ' alpha      = %+f\n', alpha);
        fprintf(1, '\n');
    end %if DEBUG
%
% -> Step 6a: Change the design as follows: set k = k+1, write data,
%             check convergence criteria/boundary upate criteria, and
%             choose to repeat procedure or exit and accept optimum.
%
    dxRBF = [];
    dyRBF = [];
    daRBF = [];
    if(comp_calc == 1)
        dstep = alpha*dnew;
        dxRBF = dstep(       1 : 1*mRBF);
        dyRBF = dstep(1*mRBF+1: 2*mRBF);
        daRBF = dstep(2*mRBF+1: 3*mRBF);
    elseif(comp_calc == 2)
        dstep = alpha*dnew;
        dxRBF = dstep(       1 : 1*mRBF);
        dyRBF = dstep(1*mRBF+1: 2*mRBF);
        daRBF = zeros(size(dstep(       1 : 1*mRBF)));
    elseif(comp_calc == 3)
        daRBF = alpha*dnew;
        dxRBF = zeros(size(daRBF));
        dyRBF = zeros(size(daRBF));
    elseif(comp_calc == 4)
        dxRBF = alpha*dnew;
        dyRBF = zeros(size(dxRBF));
        daRBF = zeros(size(dxRBF));
    end %if
    xRBF_new = xRBF_old + dxRBF;
    yRBF_new = yRBF_old + dyRBF;
    aRBF_new = aRBF_old + daRBF;

    if(DEBUG == 1)
        LSF  = GenerateLSF(  xRBF_new   , ...
                             yRBF_new   , ...
                             hmax       , ...
                             hmin       , ...
                             vmax       , ...
                             vmin       , ...
                             aRBF_new   , ...
                             SR         , ...
                             OFFSET      );

        figure(1026)
        clf;
        set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
        contourf(xmesh, ymesh, LSF, [0 0])
        axis image
        axis([hmin hmax vmin vmax])
        set(gca, 'FontSize', 20)
        title(sprintf('Zero level-set curve'))
        xlabel('x')
        ylabel('y')
    end %if DEBUG

    xy_spln = [];
    xy_tpar = [];
    xy_curv = [];
    xy_topo = [];
    [ xy_spln, ...
      xy_tpar, ...
      xy_curv, ...
      xy_topo, ...
      nbdy     ...
            ] = LevelSetSpline( xg       , ...
                                yg       , ...
                                xRBF_new, ...
                                yRBF_new, ...
                                aRBF_new, ...
                                SR       , ...
                                OFFSET   , ...
                                x_tol    , ...
                                intol    , ...
                                tspan    , ...
                                hstep    , ...
                                eta_tol  , ...
                                poly_fit , ...
                                mpts     , ...
                                hmin     , ...
                                hmax     , ...
                                vmin     , ...
                                vmax     , ...
                                DEBUG     );
    xySpln_new = [];
    xyPoly_new = [];
    xyCurv_new = [];
    xyTpar_new = [];
    if(poly_fit == 1 || poly_fit == 3)
        npts       = length(xy_spln(:,1));
        xySpln_new = zeros(2*(npts),1);
        for i = 1 : npts
            xySpln_new(2*i-1) = xy_spln(i,1);
            xySpln_new(2*i  ) = xy_spln(i,2);
        end %for i
    elseif(poly_fit == 2)
        npts       = length(xy_spln(:,1));
        xySpln_new = zeros(2*(npts),1);
        for i = 1 : npts
            xySpln_new(2*i-1) = xy_spln(i,1);
            xySpln_new(2*i  ) = xy_spln(i,2);
        end %for i
        npts        = length(xy_topo(:,1));
        xyPoly_new = zeros(2*(npts),1);
        for i = 1 : npts
            xyPoly_new(2*i-1) = xy_topo(i,1);
            xyPoly_new(2*i  ) = xy_topo(i,2);
        end %for i
    end %if
    xyCurv_new = xy_curv;
    xyTpar_new = xy_tpar;

    if(poly_fit == 1 || poly_fit == 3)

        ObjFunc  = EvaluateObjective(  xySpln_new    , ...
                                         xyCurv_new    , ...
                                         xyTpar_new    , ...
                                         poly_fit      , ...
                                         xrays_des , ...
                                         NRML          , ...
                                         nrays         , ...
                                         hmin          , ...
                                         hmax          , ...
                                         vmin          , ...
                                         vmax          , ...
                                         1               );
    elseif(poly_fit == 2)

        ObjFunc  = EvaluateObjective(  xyPoly_new    , ...
                                         xyCurv_new    , ...
                                         xyTpar_new    , ...
                                         poly_fit      , ...
                                         xrays_des , ...
                                         NRML          , ...
                                         nrays         , ...
                                         hmin          , ...
                                         hmax          , ...
                                         vmin          , ...
                                         vmax          , ...
                                         1               );
    end %if

    LSF    = GenerateLSF(  xRBF_new, ...
                            yRBF_new, ...
                            hmax     , ...
                            hmin     , ...
                            vmax     , ...
                            vmin     , ...
                            aRBF_new, ...
                            SR       , ...
                            OFFSET    );

    figure(45322)
    subplot(2,2,2)
    contourf(xmesh, ymesh, LSF, 'LineStyle', 'none')
    hold on
    contour(xmesh, ymesh, LSF, [0 0], 'k--', 'LineWidth', 1.5)
    for iRBF = 1 : length(xRBF_new)

        LSFchck= EvaluateLSF(  xRBF_new(iRBF), ...
                                yRBF_new(iRBF), ...
                                xRBF_new       , ...
                                yRBF_new       , ...
                                aRBF_new       , ...
                                SR             , ...
                                OFFSET           );
        if(LSFchck > 0) %inside
            plot(xRBF_new(iRBF), yRBF_new(iRBF), 'bp')
        elseif(LSFchck < 0) %outside
```

```
            plot(xRBF_new(iRBF), yRBF_new(iRBF), 'rp')
        else
            plot(xRBF_new(iRBF), yRBF_new(iRBF), 'kp')
        end %if

    end %for iRBF

    kpts = length(xySpln_new)/2;
    xyS  = zeros(kpts, 2);
    for ip = 1 : kpts
        xyS(ip,1) = xySpln_new(2*ip-1);
        xyS(ip,2) = xySpln_new(2*ip  );
    end %for ip

    figure(45322)
    ws = plot(xyS(:,1), xyS(:,2), 'bo');
    hold off
    xlabel('x')
    ylabel('y')
    title( [{sprintf('Level-set function')                          }, ...
           {sprintf('iteration: %4d, alpha = %f', opt_iter, alpha)}] )
    axis image
    axis([hmin hmax vmin vmax])
    colorbar
    caxis([-1 1])
    set(gca, 'FontSize', 15)
    ipng = ipng + 1;
    saveas(gcf, sprintf('./%s/iter_%05d.png', opt_dir,opt_iter));
    saveas(gcf, sprintf('./%s/iter_%05d.png', gss_dir,    ipng));
    pause(0.1)

    deta = zeros(length(xySpln_new)/2,1);
    dphi = zeros(length(xySpln_new)/2,2);
    for i = 1 : length(xySpln_new)/2
        if(isnan(xySpln_new(2*i-1)))
            continue
        end %if
        xpnt = xySpln_new(2*i-1);
        ypnt = xySpln_new(2*i  );
        phi  = EvaluateLSF( xpnt    , ...
                            ypnt    , ...
                            xRBF_new, ...
                            yRBF_new, ...
                            aRBF_new    , ...
                            SR      , ...
                            OFFSET    );

        dphi(i,:)= grad_phi( [xpnt, ypnt], ...
                             xRBF_new    , ...
                             yRBF_new    , ...
                             SR          , ...
                             aRBF_new      );
        deta(i)= -phi /(dphi(i,1)^2 + dphi(i,2)^2);
    end %for i

    if(DEBUG == 1)
        for ipnt = 1 : length(xySpln_new)/2-1
            fprintf(1,'norm(grad_phi[%4d]) = %8.6f\n', ipnt, norm(dphi(ipnt,:)));
        end %for ipnt
    end %if
    conv_hist(opt_iter+1) = ObjFunc;
    df                    = conv_hist(opt_iter) - conv_hist(opt_iter+1);
%
% -> Step 6b: Write data for iteration
%
    timing = toc;
%
% Write iteration data:
```

```
%
    npts = length(xySpln_new)/2;
    if(DEBUG == 0)
        nvar = 3;
        dim  = 2;

        desparam = zeros(3*mRBF,1);
        for i = 1 : mRBF
            desparam(3*i-2) = xRBF_new(i);
            desparam(3*i-1) = yRBF_new(i);
            desparam(3*i-0) = aRBF_new(i);
        end %for i

        if(comp_calc == 1)
            gradient = zeros(3*mRBF,1);
            for i = 1 : mRBF
                gradient(3*i-2) = cold(i + 0*mRBF);
                gradient(3*i-1) = cold(i + 1*mRBF);
                gradient(3*i-0) = cold(i + 2*mRBF);
            end %for i

            conj_gradient = zeros(3*mRBF,1);
            for i = 1 : mRBF
                conj_gradient(3*i-2) = dold(i + 0*mRBF);
                conj_gradient(3*i-1) = dold(i + 1*mRBF);
                conj_gradient(3*i-0) = dold(i + 2*mRBF);
            end %for i
        elseif(comp_calc == 2)
            gradient = zeros(2*mRBF,1);
            for i = 1 : mRBF
                gradient(2*i-1) = cold(i + 0*mRBF);
                gradient(2*i  ) = cold(i + 1*mRBF);
            end %for i

            conj_gradient = zeros(2*mRBF,1);
            for i = 1 : mRBF
                conj_gradient(2*i-1) = dold(i + 0*mRBF);
                conj_gradient(2*i  ) = dold(i + 1*mRBF);
            end %for i
        elseif(comp_calc == 3 || comp_calc == 4)
            gradient = zeros(mRBF,1);
            for i = 1 : mRBF
                gradient(i) = cold(i);
            end %for i

            conj_gradient = zeros(mRBF,1);
            for i = 1 : mRBF
                conj_gradient(i) = dold(i);
            end %for i
        end %if

        directory = sprintf('./%s/iter_%05d/', opt_dir, opt_iter);

        write_data(  mRBF          , ...
                     nvar          , ...
                     desparam      , ...
                     npts          , ...
                     dim           , ...
                     xySpln_new    , ...
                     comp_calc     , ...
                     gradient      , ...
                     conj_gradient , ...
                     ObjFunc       , ...
                     alpha         , ...
                     beta          , ...
                     GSS_brak      , ...
                     GSS_iter      , ...
                     directory      );
    end %if DEBUG
    fpo = 1;
    fprintf(fpo,'%4d, %6.2f, %6.1e, %6.2e, %+9.4f, %+9.4f, %5d\n', ...
              opt_iter, timing, df, min(ObjFunc, 9999), beta, norm(dOdp), npts);

    fname = sprintf('./%s/optimization_history.txt', case_dir);
    fpo   = fopen(fname, 'a');
    fprintf(fpo,'%4d, %6.2f, %6.1e, %6.2e, %+9.4f, %+9.4f, %5d\n', ...
              opt_iter, timing, df, min(ObjFunc, 9999), beta, norm(dOdp), npts);
    fclose(fpo);
%
% -> Step 6c: Check convergence criteria and choose to continue or exit
%
% ==> Exit, maximum number of iterations reached
%
    if(opt_iter == MAX_ITER)
        xyRBF_opt = zeros(2*length(xRBF_new),1);
        for i = 1 : length(xRBF_new)
            xyRBF_opt(2*i-1) = xRBF_new(i);
            xyRBF_opt(2*i  ) = yRBF_new(i);
        end %for i
        aRBF_opt     = aRBF_new;
        if(poly_fit == 1 || poly_fit == 3)
            xySpln_opt  = xySpln_new;
        elseif(poly_fit == 2)
            xySpln_opt  = xyPoly_new;
        end %if

        fpo = 1;
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> maximum iterations reached.\n');

        fname = sprintf('./%s/optimization_history.txt', case_dir);
        fpo   = fopen(fname, 'a');
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> maximum iterations reached.\n');
        fclose(fpo);

        break;
    end %if
%
% ==> Exit, change in objective is less than dftol
%
    if(df < dftol)
        xyRBF_opt = zeros(2*length(xRBF_new),1);
        for i = 1 : length(xRBF_new)
            xyRBF_opt(2*i-1) = xRBF_new(i);
            xyRBF_opt(2*i  ) = yRBF_new(i);
        end %for i
        aRBF_opt     = aRBF_new;
        if(poly_fit == 1 || poly_fit == 3)
            xySpln_opt  = xySpln_new;
        elseif(poly_fit == 2)
            xySpln_opt  = xyPoly_new;
        end %if
        fpo = 1;
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> dftol exceeded.\n');

        fname = sprintf('./%s/optimization_history.txt', case_dir);
        fpo   = fopen(fname, 'a');
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> dftol exceeded.\n');
        fclose(fpo);

        break;
    end %if
%
```

```
% ==> Exit, objective is less than ftol
%
    if(ObjFunc < ftol)
        xyRBF_opt = zeros(2*length(xRBF_new),1);
        for i = 1 : length(xRBF_new)
            xyRBF_opt(2*i-1) = xRBF_new(i);
            xyRBF_opt(2*i   ) = yRBF_new(i);
        end %for i
        aRBF_opt     = aRBF_new;
        if(poly_fit == 1 || poly_fit == 3)
            xySpln_opt  = xySpln_new;
        elseif(poly_fit == 2)
            xySpln_opt  = xyPoly_new;
        end %if
        fpo = 1;
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> ftol exceeded.\n');

        fname = sprintf('./%s/optimization_history.txt', case_dir);
        fpo   = fopen(fname, 'a');
        fprintf(fpo, 'Optimization scheme complete!\n');
        fprintf(fpo, '===> ftol exceeded.\n');
        fclose(fpo);

        break;
    end %if
%
% -> Continue to next iteration and update _old variables and _new
%
    tic % Start timer over for iteration
    cold       = [];
    dold       = [];
    aRBF_old   = [];
    xRBF_old   = [];
    yRBF_old   = [];
    xySpln_old= [];
    xyTpar_old= [];
    xyCurv_old= [];

    cold       = cnew;
    dold       = dnew;
    aRBF_old   = aRBF_new;
    xRBF_old   = xRBF_new;
    yRBF_old   = yRBF_new;
    xySpln_old= xySpln_new;
    xyTpar_old= xyTpar_new;
    xyCurv_old= xyCurv_new;
    if(DEBUG == 1)
        fprintf(1,'Number of bodies: %3d\n', nbdy);
    end %if DEBUG
%
% Calculate sensitivities
%
    if(DEBUG == 1)
        fprintf(1, 'Calculate sensitivities:\n');
    end %if DEBUG
    if(sens_calc == 1) % FD
        dOdp = finite_diff_sensitivities( xRBF_old  , ...
                                          yRBF_old  , ...
                                          aRBF_old  , ...
                                          SR        , ...
                                          nbdy      , ...
                                          poly_fit  , ...
                                          xySpln_old, ...
                                          xyCurv_old, ...
                                          xyTpar_old, ...
                                          xrays_des , ...
                                          nrays     , ...
                                          NRML       , ...
                                          hmin       , ...
                                          hmax       , ...
                                          vmin       , ...
                                          vmax        );
        elseif(sens_calc == 2) % TL
            dOdp = tangent_sensitivities(  xRBF_old      , ...
                                           yRBF_old      , ...
                                           aRBF_old      , ...
                                           SR            , ...
                                           nbdy          , ...
                                           poly_fit      , ...
                                           xySpln_old    , ...
                                           xyCurv_old    , ...
                                           xyTpar_old    , ...
                                           xrays_des , ...
                                           nrays         , ...
                                           NRML          , ...
                                           hmin          , ...
                                           hmax          , ...
                                           vmin          , ...
                                           vmax           );
        elseif(sens_calc == 3) % AD
            dOdp = adjoint_sensitivities(   xRBF_old      , ...
                                            yRBF_old      , ...
                                            aRBF_old      , ...
                                            SR            , ...
                                            nbdy          , ...
                                            poly_fit      , ...
                                            xySpln_old    , ...
                                            xyCurv_old    , ...
                                            xyTpar_old    , ...
                                            xrays_des , ...
                                            nrays         , ...
                                            NRML          , ...
                                            hmin          , ...
                                            hmax          , ...
                                            vmin          , ...
                                            vmax           );
        elseif(sens_calc == 4)
            dOdp = complexstep_sensitivities(   xRBF_old      , ...
                                                yRBF_old      , ...
                                                aRBF_old      , ...
                                                SR            , ...
                                                nbdy          , ...
                                                poly_fit      , ...
                                                mpts          , ...
                                                xySpln_old    , ...
                                                xrays_des , ...
                                                nrays         , ...
                                                NRML          , ...
                                                hmin          , ...
                                                hmax          , ...
                                                vmin          , ...
                                                vmax          , ...
                                                comp_calc      );
        end %if
    end %for opt_iter
end
%function levelset_topology_optimizer

%-----------------------------------------------------------------------
```

```
function [ alpha    , ...
           GSS_brak, ...
           GSS_iter ] = GoldenSectionSearch(  delta         , ...
                                              I_tol         , ...
                                              dnew          , ...
                                              xRBF_old      , ...
                                              yRBF_old      , ...
                                              aRBF_old      , ...
                                              SR            , ...
                                              OFFSET        , ...
                                              hmax          , ...
                                              hmin          , ...
                                              vmax          , ...
                                              vmin          , ...
                                              xmesh         , ...
                                              ymesh         , ...
                                              xg            , ...
                                              yg            , ...
                                              x_tol         , ...
                                              intol         , ...
                                              tspan         , ...
                                              hstep         , ...
                                              eta_tol       , ...
                                              poly_fit      , ...
                                              mpts          , ...
                                              xrays_des     , ...
                                              NRML          , ...
                                              nrays         , ...
                                              comp_calc     , ...
                                              ipng          , ...
                                              gss_dir       , ...
                                              DEBUG           )
    % function GoldenSectionSearch( ) uses the 1D golden section search
    % algorithm to find the optimum step to take along the current gradient
    % direction
    %
    % Inputs:
    %                 delta      -> Initial step for 1D search and delta
    %                               parameter for calculating successive
    %                               step sizes
    %                 I_tol      -> Tolerance for interval of
    %                               uncertainty
    %                 dnew       -> Total number of bodies
    %                 xRBF_old   -> Inital x-coord.   of RBFs
    %                 yRBF_old   -> Inital y-coord.   of RBFs
    %                 aRBF_old   -> Inital coefficient of RBFs
    %                 SR         -> Support radius
    %                 OFFSET     -> Offset for LSF calcs
    %                 hmax       -> Maximum y-coordinate of the
    %                               horizontal x-rays
    %                 hmin       -> Minimum y-coordinate of the
    %                               horizontal x-rays
    %                 vmax       -> Maximum x-coordinate of the
    %                               vertical x-rays
    %                 vmin       -> Minimum x-coordinate of the
    %                               vertical x-rays
    %                 xmesh      -> x-coordinates for LSF plotting
    %                 ymesh      -> y-coordinates for LSF plotting
    %                 xg         -> Array of x-values for zero-point
    %                               identification on one or more
    %                               level-set curves
    %                 yg         -> Array of y-values for zero-point
    %                               identification on one or more
    %                               level-set curves
    %                 x_tol      -> Tolerance for the difference between
    %                               free variables in the zero-point
    %                               identification algorithm
    %                 intol      -> Tolerance for whether a point is
%                                 inside/outside an existing boundary
%                                 in the zero-point identification
%                                 algorithm
%                   tspan      -> Range of parametric coordinate for
%                                 the level-set RK4 algorithm
%                   hstep      -> Initial step size for the level-set
%                                 RK4 algorithm
%                   eta_tol    -> Tolerance used to test whether a
%                                 step taken by the RK4 algorithm is
%                   poly_fit   -> Fit type
%                   mpts       -> number of points along spline
%                   xrays_des  -> Desired x-rays for objective calcs
%                   NRML       -> Whether the objective function
%                                 should be normalized or not
%                   nrays      -> Number of rays used for objective
%                                 calculations
%                   comp_calc  -> Indicator for number of design
%                                 variables:
%                                 1 - RBF locations and coefficients;
%                                 2 - RBF locations;
%                                 3 - RBF coefficients
%                                 within an acceptable distance
%                   ipng       -> Counter for golden section search
%                                 figures
%                   gss_dir    -> golden section search directory for
%                                 saving files at each search
%                                 iteration
%                   DEBUG      -> Indicator for debugging the code:
%                                 0 - run as normal;
%                                 1 - debugging output to screen;
%
% Outputs:
%                   alpha      -> Optimal 1D step along gradient
%                                 direction
%                   GSS_brak   -> Variable for storing the bracketing
%                                 phase of the golden section search
%                                 algorithm
%                   GSS_iter   -> Variable for storing the iterating
%                                 phase of the golden section search
%                                 algorithm
%
% Annontated by Jack Rossetti________________________________02/24/20

% ====================== GOLDEN SEARCH METHOD ======================== %
% ===> Step 1: (Phase 1) For a chosen small number delta, calculate
%               f(0), f(a0), ... , f(ai) where ai are given by
%
%                 i
%               ____
%               \
%       ai   =  /      delta* (1.618)^j; q = 0, 1, 2, ...
%               ____
%               j = 0
%
%                Let q be the smallest integer that satisfies
%                f(a_{q-1}) < f(a_{q-2}) and f(a_{q-1}) < f(a_{q}).
%                The upper and lower bounds (aU and aL) on a* (optimum
%                value for a) are given by a_{q} and a_{q-2},
%                respectively. The interval of uncertainty is given as
%                I = aU - aL.
%
DEBUG    = 2;
GSS_brak = []; % Initialize data storage variables
GSS_iter = []; % Initialize data storage variables
f = zeros(1000,1);
a = zeros(1000,1);
mRBF  = length(xRBF_old);
alpha     = [];
```

```
dxRBF_GSS = [];
dyRBF_GSS = [];
daRBF_GSS = [];
if(comp_calc == 1)
    dstep = a(1)*dnew;
    dxRBF_GSS = dstep(       1 : 1*mRBF);
    dyRBF_GSS = dstep(1*mRBF+1: 2*mRBF);
    daRBF_GSS = dstep(2*mRBF+1: 3*mRBF);
elseif(comp_calc == 2)
    dstep = a(1)*dnew;
    dxRBF_GSS = dstep(       1 : 1*mRBF);
    dyRBF_GSS = dstep(1*mRBF+1: 2*mRBF);
    daRBF_GSS = zeros(size(dstep(       1 : 1*mRBF)));
elseif(comp_calc == 3)
    daRBF_GSS = a(1)*dnew;
    dxRBF_GSS = zeros(size(daRBF_GSS));
    dyRBF_GSS = zeros(size(daRBF_GSS));
elseif(comp_calc == 4)
    dxRBF_GSS = a(1)*dnew;
    dyRBF_GSS = zeros(size(dxRBF_GSS));
    daRBF_GSS = zeros(size(dxRBF_GSS));
end %if
xRBF_GSS = xRBF_old + dxRBF_GSS;
yRBF_GSS = yRBF_old + dyRBF_GSS;
aRBF_GSS = aRBF_old + daRBF_GSS;

if(DEBUG == 1)
    LSF  = GenerateLSF(  xRBF_GSS   , ...
                         yRBF_GSS   , ...
                         hmax       , ...
                         hmin       , ...
                         vmax       , ...
                         vmin       , ...
                         aRBF_GSS   , ...
                         SR         , ...
                         OFFSET       );

    figure(1026);
    clf;
    set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
    contourf(xmesh, ymesh, LSF, [0 0])
    axis image
    axis([hmin hmax vmin vmax])
    set(gca, 'FontSize', 20)
    title(sprintf('Zero level-set curve'))
    xlabel('x')
    ylabel('y')
end %if DEBUG

xy_spln = [];
xy_tpar = [];
xy_curv = [];
xy_topo = [];
[ xy_spln, ...
  xy_tpar, ...
  xy_curv, ...
  xy_topo, ...
  ~        ...
         ] = LevelSetSpline( xg       , ...
                             yg       , ...
                             xRBF_GSS, ...
                             yRBF_GSS, ...
                             aRBF_GSS, ...
                             SR       , ...
                             OFFSET   , ...
                             x_tol    , ...
                             intol    , ...
                             tspan    , ...
```

```
                                    hstep    , ...
                                    eta_tol  , ...
                                    poly_fit , ...
                                    mpts     , ...
                                    hmin     , ...
                                    hmax     , ...
                                    vmin     , ...
                                    vmax     , ...
                                    DEBUG     );
if(poly_fit == 1 || poly_fit == 3)
    npts       = length(xy_spln(:,1));
    xySpln_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xySpln_GSS(2*i-1) = xy_spln(i,1);
        xySpln_GSS(2*i  ) = xy_spln(i,2);
    end %for i
elseif(poly_fit == 2)
    npts       = length(xy_spln(:,1));
    xySpln_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xySpln_GSS(2*i-1) = xy_spln(i,1);
        xySpln_GSS(2*i  ) = xy_spln(i,2);
    end %for i

    npts       = length(xy_topo(:,1));
    xyPoly_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xyPoly_GSS(2*i-1) = xy_topo(i,1);
        xyPoly_GSS(2*i  ) = xy_topo(i,2);
    end %for i
end %if
xyCurv_GSS = xy_curv;
xyTpar_GSS = xy_tpar;
%
% Evaluate the objective function
%
if(poly_fit == 1 || poly_fit == 3)

    f(1)  = EvaluateObjective(  xySpln_GSS   , ...
                                     xyCurv_GSS   , ...
                                     xyTpar_GSS   , ...
                                     poly_fit     , ...
                                     xrays_des , ...
                                     NRML         , ...
                                     nrays        , ...
                                     hmin         , ...
                                     hmax         , ...
                                     vmin         , ...
                                     vmax         , ...
                                     DEBUG         );
elseif(poly_fit == 2)

    f(1)  = EvaluateObjective(  xyPoly_GSS   , ...
                                     xyCurv_GSS   , ...
                                     xyTpar_GSS   , ...
                                     poly_fit     , ...
                                     xrays_des , ...
                                     NRML         , ...
                                     nrays        , ...
                                     hmin         , ...
                                     hmax         , ...
                                     vmin         , ...
                                     vmax         , ...
                                     DEBUG         );
end %if

if(DEBUG == 1)
    LSF   = GenerateLSF(  xRBF_GSS, ...
                        yRBF_GSS,  ...
                        hmax       ,  ...
                        hmin       ,  ...
                        vmax       ,  ...
                        vmin       ,  ...
                        aRBF_GSS,  ...
                        SR        ,  ...
                        OFFSET    );

    figure(45322)
    subplot(2,2,2)
    contourf(xmesh, ymesh, LSF, 'LineStyle', 'none')
    hold on
    contour(xmesh, ymesh, LSF, [0 0], 'k--', 'LineWidth', 1.5)
    for iRBF = 1 : length(xRBF_GSS)

        LSFchck= EvaluateLSF(  xRBF_GSS(iRBF), ...
                               yRBF_GSS(iRBF), ...
                               xRBF_GSS       , ...
                               yRBF_GSS       , ...
                               aRBF_GSS       , ...
                               SR             , ...
                               OFFSET         );
        if(LSFchck > 0) %inside
            plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'bp')
        elseif(LSFchck < 0) %outside
            plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'rp')
        else
            plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'kp')
        end %if

    end %for iRBF

    kpts = length(xySpln_GSS)/2;
    xy_GSS  = zeros(kpts, 2);
    for ip = 1 : kpts
        xy_GSS(ip,1) = xySpln_GSS(2*ip-1);
        xy_GSS(ip,2) = xySpln_GSS(2*ip  );
    end %for ip

    plot(xy_GSS(:,1), xy_GSS(:,2), 'bo');
    hold off
    xlabel('x')
    ylabel('y')
    title( [{sprintf('Level-set function')                    }, ...
            {sprintf('Golden section search, alpha = %f', a(1))}] )
    axis square
    axis([hmin hmax vmin vmax])
    colorbar
    caxis([-1 1])
    set(gca, 'FontSize', 15)
    ipng = ipng + 1;
    saveas(gcf, sprintf('./%s/iter_%05d.png', gss_dir,ipng));
    pause(0.1)
end %if DEBUG

if(DEBUG == 2)
    fprintf(1, 'k = %6d, f = %+7.6f\n', 1, f(1));
end %if DEBUG
k   = 2;
alf = 0;
for gss_iter = 2 : 10000
    a(k)      = a(k-1) + delta*(1.618)^(k-1);

    dxRBF_GSS = [];
    dyRBF_GSS = [];
    daRBF_GSS = [];
    if(comp_calc == 1)
dstep = a(k)*dnew;
dxRBF_GSS = dstep(       1 : 1*mRBF);
dyRBF_GSS = dstep(1*mRBF+1: 2*mRBF);
daRBF_GSS = dstep(2*mRBF+1: 3*mRBF);
elseif(comp_calc == 2)
dstep = a(k)*dnew;
dxRBF_GSS = dstep(       1 : 1*mRBF);
dyRBF_GSS = dstep(1*mRBF+1: 2*mRBF);
daRBF_GSS = zeros(size(dstep(       1 : 1*mRBF)));
elseif(comp_calc == 3)
daRBF_GSS = a(k)*dnew;
dxRBF_GSS = zeros(size(daRBF_GSS));
dyRBF_GSS = zeros(size(daRBF_GSS));
elseif(comp_calc == 4)
dxRBF_GSS = a(k)*dnew;
dyRBF_GSS = zeros(size(dxRBF_GSS));
daRBF_GSS = zeros(size(dxRBF_GSS));
end %if
xRBF_GSS = xRBF_old + dxRBF_GSS;
yRBF_GSS = yRBF_old + dyRBF_GSS;
aRBF_GSS = aRBF_old + daRBF_GSS;

if(DEBUG == 1)
LSF  = GenerateLSF(  xRBF_GSS   , ...
                     yRBF_GSS   , ...
                     hmax       , ...
                     hmin       , ...
                     vmax       , ...
                     vmin       , ...
                     aRBF_GSS   , ...
                     SR         , ...
                     OFFSET      );

figure(1026)
clf;
set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
contourf(xmesh, ymesh, LSF, [0 0])
axis image
axis([hmin hmax vmin vmax])
set(gca, 'FontSize', 20)
title(sprintf('Zero level-set curve'))
xlabel('x')
ylabel('y')
end %if DEBUG

xy_spln = [];
xy_tpar = [];
xy_curv = [];
xy_topo = [];
[ xy_spln, ...
xy_tpar, ...
xy_curv, ...
xy_topo, ...
~        ...
      ] = LevelSetSpline( xg       , ...
                          yg       , ...
                          xRBF_GSS, ...
                          yRBF_GSS, ...
                          aRBF_GSS, ...
                          SR       , ...
                          OFFSET   , ...
                          x_tol    , ...
                          intol    , ...
                          tspan    , ...
                          hstep    , ...
                          eta_tol  , ...
                          poly_fit , ...
                          mpts     , ...
```

```
                                  hmin      , ...
                                  hmax      , ...
                                  vmin      , ...
                                  vmax      , ...
                                  DEBUG      );
if(poly_fit == 1 || poly_fit == 3)
    npts       = length(xy_spln(:,1));
    xySpln_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xySpln_GSS(2*i-1) = xy_spln(i,1);
        xySpln_GSS(2*i  ) = xy_spln(i,2);
    end %for i
elseif(poly_fit == 2)
    npts       = length(xy_spln(:,1));
    xySpln_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xySpln_GSS(2*i-1) = xy_spln(i,1);
        xySpln_GSS(2*i  ) = xy_spln(i,2);
    end %for i

    npts       = length(xy_topo(:,1));
    xyPoly_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xyPoly_GSS(2*i-1) = xy_topo(i,1);
        xyPoly_GSS(2*i  ) = xy_topo(i,2);
    end %for i
end %if
xyCurv_GSS = xy_curv;
xyTpar_GSS = xy_tpar;
%
% Evaluate the objective function
%
if(poly_fit == 1 || poly_fit == 3)

    f(k)  = EvaluateObjective(  xySpln_GSS    , ...
                                 xyCurv_GSS    , ...
                                 xyTpar_GSS    , ...
                                 poly_fit      , ...
                                 xrays_des , ...
                                 NRML          , ...
                                 nrays         , ...
                                 hmin          , ...
                                 hmax          , ...
                                 vmin          , ...
                                 vmax          , ...
                                 DEBUG          );
elseif(poly_fit == 2)

    f(k)  = EvaluateObjective(  xyPoly_GSS    , ...
                                 xyCurv_GSS    , ...
                                 xyTpar_GSS    , ...
                                 poly_fit      , ...
                                 xrays_des , ...
                                 NRML          , ...
                                 nrays         , ...
                                 hmin          , ...
                                 hmax          , ...
                                 vmin          , ...
                                 vmax          , ...
                                 DEBUG          );
end %if

if(DEBUG == 1)
    LSF   = GenerateLSF(  xRBF_GSS, ...
                          yRBF_GSS, ...
                          hmax      , ...
                          hmin      , ...
                          vmax      , ...
```

```
                                vmin        , ...
                                aRBF_GSS, ...
                                SR          , ...
                                OFFSET      );

            figure(45322)
            subplot(2,2,2)
            contourf(xmesh, ymesh, LSF, 'LineStyle', 'none')
            hold on
            contour(xmesh, ymesh, LSF, [0 0], 'k--', 'LineWidth', 1.5)
            for iRBF = 1 : length(xRBF_GSS)

                LSFchck= EvaluateLSF(  xRBF_GSS(iRBF), ...
                                       yRBF_GSS(iRBF), ...
                                       xRBF_GSS       , ...
                                       yRBF_GSS       , ...
                                       aRBF_GSS       , ...
                                       SR             , ...
                                       OFFSET          );
                if(LSFchck > 0) %inside
                    plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'bp')
                elseif(LSFchck < 0) %outside
                    plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'rp')
                else
                    plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'kp')
                end %if

            end %for iRBF

            kpts = length(xySpln_GSS)/2;
            xy_GSS  = zeros(kpts, 2);
            for ip = 1 : kpts
                xy_GSS(ip,1) = xySpln_GSS(2*ip-1);
                xy_GSS(ip,2) = xySpln_GSS(2*ip  );
            end %for ip

            plot(xy_GSS(:,1), xy_GSS(:,2), 'bo');
            hold off
            xlabel('x')
            ylabel('y')
            title( [{sprintf('Level-set function')                    }, ...
                    {sprintf('Golden section search, alpha = %f', a(k))}] )
            axis square
            axis([hmin hmax vmin vmax])
            colorbar
            caxis([-1 1])
            set(gca, 'FontSize', 15)
            ipng = ipng + 1;
            saveas(gcf, sprintf('./%s/iter_%05d.png', gss_dir,ipng));
            pause(0.01)
        end %if DEBUG
        if(DEBUG == 2)
            fprintf(1, 'k = %6d, f = %+7.6f\n', k, f(k));
        end %if DEBUG
%
% Check the objective to see if the solution is bracketed
%
        if(f(k-1) < f(k)) && (k > 2)
            break;
        elseif (f(k-1) < f(k)) && (k == 2)
            delta= delta/10;
            alf  = 0;
            continue;
        end %if
        k = k+1;
end %for iter
GSS_brak = zeros(2*k+2,1); % Set up output variable for saving GSS
                           % iteration data
```

```
GSS_brak(1) = k;
GSS_brak(2) = 2;
for k1 = 2 : k+1
    GSS_brak(2*k1-1) = a(k1-1);
    GSS_brak(2*k1  ) = f(k1-1);
end %for k1
%
% ==> Step 2: (Phase 2) Compute f(aB), where aB = aL + 0.618*I.
%               Note that, at the first iteration, aA = aL + 0.382*I =
%               a_{q-1}, so f(aA) is already known.
%
if(alf == 0)
    aU = a(k);
    aL = a(k-2);
    I  = aU - aL;
    aB   = aL + 0.618*I;
    aA   = aL + 0.382*I;
    if(DEBUG == 2)
        fprintf(1, '\nf(%2d) = %10.4f, f(%2d) = %10.4f, f(%2d) = %10.4f\n', k-2, f(k-2),
             k-1, f(k-1), k, f(k))
    end %if DEBUG
end %if

clear a
clear f

a_GSS = zeros(2000,1);
f_GSS = zeros(2000,1);
k     = 0;
for gss_iter = 1 : 1000
    if(alf == 1)
        break;
    end %if
%
% alpha = aB
%
    dxRBF_GSS = [];
    dyRBF_GSS = [];
    daRBF_GSS = [];
    if(comp_calc == 1)
        dstep = aB*dnew;
        dxRBF_GSS = dstep(       1 : 1*mRBF);
        dyRBF_GSS = dstep(1*mRBF+1: 2*mRBF);
        daRBF_GSS = dstep(2*mRBF+1: 3*mRBF);
    elseif(comp_calc == 2)
        dstep = aB*dnew;
        dxRBF_GSS = dstep(       1 : 1*mRBF);
        dyRBF_GSS = dstep(1*mRBF+1: 2*mRBF);
        daRBF_GSS = zeros(size(dstep(       1 : 1*mRBF)));
    elseif(comp_calc == 3)
        daRBF_GSS = aB*dnew;
        dxRBF_GSS = zeros(size(daRBF_GSS));
        dyRBF_GSS = zeros(size(daRBF_GSS));
    elseif(comp_calc == 4)
        dxRBF_GSS = aB*dnew;
        dyRBF_GSS = zeros(size(dxRBF_GSS));
        daRBF_GSS = zeros(size(dxRBF_GSS));
    end %if
    xRBF_GSS = xRBF_old + dxRBF_GSS;
    yRBF_GSS = yRBF_old + dyRBF_GSS;
    aRBF_GSS = aRBF_old + daRBF_GSS;

    if(DEBUG == 1)
        LSF  = GenerateLSF(  xRBF_GSS   , ...
                             yRBF_GSS   , ...
                             hmax       , ...
                             hmin       , ...
                             vmax       , ...
```

```
                              vmin        , ...
                              aRBF_GSS    , ...
                              SR          , ...
                              OFFSET       );

    figure(1026)
    clf;
    set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
    contourf(xmesh, ymesh, LSF, [0 0])
    axis image
    axis([hmin hmax vmin vmax])
    set(gca, 'FontSize', 20)
    title(sprintf('Zero level-set curve'))
    xlabel('x')
    ylabel('y')
end %if DEBUG

xy_spln = [];
xy_tpar = [];
xy_curv = [];
xy_topo = [];
[ xy_spln, ...
  xy_tpar, ...
  xy_curv, ...
  xy_topo, ...
  ~        ...
          ] = LevelSetSpline( xg       , ...
                              yg       , ...
                              xRBF_GSS, ...
                              yRBF_GSS, ...
                              aRBF_GSS, ...
                              SR       , ...
                              OFFSET   , ...
                              x_tol    , ...
                              intol    , ...
                              tspan    , ...
                              hstep    , ...
                              eta_tol  , ...
                              poly_fit , ...
                              mpts     , ...
                              hmin     , ...
                              hmax     , ...
                              vmin     , ...
                              vmax     , ...
                              DEBUG     );
if(poly_fit == 1 || poly_fit == 3)
    npts       = length(xy_spln(:,1));
    xySpln_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xySpln_GSS(2*i-1) = xy_spln(i,1);
        xySpln_GSS(2*i  ) = xy_spln(i,2);
    end %for i
elseif(poly_fit == 2)
    npts       = length(xy_spln(:,1));
    xySpln_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xySpln_GSS(2*i-1) = xy_spln(i,1);
        xySpln_GSS(2*i  ) = xy_spln(i,2);
    end %for i

    npts       = length(xy_topo(:,1));
    xyPoly_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xyPoly_GSS(2*i-1) = xy_topo(i,1);
        xyPoly_GSS(2*i  ) = xy_topo(i,2);
    end %for i
end %if
xyCurv_GSS = xy_curv;
```

```
xyTpar_GSS = xy_tpar;
%
% Evaluate the objective function
%
if(poly_fit == 1 || poly_fit == 3)

    faB   = EvaluateObjective(  xySpln_GSS   , ...
                                xyCurv_GSS   , ...
                                xyTpar_GSS   , ...
                                poly_fit     , ...
                                xrays_des , ...
                                NRML         , ...
                                nrays        , ...
                                hmin         , ...
                                hmax         , ...
                                vmin         , ...
                                vmax         , ...
                                DEBUG         );
elseif(poly_fit == 2)

    faB   = EvaluateObjective(  xyPoly_GSS   , ...
                                xyCurv_GSS   , ...
                                xyTpar_GSS   , ...
                                poly_fit     , ...
                                xrays_des , ...
                                NRML         , ...
                                nrays        , ...
                                hmin         , ...
                                hmax         , ...
                                vmin         , ...
                                vmax         , ...
                                DEBUG         );
end %if

k = k + 1;
a_GSS(k) = aB;
f_GSS(k) = faB;

if(DEBUG == 1)
    LSF   = GenerateLSF(  xRBF_GSS, ...
                          yRBF_GSS, ...
                          hmax    , ...
                          hmin    , ...
                          vmax    , ...
                          vmin    , ...
                          aRBF_GSS, ...
                          SR      , ...
                          OFFSET   );

    figure(45322)
    subplot(2,2,2)
    contourf(xmesh, ymesh, LSF, 'LineStyle', 'none')
    hold on
    contour(xmesh, ymesh, LSF, [0 0], 'k--', 'LineWidth', 1.5)
    for iRBF = 1 : length(xRBF_GSS)

        LSFchck= EvaluateLSF(  xRBF_GSS(iRBF), ...
                               yRBF_GSS(iRBF), ...
                               xRBF_GSS      , ...
                               yRBF_GSS      , ...
                               aRBF_GSS      , ...
                               SR            , ...
                               OFFSET         );
        if(LSFchck > 0) %inside
            plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'bp')
        elseif(LSFchck < 0) %outside
            plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'rp')
        else
```

```
                plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'kp')
            end %if

        end %for iRBF

        kpts = length(xySpln_GSS)/2;
        xy_GSS  = zeros(kpts, 2);
        for ip = 1 : kpts
            xy_GSS(ip,1) = xySpln_GSS(2*ip-1);
            xy_GSS(ip,2) = xySpln_GSS(2*ip  );
        end %for ip

        plot(xy_GSS(:,1), xy_GSS(:,2), 'bo');
        hold off
        xlabel('x')
        ylabel('y')
        title( [{sprintf('Level-set function')                    }, ...
                {sprintf('Golden section search, alpha = %f', aB')}] )
        axis square
        axis([hmin hmax vmin vmax])
        colorbar
        caxis([-1 1])
        set(gca, 'FontSize', 15)
        ipng = ipng + 1;
        saveas(gcf, sprintf('./%s/iter_%05d.png', gss_dir,ipng));
        pause(0.01)
    end %if DEBUG
%
% alpha = aA
%
    dxRBF_GSS = [];
    dyRBF_GSS = [];
    daRBF_GSS = [];
    if(comp_calc == 1)
        dstep = aA*dnew;
        dxRBF_GSS = dstep(       1 : 1*mRBF);
        dyRBF_GSS = dstep(1*mRBF+1: 2*mRBF);
        daRBF_GSS = dstep(2*mRBF+1: 3*mRBF);
    elseif(comp_calc == 2)
        dstep = aA*dnew;
        dxRBF_GSS = dstep(       1 : 1*mRBF);
        dyRBF_GSS = dstep(1*mRBF+1: 2*mRBF);
        daRBF_GSS = zeros(size(dstep(       1 : 1*mRBF)));
    elseif(comp_calc == 3)
        daRBF_GSS = aA*dnew;
        dxRBF_GSS = zeros(size(daRBF_GSS));
        dyRBF_GSS = zeros(size(daRBF_GSS));
    elseif(comp_calc == 4)
        dxRBF_GSS = aA*dnew;
        dyRBF_GSS = zeros(size(dxRBF_GSS));
        daRBF_GSS = zeros(size(dxRBF_GSS));
    end %if
    xRBF_GSS = xRBF_old + dxRBF_GSS;
    yRBF_GSS = yRBF_old + dyRBF_GSS;
    aRBF_GSS = aRBF_old + daRBF_GSS;

    if(DEBUG == 1)
        LSF  = GenerateLSF(  xRBF_GSS   , ...
                             yRBF_GSS   , ...
                             hmax       , ...
                             hmin       , ...
                             vmax       , ...
                             vmin       , ...
                             aRBF_GSS   , ...
                             SR         , ...
                             OFFSET      );

        figure(1026)
    clf;
    set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
    contourf(xmesh, ymesh, LSF, [0 0])
    axis image
    axis([hmin hmax vmin vmax])
    set(gca, 'FontSize', 20)
    title(sprintf('Zero level-set curve'))
    xlabel('x')
    ylabel('y')
end %if DEBUG


xy_spln = [];
xy_tpar = [];
xy_curv = [];
xy_topo = [];
[ xy_spln, ...
  xy_tpar, ...
  xy_curv, ...
  xy_topo, ...
  ~        ...
          ] = LevelSetSpline( xg       , ...
                              yg       , ...
                              xRBF_GSS, ...
                              yRBF_GSS, ...
                              aRBF_GSS, ...
                              SR       , ...
                              OFFSET   , ...
                              x_tol    , ...
                              intol    , ...
                              tspan    , ...
                              hstep    , ...
                              eta_tol  , ...
                              poly_fit , ...
                              mpts     , ...
                              hmin     , ...
                              hmax     , ...
                              vmin     , ...
                              vmax     , ...
                              DEBUG     );
if(poly_fit == 1 || poly_fit == 3)
    npts       = length(xy_spln(:,1));
    xySpln_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xySpln_GSS(2*i-1) = xy_spln(i,1);
        xySpln_GSS(2*i  ) = xy_spln(i,2);
    end %for i
elseif(poly_fit == 2)
    npts       = length(xy_spln(:,1));
    xySpln_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xySpln_GSS(2*i-1) = xy_spln(i,1);
        xySpln_GSS(2*i  ) = xy_spln(i,2);
    end %for i

    npts       = length(xy_topo(:,1));
    xyPoly_GSS = zeros(2*(npts),1);
    for i = 1 : npts
        xyPoly_GSS(2*i-1) = xy_topo(i,1);
        xyPoly_GSS(2*i  ) = xy_topo(i,2);
    end %for i
end %if
xyCurv_GSS = xy_curv;
xyTpar_GSS = xy_tpar;
%
% Evaluate the objective function
%
if(poly_fit == 1 || poly_fit == 3)
```

```
    faA  = EvaluateObjective(  xySpln_GSS    , ...
                                   xyCurv_GSS    , ...
                                   xyTpar_GSS    , ...
                                   poly_fit      , ...
                                   xrays_des , ...
                                   NRML          , ...
                                   nrays         , ...
                                   hmin          , ...
                                   hmax          , ...
                                   vmin          , ...
                                   vmax          , ...
                                   DEBUG          );
elseif(poly_fit == 2)

    faA  = EvaluateObjective(  xyPoly_GSS    , ...
                                   xyCurv_GSS    , ...
                                   xyTpar_GSS    , ...
                                   poly_fit      , ...
                                   xrays_des , ...
                                   NRML          , ...
                                   nrays         , ...
                                   hmin          , ...
                                   hmax          , ...
                                   vmin          , ...
                                   vmax          , ...
                                   DEBUG          );
end %if

k = k + 1;
a_GSS(k) = aA;
f_GSS(k) = faA;

if(DEBUG == 1)
    LSF   = GenerateLSF(  xRBF_GSS, ...
                          yRBF_GSS, ...
                          hmax    , ...
                          hmin    , ...
                          vmax    , ...
                          vmin    , ...
                          aRBF_GSS, ...
                          SR      , ...
                          OFFSET    );

    figure(45322)
    subplot(2,2,2)
    contourf(xmesh, ymesh, LSF, 'LineStyle', 'none')
    hold on
    contour(xmesh, ymesh, LSF, [0 0], 'k--', 'LineWidth', 1.5)
    for iRBF = 1 : length(xRBF_GSS)

        LSFchck= EvaluateLSF(  xRBF_GSS(iRBF), ...
                               yRBF_GSS(iRBF), ...
                               xRBF_GSS      , ...
                               yRBF_GSS      , ...
                               aRBF_GSS      , ...
                               SR            , ...
                               OFFSET         );
        if(LSFchck > 0) %inside
            plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'bp')
        elseif(LSFchck < 0) %outside
            plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'rp')
        else
            plot(xRBF_GSS(iRBF), yRBF_GSS(iRBF), 'kp')
        end %if

    end %for iRBF
```

```
            kpts = length(xySpln_GSS)/2;
            xy_GSS  = zeros(kpts, 2);
            for ip = 1 : kpts
                xy_GSS(ip,1) = xySpln_GSS(2*ip-1);
                xy_GSS(ip,2) = xySpln_GSS(2*ip  );
            end %for ip

            plot(xy_GSS(:,1), xy_GSS(:,2), 'bo');
            hold off
            xlabel('x')
            ylabel('y')
            title( [{sprintf('Level-set function')                      }, ...
                    {sprintf('Golden section search, alpha = %f', aA)}] )
            axis square
            axis([hmin hmax vmin vmax])
            colorbar
            caxis([-1 1])
            set(gca, 'FontSize', 15)
            ipng = ipng + 1;
            saveas(gcf, sprintf('./%s/iter_%05d.png', gss_dir,ipng));
            pause(0.01)
        end %if DEBUG
    %
    % ===> Step 3: Compare f(aA) and f(aB), and go to (i), (ii), or (iii).
    %
    %           (i) If f(aA) < f(aB), then minimum point a* lies between aA
    %               and aU. New limits are aL = aL and aU = aB. Also,
    %               aB = aA. Compute f(aA), where aA = aL + 0.382(aU - aL)
    %               and go to Step 4.
    %
        if(DEBUG == 2)
            fprintf(1, 'GSS iter = %5d, faA = %f, faB = %f, aU = %g, aL = %g, I = %g\n',
                gss_iter, faA, faB, aU, aL, I);
        end %if DEBUG
        if (faA < faB)
%           aL = aL;
            aU = aB;
            aB = aA;
            aA = aL + 0.382*(aU - aL);
    %
    %          (ii) If f(aA) > f(aB), then minimum point a* lies between aL
    %               and aB. New limits are aL = aA and aU = aU. Also,
    %               aA = aB. Compute f(aB), where aB = aL + 0.618(aU - aL)
    %               and go to Step 4.
    %
        elseif (faA > faB)
%           aU = aU;
            aL = aA;
            aA = aB;
            aB = aL + 0.618*(aU - aL);
    %
    %         (iii) If f(aA) == f(aB), let aL = aA and aU = aB and return
    %               to Step 2.
    %
        else
            aL = aA;
            aU = aB;
            aB = aL + 0.618*(aU - aL);
            aA = aL + 0.382*(aU - aL);
        end %if
    %
    % ===> Step 4: If the new interval of uncertainty I = aU - aL is less
    %              than a stopping criterion tol, let a* = (aU + aL)/2 and
    %              stop. Otherwise, return to Step 3.
    %
        I = (aU - aL);
        if (I < I_tol)
            alpha = 0.5 * (aU + aL);
```

```
                break;
            end %if
        end %for gss_iter

        GSS_iter     = zeros(2*k+2,1);
        GSS_iter(1) = k;
        GSS_iter(2) = 2;

        for k1 = 2 : k+1
            GSS_iter(2*k1-1) = a_GSS(k1-1);
            GSS_iter(2*k1   ) = f_GSS(k1-1);
        end %for k1

        clear a_GSS
        clear f_GSS

end
%function GoldenSectionSearch

%--------------------------------------------------------------------------
```

```
function [ xRBF, ...
           yRBF, ...
           aRBF, ...
           mRBF ] = RBF_parameters( xRBF_mid, ...
                                    yRBF_mid, ...
                                    SR      , ...
                                    fSR     , ...
                                    nRBF    , ...
                                    nbdy    , ...
                                    OFFSET  , ...
                                    inout   , ...
                                    int_fig , ...
                                    int_dir , ...
                                    DEBUG     )
    % function RBF_parameters( ) takes in the midpoints of the RBF pairs
    % and distributes RBFs inside and outside the boundary represented by
    % the midpoints. Since the midpoints are calculated from the initial
    % curve, it is assumed that these points approximate the location of
    % the zero level-set curve. The RBFs are distributed to surround this
    % curve and when the coefficients are solved for, such that phi = +1
    % at the inside locations and phi = -1 at the outside locations, the
    % zero level-set curve is close to the initial curve.
    %
    % Inputs:
    %                 xRBF_mid   -> x-coordinate of the midpoint for RBF
    %                               pairs
    %                 yRBF_mid   -> y-coordinate of the midpoint for RBF
    %                               pairs
    %                 SR         -> RBF support radius
    %                 fSR        -> fraction of the support radius that
    %                               the inside and outside RBFs are
    %                               separated by
    %                 nRBF       -> number of inside RBFs
    %                 nbdy       -> number of boundaries in design
    %                 OFFSET     -> Offset value for solving for RBF
    %                               coefficients
    %                 inout      -> Indicator prescribing whether there
    %                               are both inside and outside RBFs or
    %                               just inside RBFs:
    %                               0 - only inside RBFs;
    %                               1 - both inside and outside RBFs
    %                 int_fig    -> initial figure parameter
    %                 int_dir    -> initial directory name
    %                 DEBUG      -> debugging parameter
    %
    % Outputs:
    %                 xRBF       -> x-coordinate of RBFs
    %                 yRBF       -> y-coordinate of RBFs
    %                 aRBF       -> RBF coefficients
    %                 mRBF       -> total number of RBFs
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    % If inout == 1 there are inside and outside RBFs defined below. First
    % calculate the approximate tangents at each point using central
    % differences and use them to calculate the normals:
    %
    if(inout == 1)
        dx       = zeros(nbdy*nRBF,1);
        dy       = zeros(nbdy*nRBF,1);
        theta    = zeros(nbdy*nRBF,1);
        for jbody = 1 : nbdy
            for i = 1 : nRBF
                ip  = i + (jbody-1)*nRBF;
                im1 = ip-1;
                ip1 = ip+1;
                if(ip == 1 + (jbody-1)*nRBF)
                    im1 = jbody*nRBF;
                    ip1 = ip+1;
                elseif(ip == jbody*nRBF)
                    im1 = ip-1;
                    ip1 = 1 + (jbody-1)*nRBF;
                end %if

                dx(ip)     = xRBF_mid(ip1) - xRBF_mid(im1);
                dy(ip)     = yRBF_mid(ip1) - yRBF_mid(im1);
                theta(ip) = atan2( dy(ip), dx(ip)) - pi/2;
            end %for i
        end %for jbody

        xRBF_in  = zeros(nbdy*nRBF,1);
        yRBF_in  = zeros(nbdy*nRBF,1);
        xRBF_out = zeros(nbdy*nRBF,1);
        yRBF_out = zeros(nbdy*nRBF,1);
        dnorm    = SR*ones(nbdy*nRBF,1);
    %
    % Distribute the inside RBFs a quarter of the support radius in the
    % negative normal direction (with the normal being outward facing) and
    % the outside RBFs a quarter of the support radius in the direction of
    % the normal. The result is an outside and inside RBF that are
    % separated by a distance equal to half the support radius.
    %
        for i = 1 : nbdy*nRBF
            xRBF_in(i)  = -fSR*dnorm(i) * cos(theta(i)) + xRBF_mid(i);
            yRBF_in(i)  = -fSR*dnorm(i) * sin(theta(i)) + yRBF_mid(i);
            xRBF_out(i) = +fSR*dnorm(i) * cos(theta(i)) + xRBF_mid(i);
            yRBF_out(i) = +fSR*dnorm(i) * sin(theta(i)) + yRBF_mid(i);
        end %for i
    %
    % Define the RHS such that the system of equations solves for the RBF
    % heights that set the value of the LSF at the inside RBFs equal to +1
    % and the LSF value at the outside RBFs equal to -1.
    %
        b_in      =2*OFFSET*ones(nbdy*nRBF,1);
        b_out     =0*ones(nbdy*nRBF,1);
        RHS       = [b_in; b_out];
        xRBF      = [xRBF_in; xRBF_out];
        yRBF      = [yRBF_in; yRBF_out];
        mRBF      = 2*nbdy*nRBF; % Total number of RBFs
    elseif(inout == 0)
        xRBF      = xRBF_mid;
        yRBF      = yRBF_mid;
        mRBF      = nRBF;
        RHS       =2*OFFSET*ones(mRBF,1);
    end %if

    if(DEBUG == 1)
        figure(2392)
        hold on
        plot(xRBF_in , yRBF_in , 'bp')
        plot(xRBF_out, yRBF_out, 'rp')
        hold off
        axis([min(xRBF_out) max(xRBF_out) min(yRBF_out) max(yRBF_out)])
        saveas(gcf, sprintf('./%s/RBF_distribution.png', int_dir))
    end %if

    M = zeros(mRBF, mRBF);
    if(DEBUG == 1)
        fprintf(1, '   \n');
        fprintf(1, '[ M ]\n');
    end %if

    for j = 1 : mRBF
        if(DEBUG == 1)
            fprintf(1, '[ ');
        end %if
        for i = 1 : mRBF
    %
    % Calculate the radius from the RBF location to the
    % intersection point:
    %
            r =((xRBF(j) -xRBF(i))^2 +...
                (yRBF(j) -yRBF(i))^2)^(1/2);
    %
    % Check if intersection point is outside of the RBFs support
    % radius:
    %
            if (r > SR )
                RBF = 0;
            elseif( r <= SR )
                RBF = (1 - (r/SR ))^4 *(4*(r/SR ) + 1);
            end %if
    %
    % Build the M matrix
    %
            M(j,i) = RBF;
            if(DEBUG == 1)
                fprintf(1, '%+5.2f, ', M(j,i));
            end %if
        end %for i
        if(DEBUG == 1)
            fprintf(1, ']\n');
        end %if
    end %for j

    aRBF = SquareMatrixSolver( M   , ...
                               RHS, ...
                               0    );

    if(DEBUG == 1)
        fprintf(1, '    \n');
        fprintf(1, '[ RHS ]\n');
        for iRBF = 1 : mRBF
        fprintf(1, '[ %+5.2f ]\n', RHS(iRBF));
        end %for iRBF
    end %if

    if(DEBUG == 1)
        fprintf(1, '    \n');
        fprintf(1, '[ ALFA ]\n');
        for iRBF = 1 : mRBF
        fprintf(1, '[ %+5.2f ]\n', aRBF(iRBF));
        end %for iRBF
    end %if
end
%function RBF_parameters

%--------------------------------------------------------------------------
```

```
function [ xyPoly, ...
           x_geo , ...
           y_geo , ...
           xyBOX   ...
                 ] = desired_geometry( icase )
    % function desired_geometry( ) produces the xy-coordinate array for the
    % requested geometry as well as the x- and y- coordinate arrays for
    % plotting. Also, the bounding box of the geometry is output.
    %
    % Inputs:
    %                  icase      -> case number for geometry
    %
    % Outputs:
    %                  xyPoly     -> xy-coordinates of the output geometry
    %                  x_geo      -> x-coordinates of desired geometry for
    %                                plotting
    %                  y_geo      -> y-coordinates of desired geometry for
    %                                plotting
    %                  xyBOX      -> bounding box of the geometry
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti________________________________02/24/20

    npts               = 1001;
    [ x_geo, ...
      y_geo, ...
      ~      ] = GeometryGenerator( icase, ...
                                    npts   );

    xyPoly = zeros(2*length(x_geo),1);

    for i = 1 : length(x_geo)
        xyPoly(2*i-1) = x_geo(i);
        xyPoly(2*i  ) = y_geo(i);
    end %for i

    xyBOX = [max(x_geo); ...
             min(x_geo); ...
             max(y_geo); ...
             min(y_geo) ];

end
%function desired_geometry

%------------------------------------------------------------------------
```

```
function [ xShap, ...
           yShap, ...
           nbdy  ] = GeometryGenerator( nShape, ...
                                        npts   )
    % function GeometryGenerator( ) produces the x- and y- coordinate
    % arrays as well as the number of bodies in the design domain.
    %
    % Inputs:
    %                  nShape     -> shape number for geometry
    %                  npts       -> number of points distributed around
    %                                each body
    %
    % Outputs:
    %                  xShap      -> x-coordinates of desired geometry for
    %                                plotting
    %                  yShap      -> y-coordinates of desired geometry for
    %                                plotting
    %                  nbdy       -> number of bodies
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    % nShape = 1 : Circle
    % nShape = 2 : Two ellipses side-by-side
    % nShape = 3 : Two ellipses diagonal side-by-side
    % nShape = 4 : NACA4420
    % nShape = 4 : Three NACA4420
    % nShape = 6 : Potato with varying concavities
    % nShape = 7 : One ellipse

    if nShape == 1
       % Circle
       n      = npts;
       dth    = 2*pi/n;
       theta  = 0 : dth : 2 *pi - dth;
       xShap1 = 0.5 *cos(theta) + 0;
       yShap1 = 0.5 *sin(theta) + 0;

       xShap = [xShap1, NaN];
       yShap = [yShap1, NaN];

       nbdy  = 1;

    elseif nShape == 2
       % Ellipse
       n      = npts;
       dth    = 2*pi/n;
       theta  = 0 : dth : 2 *pi - dth;
       xShap1 = 0.200 *cos(theta);
       yShap1 = 0.800 *sin(theta);

       %--- Rotate ellipse according to angle of attack ---%
       alfa = 0.0;
       RotM = [cos(alfa) -sin(alfa);...
               sin(alfa)  cos(alfa)];

       xyU  = RotM * [xShap1; yShap1];
       xU1  = xyU(1,:) - .4; %.7
       yU1  = xyU(2,:) - .0;
       xU2  = xyU(1,:) + .4; %.7
       yU2  = xyU(2,:) + .0;

       xShap = [xU1, NaN, xU2, NaN];
       yShap = [yU1, NaN, yU2, NaN];

       nbdy  = 2;

    elseif nShape == 3
```

```
% Ellipse
n      = npts;
dth    = 2*pi/n;
theta  = 0 : dth : 2 *pi - dth;
xShap1 = 0.200 *cos(theta);
yShap1 = 0.800 *sin(theta);

%—— Rotate ellipse according to angle of attack ——%
alfa = pi/4;
RotM = [cos(alfa) -sin(alfa);...
        sin(alfa)  cos(alfa)];

xyU  = RotM * [xShap1; yShap1];
xU1  = xyU(1,:) - .5;
yU1  = xyU(2,:) - .5;
xU2  = xyU(1,:) + .5;
yU2  = xyU(2,:) + .5;

% Turbine blade
xy = [  0.98536      -0.45539; ...
        0.95547      -0.42515; ...
        0.92549      -0.39421; ...
        0.89545      -0.36275; ...
        0.86534      -0.33090; ...
        0.83518      -0.29883; ...
        0.80498      -0.26670; ...
        0.77474      -0.23467; ...
        0.74447      -0.20288; ...
        0.71420      -0.17151; ...
        0.68391      -0.14069; ...
        0.65363      -0.11060; ...
        0.62337      -0.08139; ...
        0.59312      -0.05321; ...
        0.56292      -0.02623; ...
        0.53275      -0.00060; ...
        0.50263       0.02353; ...
        0.47258       0.04600; ...
        0.44260       0.06664; ...
        0.41270       0.08531; ...
        0.38289       0.10184; ...
        0.35318       0.11608; ...
        0.32358       0.12787; ...
        0.29410       0.13706; ...
        0.26475       0.14348; ...
        0.23553       0.14699; ...
        0.20647       0.14742; ...
        0.17756       0.14462; ...
        0.14882       0.13842; ...
        0.12026       0.12868; ...
        0.09188       0.11523; ...
        0.06370       0.09793; ...
        0.03572       0.07660; ...
        0.02929       0.07071; ...
        0.02340       0.06428; ...
        0.01808       0.05736; ...
        0.01340       0.05000; ...
        0.00937       0.04226; ...
        0.00603       0.03420; ...
        0.00341       0.02588; ...
        0.00152       0.01736; ...
        0.00038       0.00872; ...
        0.00000       0.00000; ...
        0.00038      -0.00872; ...
        0.00152      -0.01736; ...
        0.00341      -0.02588; ...
        0.00603      -0.03420; ...
        0.00937      -0.04226; ...
        0.01340      -0.05000; ...
```

```
0.01808      -0.05736; ...
0.02340      -0.06428; ...
0.02929      -0.07071; ...
0.03572      -0.07660; ...
0.04264      -0.08192; ...
0.05000      -0.08660; ...
0.05774      -0.09063; ...
0.06580      -0.09397; ...
0.07412      -0.09659; ...
0.08264      -0.09848; ...
0.09128      -0.09962; ...
0.10000      -0.10000; ...
0.10872      -0.09962; ...
0.11736      -0.09848; ...
0.12588      -0.09659; ...
0.13420      -0.09397; ...
0.15323      -0.08835; ...
0.17301      -0.08500; ...
0.19350      -0.08382; ...
0.21467      -0.08470; ...
0.23647      -0.08754; ...
0.25887      -0.09224; ...
0.28182      -0.09869; ...
0.30529      -0.10680; ...
0.32924      -0.11646; ...
0.35363      -0.12756; ...
0.37841      -0.14002; ...
0.40356      -0.15372; ...
0.42902      -0.16856; ...
0.45477      -0.18445; ...
0.48075      -0.20127; ...
0.50694      -0.21892; ...
0.53329      -0.23732; ...
0.55977      -0.25634; ...
0.58632      -0.27589; ...
0.61293      -0.29587; ...
0.63954      -0.31617; ...
0.66611      -0.33670; ...
0.69261      -0.35734; ...
0.71900      -0.37801; ...
0.74523      -0.39859; ...
0.77128      -0.41898; ...
0.79709      -0.43909; ...
0.82264      -0.45880; ...
0.84787      -0.47802; ...
0.87275      -0.49665; ...
0.89725      -0.51458; ...
0.92132      -0.53171; ...
0.92524      -0.53418; ...
0.92936      -0.53629; ...
0.93367      -0.53801; ...
0.93812      -0.53931; ...
0.94266      -0.54021; ...
0.94727      -0.54067; ...
0.95191      -0.54071; ...
0.95653      -0.54032; ...
0.96109      -0.53950; ...
0.96555      -0.53827; ...
0.96989      -0.53662; ...
0.97405      -0.53458; ...
0.97800      -0.53217; ...
0.98172      -0.52940; ...
0.98516      -0.52630; ...
0.98830      -0.52289; ...
0.99111      -0.51920; ...
0.99357      -0.51527; ...
0.99566      -0.51113; ...
0.99735      -0.50682; ...
```

```
                0.99863      -0.50237; ...
                0.99950      -0.49781; ...
                0.99994      -0.49320; ...
                0.99995      -0.48857; ...
                0.99954      -0.48395; ...
                0.99869      -0.47939; ...
                0.99743      -0.47493; ...
                0.99577      -0.47061; ...
                0.99370      -0.46646; ...
                0.99127      -0.46252; ...
                0.98848      -0.45882 ];

        xShap = [xU1, NaN, xy(:,1)', NaN];
        yShap = [yU1+.25, NaN, xy(:,2)'+.75, NaN];

        nbdy  = 2;

    elseif nShape == 4
        % Turbine blade
        xy = [  0.98536      -0.45539; ...
                0.95547      -0.42515; ...
                0.92549      -0.39421; ...
                0.89545      -0.36275; ...
                0.86534      -0.33090; ...
                0.83518      -0.29883; ...
                0.80498      -0.26670; ...
                0.77474      -0.23467; ...
                0.74447      -0.20288; ...
                0.71420      -0.17151; ...
                0.68391      -0.14069; ...
                0.65363      -0.11060; ...
                0.62337      -0.08139; ...
                0.59312      -0.05321; ...
                0.56292      -0.02623; ...
                0.53275      -0.00060; ...
                0.50263       0.02353; ...
                0.47258       0.04600; ...
                0.44260       0.06664; ...
                0.41270       0.08531; ...
                0.38289       0.10184; ...
                0.35318       0.11608; ...
                0.32358       0.12787; ...
                0.29410       0.13706; ...
                0.26475       0.14348; ...
                0.23553       0.14699; ...
                0.20647       0.14742; ...
                0.17756       0.14462; ...
                0.14882       0.13842; ...
                0.12026       0.12868; ...
                0.09188       0.11523; ...
                0.06370       0.09793; ...
                0.03572       0.07660; ...
                0.02929       0.07071; ...
                0.02340       0.06428; ...
                0.01808       0.05736; ...
                0.01340       0.05000; ...
                0.00937       0.04226; ...
                0.00603       0.03420; ...
                0.00341       0.02588; ...
                0.00152       0.01736; ...
                0.00038       0.00872; ...
                0.00000       0.00000; ...
                0.00038      -0.00872; ...
                0.00152      -0.01736; ...
                0.00341      -0.02588; ...
                0.00603      -0.03420; ...
                0.00937      -0.04226; ...
                0.01340      -0.05000; ...
```

```
0.01808     -0.05736;  ...
0.02340     -0.06428;  ...
0.02929     -0.07071;  ...
0.03572     -0.07660;  ...
0.04264     -0.08192;  ...
0.05000     -0.08660;  ...
0.05774     -0.09063;  ...
0.06580     -0.09397;  ...
0.07412     -0.09659;  ...
0.08264     -0.09848;  ...
0.09128     -0.09962;  ...
0.10000     -0.10000;  ...
0.10872     -0.09962;  ...
0.11736     -0.09848;  ...
0.12588     -0.09659;  ...
0.13420     -0.09397;  ...
0.15323     -0.08835;  ...
0.17301     -0.08500;  ...
0.19350     -0.08382;  ...
0.21467     -0.08470;  ...
0.23647     -0.08754;  ...
0.25887     -0.09224;  ...
0.28182     -0.09869;  ...
0.30529     -0.10680;  ...
0.32924     -0.11646;  ...
0.35363     -0.12756;  ...
0.37841     -0.14002;  ...
0.40356     -0.15372;  ...
0.42902     -0.16856;  ...
0.45477     -0.18445;  ...
0.48075     -0.20127;  ...
0.50694     -0.21892;  ...
0.53329     -0.23732;  ...
0.55977     -0.25634;  ...
0.58632     -0.27589;  ...
0.61293     -0.29587;  ...
0.63954     -0.31617;  ...
0.66611     -0.33670;  ...
0.69261     -0.35734;  ...
0.71900     -0.37801;  ...
0.74523     -0.39859;  ...
0.77128     -0.41898;  ...
0.79709     -0.43909;  ...
0.82264     -0.45880;  ...
0.84787     -0.47802;  ...
0.87275     -0.49665;  ...
0.89725     -0.51458;  ...
0.92132     -0.53171;  ...
0.92524     -0.53418;  ...
0.92936     -0.53629;  ...
0.93367     -0.53801;  ...
0.93812     -0.53931;  ...
0.94266     -0.54021;  ...
0.94727     -0.54067;  ...
0.95191     -0.54071;  ...
0.95653     -0.54032;  ...
0.96109     -0.53950;  ...
0.96555     -0.53827;  ...
0.96989     -0.53662;  ...
0.97405     -0.53458;  ...
0.97800     -0.53217;  ...
0.98172     -0.52940;  ...
0.98516     -0.52630;  ...
0.98830     -0.52289;  ...
0.99111     -0.51920;  ...
0.99357     -0.51527;  ...
0.99566     -0.51113;  ...
0.99735     -0.50682;  ...
```

```
            0.99863      −0.50237; ...
            0.99950      −0.49781; ...
            0.99994      −0.49320; ...
            0.99995      −0.48857; ...
            0.99954      −0.48395; ...
            0.99869      −0.47939; ...
            0.99743      −0.47493; ...
            0.99577      −0.47061; ...
            0.99370      −0.46646; ...
            0.99127      −0.46252; ...
            0.98848      −0.45882 ];

    xShap = [xy(:,1)', NaN] − 0.5;
    yShap = [xy(:,2)', NaN] + 0.2;

    nbdy  = 1;

elseif nShape == 5
    % Turbine blades
    xy = [  0.98536      −0.45539; ...
            0.95547      −0.42515; ...
            0.92549      −0.39421; ...
            0.89545      −0.36275; ...
            0.86534      −0.33090; ...
            0.83518      −0.29883; ...
            0.80498      −0.26670; ...
            0.77474      −0.23467; ...
            0.74447      −0.20288; ...
            0.71420      −0.17151; ...
            0.68391      −0.14069; ...
            0.65363      −0.11060; ...
            0.62337      −0.08139; ...
            0.59312      −0.05321; ...
            0.56292      −0.02623; ...
            0.53275      −0.00060; ...
            0.50263       0.02353; ...
            0.47258       0.04600; ...
            0.44260       0.06664; ...
            0.41270       0.08531; ...
            0.38289       0.10184; ...
            0.35318       0.11608; ...
            0.32358       0.12787; ...
            0.29410       0.13706; ...
            0.26475       0.14348; ...
            0.23553       0.14699; ...
            0.20647       0.14742; ...
            0.17756       0.14462; ...
            0.14882       0.13842; ...
            0.12026       0.12868; ...
            0.09188       0.11523; ...
            0.06370       0.09793; ...
            0.03572       0.07660; ...
            0.02929       0.07071; ...
            0.02340       0.06428; ...
            0.01808       0.05736; ...
            0.01340       0.05000; ...
            0.00937       0.04226; ...
            0.00603       0.03420; ...
            0.00341       0.02588; ...
            0.00152       0.01736; ...
            0.00038       0.00872; ...
            0.00000       0.00000; ...
            0.00038      −0.00872; ...
            0.00152      −0.01736; ...
            0.00341      −0.02588; ...
            0.00603      −0.03420; ...
            0.00937      −0.04226; ...
            0.01340      −0.05000; ...
```

```
0.01808      −0.05736;  ...
0.02340      −0.06428;  ...
0.02929      −0.07071;  ...
0.03572      −0.07660;  ...
0.04264      −0.08192;  ...
0.05000      −0.08660;  ...
0.05774      −0.09063;  ...
0.06580      −0.09397;  ...
0.07412      −0.09659;  ...
0.08264      −0.09848;  ...
0.09128      −0.09962;  ...
0.10000      −0.10000;  ...
0.10872      −0.09962;  ...
0.11736      −0.09848;  ...
0.12588      −0.09659;  ...
0.13420      −0.09397;  ...
0.15323      −0.08835;  ...
0.17301      −0.08500;  ...
0.19350      −0.08382;  ...
0.21467      −0.08470;  ...
0.23647      −0.08754;  ...
0.25887      −0.09224;  ...
0.28182      −0.09869;  ...
0.30529      −0.10680;  ...
0.32924      −0.11646;  ...
0.35363      −0.12756;  ...
0.37841      −0.14002;  ...
0.40356      −0.15372;  ...
0.42902      −0.16856;  ...
0.45477      −0.18445;  ...
0.48075      −0.20127;  ...
0.50694      −0.21892;  ...
0.53329      −0.23732;  ...
0.55977      −0.25634;  ...
0.58632      −0.27589;  ...
0.61293      −0.29587;  ...
0.63954      −0.31617;  ...
0.66611      −0.33670;  ...
0.69261      −0.35734;  ...
0.71900      −0.37801;  ...
0.74523      −0.39859;  ...
0.77128      −0.41898;  ...
0.79709      −0.43909;  ...
0.82264      −0.45880;  ...
0.84787      −0.47802;  ...
0.87275      −0.49665;  ...
0.89725      −0.51458;  ...
0.92132      −0.53171;  ...
0.92524      −0.53418;  ...
0.92936      −0.53629;  ...
0.93367      −0.53801;  ...
0.93812      −0.53931;  ...
0.94266      −0.54021;  ...
0.94727      −0.54067;  ...
0.95191      −0.54071;  ...
0.95653      −0.54032;  ...
0.96109      −0.53950;  ...
0.96555      −0.53827;  ...
0.96989      −0.53662;  ...
0.97405      −0.53458;  ...
0.97800      −0.53217;  ...
0.98172      −0.52940;  ...
0.98516      −0.52630;  ...
0.98830      −0.52289;  ...
0.99111      −0.51920;  ...
0.99357      −0.51527;  ...
0.99566      −0.51113;  ...
0.99735      −0.50682;  ...
```

```
                0.99863      -0.50237; ...
                0.99950      -0.49781; ...
                0.99994      -0.49320; ...
                0.99995      -0.48857; ...
                0.99954      -0.48395; ...
                0.99869      -0.47939; ...
                0.99743      -0.47493; ...
                0.99577      -0.47061; ...
                0.99370      -0.46646; ...
                0.99127      -0.46252; ...
                0.98848      -0.45882 ];

        xy1   = [xy(:,1) - 0.5, xy(:,2) + 1.2];
        xy2   = [xy(:,1) - 0.5, xy(:,2) + 0.2];
        xy3   = [xy(:,1) - 0.5, xy(:,2) - 0.8];

        xShap = [xy1(:,1)', NaN, ...
                 xy2(:,1)', NaN, ...
                 xy3(:,1)', NaN];

        yShap = [xy1(:,2)', NaN, ...
                 xy2(:,2)', NaN, ...
                 xy3(:,2)', NaN];

        nbdy  = 3;

%         xy1   = [xy(:,1) - 0.5, xy(:,2) + .5];
%         xy2   = [xy(:,1) - 0.5, xy(:,2) - .5];
%
%         xShap = [xy1(:,1)', NaN, ...
%                  xy2(:,1)', NaN];
%
%         yShap = [xy1(:,2)', NaN, ...
%                  xy2(:,2)', NaN];
%
%         nbdy  = 2;

    elseif nShape == 6
        X = bspline_func_jsr(3,npts+1);
        xShap = [X(1,1:end-1), NaN];
        yShap = [X(2,1:end-1), NaN];

        nbdy  = 1;
    elseif nShape == 7
        % ellipse
        n      = npts;
        dth    = 2*pi/n;
        theta  = 0 : dth : 2 *pi - dth;
        xShap1 = 0.15 *cos(theta) + 0;
        yShap1 = 1.00 *sin(theta) + 0;

        xShap = [xShap1, NaN];
        yShap = [yShap1, NaN];

        nbdy  = 1;
    elseif nShape == 9
        % ellipses
        n      = npts;
        dth    = 2*pi/n;
        theta  = 0 : dth : 2 *pi - dth;
        xShap1 = 0.25 *cos(theta) + 0.0;
        yShap1 = 0.10 *sin(theta) + 0.875;

        xShap2 = 0.15 *cos(theta) + 0.0;
        yShap2 = 0.15 *sin(theta) + 0.3;

        xShap3 = 0.125 *cos(theta) + 0.0;
        yShap3 = 0.500 *sin(theta) - 0.5;
        xShap = [xShap1, NaN, ...
                 xShap2, NaN, ...
                 xShap3, NaN ];
        yShap = [yShap1, NaN, ...
                 yShap2, NaN, ...
                 yShap3, NaN ];

        nbdy  = 3;
    end %if
%     toc
end
%function GeometricShape

%------------------------------------------------------------------------
```

```
function [ LSF ] = GenerateLSF(  xRBF   , ...
                                 yRBF   , ...
                                 xmax   , ...
                                 xmin   , ...
                                 ymax   , ...
                                 ymin   , ...
                                 aRBF   , ...
                                 SR     , ...
                                 OFFSET )
    % function GenerateLSF( ) generates the level-set function given the
    % RBF locations and coefficients
    %
    % Inputs:
    %                   xRBF        -> x-coord.     of RBFs
    %                   yRBF        -> y-coord.     of RBFs
    %                   xmax        -> Maximum x-coordinate of the
    %                                  level-set function domain
    %                   xmin        -> Minimum x-coordinate of the
    %                                  level-set function domain
    %                   ymax        -> Maximum y-coordinate of the
    %                                  level-set function domain
    %                   ymin        -> Minimum y-coordinate of the
    %                                  level-set function domain
    %                   aRBF        -> Coefficient of RBFs
    %                   SR          -> Support radius
    %                   OFFSET      -> Offset for LSF calcs
    %
    % Outputs:
    %                   LSF         -> 2D matrix of level-set values at the
    %                                  mesh points prescribed by the min
    %                                  and max xy-coords
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti__________________________________02/24/20

    delx  = 0.1;
    dely  = 0.1;
    x     = xmin : delx : xmax;
    y     = ymin : dely : ymax;

    [xmesh, ymesh] = meshgrid(x, y);

    LSF = ones(size(xmesh)) * (-OFFSET);
    [jmax, imax] = size(xmesh);

    [M,N] = size(xRBF);

    if(M == N)
        nRBF = M*N;
    elseif(M == 1 || N == 1)
        nRBF = length(xRBF);
    else
        error('M ~= N and neither M == 1 nor N == 1\n');
    end %if

    k = 1;
    for i = 1 : nRBF
        if(isnan(xRBF(i)))
            continue
        end %if
        RBF = zeros(size(xmesh));
        r   =sqrt((xmesh -xRBF(i)).^2 + (ymesh -yRBF(i)).^2);
        for ii = 1 : imax
            for jj = 1 : jmax
                if (r(jj,ii) < SR)
                    RBF(jj,ii) = aRBF(k)* (1 - (r(jj,ii)/SR)).^4 .*(4*(r(jj,ii)/SR) + 1);
                elseif (r(jj,ii) > SR)
                    continue
                    end %if
                end %for jj
            end %for ii
            LSF = LSF + RBF;
            k = k+1;
        end % for i

end
%function GenerateLSF

%-----------------------------------------------------------------------
```

```
function [ LSF ] = EvaluateLSF(  xpt    , ...
                                 ypt    , ...
                                 xRBF   , ...
                                 yRBF   , ...
                                 aRBF   , ...
                                 SR     , ...
                                 OFFSET )
    % function EvaluateLSF( ) evaluates the level-set function at a
    % prescribed point (xpt, ypt)
    %
    % Inputs:
    %                  xpt        -> x-coordinate of point to evalaute the
    %                                level-set function
    %                  ypt        -> y-coordinate of point to evalaute the
    %                                level-set function
    %                  xRBF       -> x-coord.    of RBFs
    %                  yRBF       -> y-coord.    of RBFs
    %                  aRBF       -> Coefficient of RBFs
    %                  SR         -> Support radius
    %                  OFFSET     -> Offset for LSF calcs
    %
    % Outputs:
    %                  LSF        -> the level-set function value
    %
    % Written by Jack Rossetti
    % Annotated by Jack Rossetti_________________________________02/24/20

    LSF = -OFFSET;
    [M,N] = size(xRBF);

    if(M == N)
        nRBF = M*N;
    elseif(M == 1 || N == 1)
        nRBF = length(xRBF);
    else
        error('M ~= N and neither M == 1 nor N == 1\n');
    end %if

    k   = 0;
    for i = 1 : nRBF
        if(isnan(xRBF(i)))
            continue
        end %if
        k   = k+1;
        RBF = 0.0;
        r   =sqrt((xpt -xRBF(i))^2 + (ypt -yRBF(i))^2);
        if (r < SR)
            RBF = aRBF(k)* (1 - (r/SR))^4 *(4*(r/SR) + 1);
        elseif (r > SR)
            continue
        end %if
        LSF = LSF + RBF;
    end % for i

end
%function EvaluateLSF

%------------------------------------------------------------------------
```

```
function [ O ...
          ] = EvaluateObjective( xyPoly        , ...
                                 d2xyPoly      , ...
                                 tPoly         , ...
                                 poly_fit      , ...
                                 xrays_des     , ...
                                 NRML          , ...
                                 nrays         , ...
                                 hmin          , ...
                                 hmax          , ...
                                 vmin          , ...
                                 vmax          , ...
                                 plotting        )
    % function EvaluateObjective( ) evaluates the objective function given
    % the spline points, curvature, and parametric distribution of the
    % points.
    %
    % Inputs:
    %                  xyPoly     -> input xy-coordinates for geometry
    %                  d2xyPoly   -> curvature information at xy-
    %                                coordinates
    %                  tPoly      -> parametric coordinate for the
    %                                geometry
    %                  poly_fit   -> type of fit
    %                  xrays_des  -> array containing the x-ray data to be
    %                                matched
    %                  NRML       -> boolean argument prescribing whether
    %                                the objective is normalized or not
    %                  nrays      -> number of rays used for x-ray
    %                                calculation
    %                  hmin       -> minimum y-value for horizontal ray
    %                  hmax       -> maximum y-value for horizontal ray
    %                  vmin       -> minimum x-value for vertical ray
    %                  vmax       -> maximum x-value for vertical ray
    %                  plotting   -> prescribes whether to plot the
    %                                objective function results
    %
    % Outputs:
    %                  O          -> the objective function value
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    mpts1 = length(xyPoly)/2;
    xShap1= zeros(mpts1, 1);
    yShap1= zeros(mpts1, 1);
    for ipt = 1 : mpts1
        xShap1(ipt) = xyPoly(2*ipt-1);
        yShap1(ipt) = xyPoly(2*ipt  );
    end %for ipt
    %
    % Produce an array containing the x-ray information for the given
    % xy-coordinates.
    %
    ho_ve = GetXray( xyPoly   , ...
                     d2xyPoly , ...
                     tPoly    , ...
                     poly_fit , ...
                     nrays    , ...
                     hmin     , ...
                     hmax     , ...
                     vmin     , ...
                     vmax      );

    h_rays1 = ones(nrays+1,1) * NaN;
    d_horz1 = ones(nrays+1,1) * NaN;
    v_rays1 = ones(nrays+1,1) * NaN;
    d_vert1 = ones(nrays+1,1) * NaN;
```

```
ray_type = 1; % indicates the ray being unzipped
ii = 0;
for i = 1 : 2*(nrays+1)
    if(isnan(ho_ve(2*i-1)) && isnan(ho_ve(2*i)))
        ray_type = 2;
        ii       = 0; % restart counter
        continue;
    end %if

    if(ray_type == 1)
        ii        = ii + 1;
        h_rays1(ii)= ho_ve(2*i-1);
        d_horz1(ii)= ho_ve(2*i  );
    elseif(ray_type == 2)
        ii        = ii + 1;
        v_rays1(ii)= ho_ve(2*i-1);
        d_vert1(ii)= ho_ve(2*i  );
    end %if

end %for i

h_rays2 = ones(nrays+1,1) * NaN;
d_horz2 = ones(nrays+1,1) * NaN;
v_rays2 = ones(nrays+1,1) * NaN;
d_vert2 = ones(nrays+1,1) * NaN;
ray_type= 1; % indicates the ray being unzipped
ii = 0;
for i = 1 : 2*(nrays+1)
    if(isnan(xrays_des(2*i-1)) && isnan(xrays_des(2*i)))
        ray_type = 2;
        ii       = 0; % restart counter
        continue;
    end %if

    if(ray_type == 1)
        ii        = ii + 1;
        h_rays2(ii)= xrays_des(2*i-1);
        d_horz2(ii)= xrays_des(2*i  );
    elseif(ray_type == 2)
        ii        = ii + 1;
        v_rays2(ii)= xrays_des(2*i-1);
        d_vert2(ii)= xrays_des(2*i  );
    end %if

end %for i
%
% Calculate the RMS of the differences between each x-ray
%
dx         = v_rays1(2) - v_rays1(1);
dy         = h_rays1(2) - h_rays1(1);
sqsumh     = 0.0;
sqsumv     = 0.0;

for i = 1 : nrays

    dh2   = ((d_horz1(i) - d_horz2(i)))^2;
    dv2   = ((d_vert1(i) - d_vert2(i)))^2;

    sqsumh= sqsumh      + dh2     ;
    sqsumv= sqsumv      + dv2     ;

end %for i

if(NRML == 0)
    ch = sqrt(dy/nrays);
    cv = sqrt(dx/nrays);
elseif(NRML == 1)
    ch = sqrt(dy/(nrays*max(d_horz2)^2));
    cv = sqrt(dx/(nrays*max(d_vert2)^2));
end %if

rms_h = ch*sqrt(sqsumh^2);
rms_v = cv*sqrt(sqsumv^2);
O     = rms_h + rms_v;

if( plotting == 1  && NRML == 1)

    figure(45322)
    set(gcf,'units', 'normalized', 'position', [0.0536 0.1752 0.5256 0.6714])

    subplot(2,2,1)
    plot(v_rays1, d_vert1/max(d_vert2), 'b-')
    hold on
    plot(v_rays2, d_vert2/max(d_vert2), 'r--')
    hold off
    grid on
    title(sprintf('rms_v = %8.6e', rms_v));
    xlabel('x')
    ylabel('height')
    axis square
    axis([vmin vmax 0 max(max(d_vert1),max(d_vert2))/max(d_vert2)*1.05])
    set(gca,'FontSize', 15);

    subplot(2,2,3)
    plot(xShap1, yShap1, 'b-')
    title(sprintf('objective = %8.6e', O));
    xlabel('x')
    ylabel('y')
    axis image
    axis([-2 2 -2 2])
    grid on
    set(gca,'FontSize', 15);

    subplot(2,2,4)
    plot(d_horz1/max(d_horz2), h_rays1, 'b-')
    hold on
    plot(d_horz2/max(d_horz2), h_rays2, 'r--')
    hold off
    grid on
    title(sprintf('rms_h = %8.6e', rms_h));
    xlabel('width')
    ylabel('y')
    axis square
    axis([0 max(max(d_horz1),max(d_horz2))/max(d_horz2)*1.05 hmin hmax])
    set(gca,'FontSize', 15);

elseif( plotting == 1  && NRML == 0)

    figure(45322)
    set(gcf,'units', 'normalized', 'position', [0.0536 0.1752 0.5256 0.6714])

    subplot(2,2,1)
    plot(v_rays1, d_vert1, 'b-')
    hold on
    plot(v_rays2, d_vert2, 'r--')
    hold off
    grid on
    title(sprintf('rms_v = %8.6e', rms_v));
    xlabel('x')
    ylabel('height')
    axis square
    axis([hmin hmax 0 max(max(d_vert1),max(d_vert2))*1.05])
    set(gca,'FontSize', 15);

    subplot(2,2,3)
    plot(xShap1, yShap1, 'b-')
```

```
        title(sprintf('objective = %8.6e', O));
        xlabel('x')
        ylabel('y')
        axis image
        axis([-2 2 -2 2])
        grid on
        set(gca,'FontSize', 15);

        subplot(2,2,4)
        plot(d_horz1, h_rays1, 'b-')
        hold on
        plot(d_horz2, h_rays2, 'r--')
        hold off
        grid on
        title(sprintf('rms_h = %8.6e', rms_h));
        xlabel('width')
        ylabel('y')
        axis square
        axis([0 max(max(d_horz1),max(d_horz2))*1.05 vmin vmax])
        set(gca,'FontSize', 15);

    end %if
end
%function EvaluateObjective

%----------------------------------------------------------------------
```

```
function [ ho_ve ...
              ] = GetXray( xyPoly   , ...
                           d2xyPoly , ...
                           tPoly    , ...
                           poly_fit , ...
                           nrays    , ...
                           hmin     , ...
                           hmax     , ...
                           vmin     , ...
                           vmax      )
    % function GetXray( ) generates the x-rays from a given spline curve
    % for comparison with the desired x-rays.
    %
    % Inputs:
    %                  xyPoly     -> input xy-coordinates for geometry
    %                  d2xyPoly   -> curvature information at xy-
    %                                coordinates
    %                  tPoly      -> parametric coordinate for the
    %                                geometry
    %                  poly_fit   -> type of fit
    %                  nrays      -> number of rays used for x-ray
    %                                calculation
    %                  hmin       -> minimum y-value for horizontal ray
    %                  hmax       -> maximum y-value for horizontal ray
    %                  vmin       -> minimum x-value for vertical ray
    %                  vmax       -> maximum x-value for vertical ray
    %
    % Outputs:
    %                  ho_ve      -> array containing x-ray information of
    %                                given geometry
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    %
    % horizontal ray definition
    %
    h_ray        = ones(nrays,2);
    h_ray(:,1)   = h_ray(:,1) * hmin;
    h_ray(:,2)   = linspace(vmin,vmax,nrays);

    %
    % vertical ray definition
    %
    v_ray        = ones(nrays,2);
    v_ray(:,1)   = linspace(hmin, hmax,nrays);
    v_ray(:,2)   = v_ray(:,2) * vmin;

    ray_type     = 1; % horizontal ray

    %
    % horizontal ray pass
    %
    d_horz = zeros(nrays,1);
    for i = 1 : nrays
        [ xyint  , ...
          icross  ] = raycast_JSR( xyPoly     , ...
                                   d2xyPoly   , ...
                                   tPoly      , ...
                                   poly_fit   , ...
                                   h_ray(i,:) , ...
                                   ray_type   );
    %
    % Rearrange intersection points so first intersection is at the
    % smallest x-value and last intersection is at largest effectively
    % arranging the points in the order the ray passes through the geometry
    %
        if(icross > 2)
        for i1 = 1 : icross
            change  = 0;

            for i2 = 1 : icross-1
                i2p1 = i2+1;
                if(xyint(4*i2-3) > xyint(4*i2p1-3))
                    % swap entries
                    change = change + 1;
                    xtemp  = xyint(4*i2p1-3);
                    ytemp  = xyint(4*i2p1-2);
                    itemp  = xyint(4*i2p1-1);
                    ip1temp= xyint(4*i2p1  );

                    xyint(4*i2p1-3) = xyint(4*i2-3);
                    xyint(4*i2p1-2) = xyint(4*i2-2);
                    xyint(4*i2p1-1) = xyint(4*i2-1);
                    xyint(4*i2p1  ) = xyint(4*i2  );

                    xyint(4*i2 -3)  = xtemp  ;
                    xyint(4*i2 -2)  = ytemp  ;
                    xyint(4*i2 -1)  = itemp  ;
                    xyint(4*i2   )  = ip1temp;
                end %if
            end %for i2

            if(change == 0 || i1 == icross)
                break;
            end %if

        end %for i1
    end %if

    if(mod(icross,2) == 0 && icross ~= 0)
        for j = 1 : icross/2
            ii   = 2*j-1;
            iip1 = ii+1;
            xint1= xyint(4*ii   -3);
            xint2= xyint(4*iip1-3);
            d_horz(i) = d_horz(i) + sqrt((xint1 -xint2)^2);
        end %if
    end %if
end %for i

ray_type    = 2; % vertical ray
%
% Vertical ray pass
%
d_vert = zeros(nrays,1);
for i = 1 : nrays
    [ xyint   , ...
      icross  ] = raycast_JSR( xyPoly    , ...
                               d2xyPoly  , ...
                               tPoly     , ...
                               poly_fit  , ...
                               v_ray(i,:), ...
                               ray_type   );
%
% Rearrange intersection points so first intersection is at the
% smallest x-value and last intersection is at largest effectively
% arranging the points in the order the ray passes through the geometry
%
    if(icross > 2)
        for i1 = 1 : icross
            change  = 0;

            for i2 = 1 : icross-1
                i2p1 = i2+1;
                if(xyint(4*i2-2) > xyint(4*i2p1-2))
```

```
                    % swap entries
                    change = change + 1;
                    xtemp  = xyint(4*i2p1-3);
                    ytemp  = xyint(4*i2p1-2);
                    itemp  = xyint(4*i2p1-1);
                    ip1temp= xyint(4*i2p1  );

                    xyint(4*i2p1-3) = xyint(4*i2-3);
                    xyint(4*i2p1-2) = xyint(4*i2-2);
                    xyint(4*i2p1-1) = xyint(4*i2-1);
                    xyint(4*i2p1  ) = xyint(4*i2  );

                    xyint(4*i2 -3)  = xtemp  ;
                    xyint(4*i2 -2)  = ytemp  ;
                    xyint(4*i2 -1)  = itemp  ;
                    xyint(4*i2   )  = ip1temp;
                end %if
            end %for i2

            if(change == 0 || i1 == icross)
                break;
            end %if

        end %for i1
    end %if

    if(mod(icross,2) == 0 && icross ~= 0)
        for j = 1 : icross/2
            ii   = 2*j-1;
            iip1 = ii+1;
            yint1= xyint(4*ii  -2);
            yint2= xyint(4*iip1-2);
            d_vert(i) = d_vert(i) + sqrt((yint1 - yint2)^2);
        end %if
    end %if
end %for i
%
% Set up an array of the x-rays
%
ho_ve = zeros(4*(nrays+1),1);
for i = 1 : nrays+1
    if(i < nrays+1)
        ho_ve(2*i-1) = h_ray(i,2);
        ho_ve(2*i  ) = d_horz(i);
    elseif( i == nrays+1)
        ho_ve(2*i-1) = NaN;
        ho_ve(2*i  ) = NaN;
    end %if
end %for i

for i = nrays+2 : 2*(nrays+1)
    ii = i -(nrays+1);
    if(i < 2*(nrays+1))
        ho_ve(2*i-1) = v_ray(ii,1);
        ho_ve(2*i  ) = d_vert(ii);
    elseif( i == 2*(nrays+1))
        ho_ve(2*i-1) = NaN;
        ho_ve(2*i  ) = NaN;
    end %if
end %for i

end
%function GetXray

%-----------------------------------------------------------------
```

```
function [ xyint , ...
           icross  ...
                 ] = raycast_JSR( xyPoly   , ...
                                  d2xyPoly , ...
                                  tPoly    , ...
                                  poly_fit , ...
                                  xy       , ...
                                  ray_type )
    % function raycast_JSR( ) casts a ray through a polygon defined by
    % xy+oly from initial point xy. The intersection points along the ray
    % are output. The orientation of the ray is determined by the ray_type
    % variable: 1 for horizontal, 2 for vertical.
    %
    % Inputs:
    %                 xyPoly     -> input xy-coordinates for geometry
    %                 d2xyPoly   -> curvature information at xy-
    %                               coordinates
    %                 tPoly      -> parametric coordinate for the
    %                               geometry
    %                 poly_fit   -> type of fit
    %                 xy         -> xy-coordinate of the ray
    %                 ray_type   -> prescribes either horizontal or
    %                               vertical ray
    %
    % Outputs:
    %                 xyint      -> xy-coordinates for the intersection
    %                               points found
    %                 icross     -> number of crossings for a particular
    %                               ray
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    npts = length(xyPoly)/2;
    %
    % Loop through the segments in the polygon
    %    == Assumes polygon is ordered ==
    %
    icross = 0;
    xyint  = zeros(120,1);
    ibeg   = 1;
    if(ray_type == 1) % horizontal ray
        for i = 1 : npts

            ip1 = i+1;

            if(isnan(xyPoly(2*i-1)) && isnan(xyPoly(2*i)))
                continue;
            end %if

            if(isnan(xyPoly(2*ip1-1)) && isnan(xyPoly(2*ip1)))
                ip1 = ibeg;
                ibeg= i+2;
            end %if

            if(poly_fit == 3)
                if( i ~= ibeg)
                    ts   = [    tPoly(   i   ),    tPoly(   ip1   )];
                elseif(i == ibeg)
                    ts   = [        0        ,    tPoly(   ip1   )];
                end %if
                d2xs = [d2xyPoly(2*i-1), d2xyPoly(2*ip1-1)];
                d2ys = [d2xyPoly(2*i   ), d2xyPoly(2*ip1   )];
            end %if
            xs   = [   xyPoly(2*i-1),    xyPoly(2*ip1-1)];
    %
    % Check if ray crosses segment
    %
```

```
        ys    = [   xyPoly(2*i   ),    xyPoly(2*ip1   )];
%
% Find the max and min y-value on the segment
%
        ymax = -100000000;
        ymin = +100000000;
        for j = 1 : 2
            if(ys(j) > ymax)
                ymax = ys(j);
            end %if
            if(ys(j) < ymin)
                ymin = ys(j);
            end %if
        end %for j
%
% Check if yp is within ymax and ymin
%
        if(ymin < xy(2) && xy(2) <= ymax)
%
% Check where the ray intersects the segment
%
            if(poly_fit == 1 || poly_fit == 2)
            % -> Linear approximation
                if(ys(1) ~= ys(2))
                    yint  = xy(2);
                    t     = (yint - ys(1))/(ys(2) - ys(1));
                    xint  = xs(1) + t*(xs(2) - xs(1));
                elseif(ys(1) == ys(2)) % segment is horiztonal
                    yint  = xy(2);
                    t     = 0;
                    xint  = xs(1) + t*(xs(2) - xs(1));
                    continue
                end %if
            end %if

            icross = icross + 1;
            xyint(4*icross-3) = xint;
            xyint(4*icross-2) = yint;
            xyint(4*icross-1) =  i   ;
            xyint(4*icross   ) =  ip1;
        end %if
    end %for i
elseif(ray_type == 2) % vertical ray
    for i = 1 : npts

        if(isnan(xyPoly(2*i-1)) && isnan(xyPoly(2*i)))
            continue;
        end %if

        ip1 = i+1;

        if(isnan(xyPoly(2*ip1-1)) && isnan(xyPoly(2*ip1)))
            ip1 = ibeg;
            ibeg= i+2;
        end %if


        if(poly_fit == 3)
            if( i ~= ibeg)
                ts    = [     tPoly(   i    ),     tPoly(   ip1   )];
            elseif(i == ibeg)
                ts    = [          0          ,     tPoly(   ip1   )];
            end %if
            d2xs = [d2xyPoly(2*i-1), d2xyPoly(2*ip1-1)];
            d2ys = [d2xyPoly(2*i   ), d2xyPoly(2*ip1   )];
        end %if
        xs    = [   xyPoly(2*i-1),    xyPoly(2*ip1-1)];
%
```

```
    % Check if ray crosses segment
    %
            ys    = [   xyPoly(2*i   ),    xyPoly(2*ip1   )];
    %
    % Find the max and min x-value on the segment
    %
            xmax = -100000000;
            xmin = +100000000;
            for j = 1 : 2
                if(xs(j) > xmax)
                    xmax = xs(j);
                end %if
                if(xs(j) < xmin)
                    xmin = xs(j);
                end %if
            end %for j
    %
    % Check if xp is within ymax and ymin
    %
            if(xmin < xy(1) && xy(1) <= xmax)
    %
    % Check where the ray intersects the segment
    %
                if(poly_fit == 1 || poly_fit == 2)
                % -> Linear
                    if(xs(1) ~= xs(2))
                        xint  = xy(1);
                        t     = (xint - xs(1))/(xs(2) - xs(1));
                        yint  = ys(1) + t*(ys(2) - ys(1));
                    elseif(xs(1) == xs(2)) % segment is vertical
                        xint  = xy(1);
                        t     = 0;
                        yint  = ys(1) + t*(ys(2) - ys(1));
                        continue
                    end %if
                end %if
                icross = icross + 1;
                xyint(4*icross-3) = xint;
                xyint(4*icross-2) = yint;
                xyint(4*icross-1) =  i   ;
                xyint(4*icross   ) =  ip1;
            end %if
        end %for i
    end %if
end
%function raycast_JSR

%-------------------------------------------------------------------------
```

```
function [ O     , ...
           O_dot ] = EvaluateObjective_Dot( xyPoly        , ...
                                            xyPoly_dot    , ...
                                            d2xyPoly      , ...
                                            d2xyPoly_dot , ...
                                            tPoly         , ...
                                            tPoly_dot     , ...
                                            poly_fit      , ...
                                            xrays_desired , ...
                                            NRML          , ...
                                            nrays         , ...
                                            hmin          , ...
                                            hmax          , ...
                                            vmin          , ...
                                            vmax          , ...
                                            plotting      )
    % function EvaluateObjective_Dot( ) evaluates the tangent linear
    % approximation of the derivative of the objective function given the
    % spline points, curvature, and parametric distribution of the points
    % and their respective derivitives w.r.t. the design variables.
    %
    % Inputs:
    %                  xyPoly       -> input xy-coordinates for geometry
    %                  xyPoly_dot   -> input derivative of the xy-
    %                                  coordinates for geometry
    %                  d2xyPoly     -> curvature information at xy-
    %                                  coordinates
    %                  d2xyPoly_dot -> derivative of the curvature
    %                                  information at xy-coordinates
    %                  tPoly        -> parametric coordinate for the
    %                                  geometry
    %                  tPoly_dot    -> derivative of the parametric
    %                                  coordinate for the geometry
    %                  poly_fit     -> type of fit
    %                  xrays_des    -> array containing the x-ray data to
    %                                  be matched
    %                  NRML         -> boolean argument prescribing
    %                                  whether the objective is normalized
    %                                  or not
    %                  nrays        -> number of rays used for x-ray
    %                                  calculation
    %                  hmin         -> minimum y-value for horizontal ray
    %                  hmax         -> maximum y-value for horizontal ray
    %                  vmin         -> minimum x-value for vertical ray
    %                  vmax         -> maximum x-value for vertical ray
    %                  plotting     -> prescribes whether to plot the
    %                                  objective function results
    %
    % Outputs:
    %                  O            -> the objective function value
    %                  O_dot        -> the derivative of the objective
    %                                  function value w.r.t. the design
    %                                  parameters
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    mpts1 = length(xyPoly)/2;
    xShap1= zeros(mpts1, 1);
    yShap1= zeros(mpts1, 1);
    for ipt = 1 : mpts1
        xShap1(ipt) = xyPoly(2*ipt-1);
        yShap1(ipt) = xyPoly(2*ipt  );
    end %for ipt

    [ ho_ve     , ...
      ho_ve_dot ] = GetXray_Dot( xyPoly       , ...
                                 xyPoly_dot   , ...
```

```
                                    d2xyPoly     , ...
                                    d2xyPoly_dot , ...
                                    tPoly        , ...
                                    tPoly_dot    , ...
                                    poly_fit     , ...
                                    nrays        , ...
                                    hmin         , ...
                                    hmax         , ...
                                    vmin         , ...
                                    vmax          );

h_rays1     = ones(nrays+1,1) * NaN;
d_horz1     = ones(nrays+1,1) * NaN;
v_rays1     = ones(nrays+1,1) * NaN;
d_vert1     = ones(nrays+1,1) * NaN;
d_horz1_dot = ones(nrays+1,1) * NaN;
d_vert1_dot = ones(nrays+1,1) * NaN;
ray_type = 1; % indicates the ray being unzipped
ii = 0;
for i = 1 : 2*(nrays+1)
    if(isnan(ho_ve(2*i-1)) && isnan(ho_ve(2*i)))
        ray_type = 2;
        ii       = 0; % restart counter
        continue;
    end %if

    if(ray_type == 1)
        ii             = ii + 1;
        h_rays1(ii)    = ho_ve(2*i-1);
        d_horz1(ii)    = ho_ve(2*i  );
        d_horz1_dot(ii)= ho_ve_dot(2*i  );
    elseif(ray_type == 2)
        ii             = ii + 1;
        v_rays1(ii)    = ho_ve(2*i-1);
        d_vert1(ii)    = ho_ve(2*i  );
        d_vert1_dot(ii)= ho_ve_dot(2*i  );
    end %if

end %for i

h_rays2 = ones(nrays+1,1) * NaN;
d_horz2 = ones(nrays+1,1) * NaN;
v_rays2 = ones(nrays+1,1) * NaN;
d_vert2 = ones(nrays+1,1) * NaN;
ray_type= 1; % indicates the ray being unzipped
ii = 0;
for i = 1 : 2*(nrays+1)
    if(isnan(xrays_desired(2*i-1)) && isnan(xrays_desired(2*i)))
        ray_type = 2;
        ii       = 0; % restart counter
        continue;
    end %if

    if(ray_type == 1)
        ii         = ii + 1;
        h_rays2(ii)= xrays_desired(2*i-1);
        d_horz2(ii)= xrays_desired(2*i  );
    elseif(ray_type == 2)
        ii         = ii + 1;
        v_rays2(ii)= xrays_desired(2*i-1);
        d_vert2(ii)= xrays_desired(2*i  );
    end %if

end %for i
%
% Calculate the RMS of the differences between each x-ray
%
dx        = v_rays1(2) - v_rays1(1);
```

```
dy         = h_rays1(2) - h_rays1(1);
sqsumh     = 0.0;
sqsumh_dot= 0.0;
sqsumv     = 0.0;
sqsumv_dot= 0.0;

for i = 1 : nrays

    dh2   = ((d_horz1(i) - d_horz2(i)))^2;
    dv2   = ((d_vert1(i) - d_vert2(i)))^2;

    dv2_dot = 2*(d_vert1(i) - d_vert2(i)) * d_vert1_dot(i);
    dh2_dot = 2*(d_horz1(i) - d_horz2(i)) * d_horz1_dot(i);

    sqsumh     = sqsumh      + dh2     ;
    sqsumh_dot= sqsumh_dot + dh2_dot;

    sqsumv     = sqsumv      + dv2     ;
    sqsumv_dot= sqsumv_dot + dv2_dot;

end %for i

if(NRML == 0)
    ch = sqrt(dy/nrays);
    cv = sqrt(dx/nrays);
elseif(NRML == 1)
    ch = sqrt(dy/(nrays*max(d_horz2)^2));
    cv = sqrt(dx/(nrays*max(d_vert2)^2));
end %if

rms_h = ch*sqrt(sqsumh^2);
rms_v = cv*sqrt(sqsumv^2);
term1 = rms_h + rms_v;
O      = term1;

rms_h_dot = ch*sqsumh*sqsumh_dot/(sqrt(sqsumh^2));
rms_v_dot = cv*sqsumv*sqsumv_dot/(sqrt(sqsumv^2));
O_dot      = rms_h_dot + rms_v_dot;

if( plotting == 1  && NRML == 1)

    figure(45322)
    set(gcf,'units', 'normalized', 'position', [0.0536 0.2269 0.3849 0.6194])

    subplot(2,2,1)
    plot(v_rays1, d_vert1/max(d_vert2), 'b-')
    hold on
    plot(v_rays2, d_vert2/max(d_vert2), 'r--')
    hold off
    grid on
    title(sprintf('rms_v = %8.6e', rms_v));
    xlabel(sprintf('term1 = %+10.6f', term1))
    ylabel('height')
    axis([-3.5 3.5 0 max(max(d_vert1),max(d_vert2))/max(d_vert2)*1.05])
    set(gca,'FontSize', 15);

    subplot(2,2,3)
    plot(xShap1, yShap1, 'b-')
    title(sprintf('objective = %10.6f', O));
    xlabel('x')
    ylabel('y')
    axis image
    axis([-3.5 3.5 -3.5 3.5])
    grid on
    set(gca,'FontSize', 15);

    subplot(2,2,4)
    plot(d_horz1/max(d_horz2), h_rays1, 'b-')
```

```
        hold on
        plot(d_horz2/max(d_horz2), h_rays2, 'r--')
        hold off
        grid on
        title(sprintf('rms_h = %8.6e', rms_h));
        ylabel('y')
        axis([0 max(max(d_horz1),max(d_horz2))/max(d_horz2)*1.05 -3.5 3.5])
        set(gca,'FontSize', 15);

    elseif( plotting == 1 && NRML == 0)

        figure(45322)
        set(gcf,'units', 'normalized', 'position', [0.0536 0.2269 0.3849 0.6194])

        subplot(2,2,1)
        plot(v_rays1, d_vert1, 'b-')
        hold on
        plot(v_rays2, d_vert2, 'r--')
        hold off
        grid on
        title(sprintf('rms_v = %8.6e', rms_v));
        xlabel(sprintf('term1 = %+10.6f', term1))
        ylabel('height')
        axis([-3.5 3.5 0 max(max(d_vert1),max(d_vert2))*1.05])
        set(gca,'FontSize', 15);

        subplot(2,2,3)
        plot(xShap1, yShap1, 'b-')
        title(sprintf('objective = %10.6f', O));
        xlabel('x')
        ylabel('y')
        axis image
        axis([-3.5 3.5 -3.5 3.5])
        grid on
        set(gca,'FontSize', 15);

        subplot(2,2,4)
        plot(d_horz1, h_rays1, 'b-')
        hold on
        plot(d_horz2, h_rays2, 'r--')
        hold off
        grid on
        title(sprintf('rms_h = %8.6e', rms_h));
        ylabel('y')
        axis([0 max(max(d_horz1),max(d_horz2))*1.05 -3.5 3.5])
        set(gca,'FontSize', 15);

    end %if
end
%function EvaluateObjective_Dot

%---------------------------------------------------------------------------
```

```
function [ ho_ve     , ...
           ho_ve_dot ] = GetXray_Dot( xyPoly       , ...
                                      xyPoly_dot   , ...
                                      d2xyPoly     , ...
                                      d2xyPoly_dot , ...
                                      tPoly        , ...
                                      tPoly_dot    , ...
                                      poly_fit     , ...
                                      nrays        , ...
                                      hmin         , ...
                                      hmax         , ...
                                      vmin         , ...
                                      vmax          )
    % function GetXray_Dot( ) evaluates the tangent linear approximation
    % of the derivative of the x-ray calculates given the spline points,
    % curvature, and parametric distribution of the points and their
    % respective derivitives w.r.t. the design variables.
    %
    % Inputs:
    %                  xyPoly       -> input xy-coordinates for geometry
    %                  xyPoly_dot   -> input derivative of the xy-
    %                                  coordinates for geometry
    %                  d2xyPoly     -> curvature information at xy-
    %                                  coordinates
    %                  d2xyPoly_dot -> derivative of the curvature
    %                                  information at xy-coordinates
    %                  tPoly        -> parametric coordinate for the
    %                                  geometry
    %                  tPoly_dot    -> derivative of the parametric
    %                                  coordinate for the geometry
    %                  poly_fit     -> type of fit
    %                  nrays        -> number of rays used for x-ray
    %                                  calculation
    %                  hmin         -> minimum y-value for horizontal ray
    %                  hmax         -> maximum y-value for horizontal ray
    %                  vmin         -> minimum x-value for vertical ray
    %                  vmax         -> maximum x-value for vertical ray
    %
    % Outputs:
    %                  ho_ve      -> array containing x-ray information of
    %                                given geometry
    %                  ho_ve_dot  -> array containing derivative of the
    %                                x-rays w.r.t. the design variables
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    %
    % horizontal ray definition
    %
    h_ray       = ones(nrays,2);
    h_ray(:,1)  = h_ray(:,1) * hmin;
    h_ray(:,2)  = linspace(vmin,vmax,nrays);
    %
    % vertical ray definition
    %
    v_ray       = ones(nrays,2);
    v_ray(:,1)  = linspace(hmin, hmax,nrays);
    v_ray(:,2)  = v_ray(:,2) * vmin;

    ray_type    = 1; % horizontal ray
    %
    % Horizontal pass
    %
    d_horz     = zeros(nrays,1);
    d_horz_dot = zeros(nrays,1);
    for i = 1 : nrays
        [ xyint    , ...
```

```
      xyint_dot , ...
      icross    ] = raycast_JSR_Dot( xyPoly       , ...
                                     xyPoly_dot   , ...
                                     d2xyPoly     , ...
                                     d2xyPoly_dot , ...
                                     tPoly        , ...
                                     tPoly_dot    , ...
                                     poly_fit     , ...
                                     h_ray(i,:)   , ...
                                     ray_type      );
%
% Rearrange intersection points so first intersection is at
% the smallest x-value and last intersection is at largest
% effectively arranging the points in the order the ray passes
% through the geometry.
%
    if(icross > 2)
        for i1 = 1 : icross
            change  = 0;

            for i2 = 1 : icross-1
                i2p1 = i2+1;
                if(xyint(4*i2-3) > xyint(4*i2p1-3))
                    % swap entries
                    change = change + 1;
                    xtemp  = xyint(4*i2p1-3);
                    ytemp  = xyint(4*i2p1-2);
                    itemp  = xyint(4*i2p1-1);
                    ip1temp= xyint(4*i2p1  );

                    xyint(4*i2p1-3) = xyint(4*i2-3);
                    xyint(4*i2p1-2) = xyint(4*i2-2);
                    xyint(4*i2p1-1) = xyint(4*i2-1);
                    xyint(4*i2p1  ) = xyint(4*i2  );

                    xyint(4*i2 -3)  = xtemp   ;
                    xyint(4*i2 -2)  = ytemp   ;
                    xyint(4*i2 -1)  = itemp   ;
                    xyint(4*i2   )  = ip1temp;

                    xtemp_dot  = xyint_dot(4*i2p1-3);
                    ytemp_dot  = xyint_dot(4*i2p1-2);
                    itemp_dot  = xyint_dot(4*i2p1-1);
                    ip1temp_dot= xyint_dot(4*i2p1  );

                    xyint_dot(4*i2p1-3) = xyint_dot(4*i2-3);
                    xyint_dot(4*i2p1-2) = xyint_dot(4*i2-2);
                    xyint_dot(4*i2p1-1) = xyint_dot(4*i2-1);
                    xyint_dot(4*i2p1  ) = xyint_dot(4*i2  );

                    xyint_dot(4*i2 -3)  = xtemp_dot   ;
                    xyint_dot(4*i2 -2)  = ytemp_dot   ;
                    xyint_dot(4*i2 -1)  = itemp_dot   ;
                    xyint_dot(4*i2   )  = ip1temp_dot;
                end %if
            end %for i2

            if(change == 0 || i1 == icross)
                break;
            end %if

        end %for i1
    end %if

    if(mod(icross,2) == 0 && icross ~= 0)
        for j = 1 : icross/2
            ii   = 2*j-1;
            iip1 = ii+1;
            xint1    = xyint(4*ii   -3);
            xint2    = xyint(4*iip1-3);
            xint1_dot= xyint_dot(4*ii   -3);
            xint2_dot= xyint_dot(4*iip1-3);
            d_horz(i)     = d_horz(i)     + sqrt((xint1 -xint2)^2);
            d_horz_dot(i) = d_horz_dot(i) + ((xint1 -xint2)*(xint1_dot -xint2_dot))/sqrt
                ((xint1 -xint2)^2);
        end %if
    end %if
end %for i

ray_type    = 2; % vertical ray

% Vertical pass
d_vert     = zeros(nrays,1);
d_vert_dot = zeros(nrays,1);
for i = 1 : nrays
    [ xyint     , ...
      xyint_dot , ...
      icross    ] = raycast_JSR_Dot( xyPoly      , ...
                                     xyPoly_dot  , ...
                                     d2xyPoly    , ...
                                     d2xyPoly_dot, ...
                                     tPoly       , ...
                                     tPoly_dot   , ...
                                     poly_fit    , ...
                                     v_ray(i,:)  , ...
                                     ray_type     );
%
% Rearrange intersection points so first intersection is at
% the smallest y-value and last intersection is at largest
% effectively arranging the points in the order the ray passes
% through the geometry.
%
    if(icross > 2)
        for i1 = 1 : icross
            change  = 0;

            for i2 = 1 : icross-1
                i2p1 = i2+1;
                if(xyint(4*i2-2) > xyint(4*i2p1-2))
                    % swap entries
                    change = change + 1;
                    xtemp  = xyint(4*i2p1-3);
                    ytemp  = xyint(4*i2p1-2);
                    itemp  = xyint(4*i2p1-1);
                    ip1temp= xyint(4*i2p1  );

                    xyint(4*i2p1-3) = xyint(4*i2-3);
                    xyint(4*i2p1-2) = xyint(4*i2-2);
                    xyint(4*i2p1-1) = xyint(4*i2-1);
                    xyint(4*i2p1  ) = xyint(4*i2  );

                    xyint(4*i2 -3)  = xtemp  ;
                    xyint(4*i2 -2)  = ytemp  ;
                    xyint(4*i2 -1)  = itemp  ;
                    xyint(4*i2   )  = ip1temp;

                    xtemp_dot  = xyint_dot(4*i2p1-3);
                    ytemp_dot  = xyint_dot(4*i2p1-2);
                    itemp_dot  = xyint_dot(4*i2p1-1);
                    ip1temp_dot= xyint_dot(4*i2p1  );

                    xyint_dot(4*i2p1-3) = xyint_dot(4*i2-3);
                    xyint_dot(4*i2p1-2) = xyint_dot(4*i2-2);
                    xyint_dot(4*i2p1-1) = xyint_dot(4*i2-1);
                    xyint_dot(4*i2p1  ) = xyint_dot(4*i2  );
```

```
                        xyint_dot(4*i2 -3)  = xtemp_dot   ;
                        xyint_dot(4*i2 -2)  = ytemp_dot   ;
                        xyint_dot(4*i2 -1)  = itemp_dot   ;
                        xyint_dot(4*i2    )  = ip1temp_dot;
                    end %if
                end %for i2

                if(change == 0 || i1 == icross)
                    break;
                end %if

            end %for i1
        end %if

        if(mod(icross,2) == 0 && icross ~= 0)
            for j = 1 : icross/2
                ii   = 2*j-1;
                iip1 = ii+1;
                yint1    = xyint(4*ii   -2);
                yint2    = xyint(4*iip1-2);
                yint1_dot= xyint_dot(4*ii   -2);
                yint2_dot= xyint_dot(4*iip1-2);
                d_vert(i)     = d_vert(i) + sqrt((yint1 - yint2)^2);
                d_vert_dot(i) = d_vert_dot(i) + ((yint1 - yint2)*(yint1_dot - yint2_dot))/
                    sqrt((yint1 - yint2)^2);
            end %if
        end %if
    end %for i
    %
    % Set up an array of the x-rays
    %
    ho_ve = zeros(4*(nrays+1),1);
    ho_ve_dot = zeros(4*(nrays+1),1);
    for i = 1 : nrays+1
        if(i < nrays+1)
            ho_ve(2*i-1) = h_ray(i,2);
            ho_ve(2*i   ) = d_horz(i);
            ho_ve_dot(2*i-1) = h_ray(i,2);
            ho_ve_dot(2*i   ) = d_horz_dot(i);
        elseif( i == nrays+1)
            ho_ve(2*i-1) = NaN;
            ho_ve(2*i   ) = NaN;
            ho_ve_dot(2*i-1) = NaN;
            ho_ve_dot(2*i   ) = NaN;
        end %if
    end %for i

    for i = nrays+2 : 2*(nrays+1)
        ii = i -(nrays+1);
        if(i < 2*(nrays+1))
            ho_ve(2*i-1) = v_ray(ii,1);
            ho_ve(2*i   ) = d_vert(ii);
            ho_ve_dot(2*i-1) = v_ray(ii,1);
            ho_ve_dot(2*i   ) = d_vert_dot(ii);
        elseif( i == 2*(nrays+1))
            ho_ve(2*i-1) = NaN;
            ho_ve(2*i   ) = NaN;
            ho_ve_dot(2*i-1) = NaN;
            ho_ve_dot(2*i   ) = NaN;
        end %if
    end %for i

end
%function GetXray_Dot

%----------------------------------------------------------------------
```

```
function [ xyint     , ...
           xyint_dot , ...
           icross    ] = raycast_JSR_Dot( xyPoly      , ...
                                          xyPoly_dot  , ...
                                          d2xyPoly    , ...
                                          d2xyPoly_dot, ...
                                          tPoly       , ...
                                          tPoly_dot   , ...
                                          poly_fit    , ...
                                          xy          , ...
                                          ray_type     )
    % function raycast_JSR_Dot( ) evaluates the tangent linear
    % approximation of the derivative of the intersection points calculated
    % from the given spline points, curvature, and parametric distribution
    % of the points and their respective derivitives w.r.t. the design
    % variables.
    %
    % Inputs:
    %                  xyPoly       -> input xy-coordinates for geometry
    %                  xyPoly_dot   -> input derivative of the xy-
    %                                  coordinates for geometry
    %                  d2xyPoly     -> curvature information at xy-
    %                                  coordinates
    %                  d2xyPoly_dot -> derivative of the curvature
    %                                  information at xy-coordinates
    %                  tPoly        -> parametric coordinate for the
    %                                  geometry
    %                  tPoly_dot    -> derivative of the parametric
    %                                  coordinate for the geometry
    %                  poly_fit     -> type of fit
    %                  xy           -> xy-coordinate of the ray
    %                  ray_type     -> prescribes either horizontal or
    %                                  vertical ray
    %
    % Outputs:
    %                  xyint        -> xy-coordinates for the intersection
    %                                  points found
    %                  xyint_dot    -> derivative of the xy-coordinates
    %                                  for the intersection points found
    %                                  w.r.t. the design variables
    %                  icross       -> number of crossings for a
    %                                  particular ray
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti__________________________________02/24/20


    npts = length(xyPoly)/2;
    %
    % Loop through the segments in the polygon
    %   == Assumes polygon is ordered ==
    %
    icross = 0;
    xyint      = zeros(120,1);
    xyint_dot  = zeros(120,1);
    ibeg   = 1;

    if(ray_type == 1) % horizontal ray
        for i = 1 : npts

            ip1 = i+1;

            if(isnan(xyPoly(2*i-1)) && isnan(xyPoly(2*i)))
                continue;
            end %if

            if(isnan(xyPoly(2*ip1-1)) && isnan(xyPoly(2*ip1)))
                ip1 = ibeg;
            ibeg= i+2;
        end %if

        if( i ~= ibeg)
            ts      = [    tPoly(   i   )      ,     tPoly(   ip1   )      ];
            ts_dot  = [    tPoly_dot(   i   ),     tPoly_dot(   ip1   )];
        elseif(i == ibeg)
            ts      = [          0              ,     tPoly(   ip1   )      ];
            ts_dot  = [          0              ,     tPoly_dot(   ip1   )];
        end %if

        xs        = [xyPoly(2*i-1)        , xyPoly(2*ip1-1)      ];
        xs_dot    = [xyPoly_dot(2*i-1)    , xyPoly_dot(2*ip1-1)];
        d2xs      = [d2xyPoly(2*i-1)      , d2xyPoly(2*ip1-1)      ];
        d2xs_dot  = [d2xyPoly_dot(2*i-1), d2xyPoly_dot(2*ip1-1)];
%
% Check if ray crosses segment
%
        ys        = [xyPoly(2*i   )        , xyPoly(2*ip1   )      ];
        ys_dot    = [xyPoly_dot(2*i   )    , xyPoly_dot(2*ip1   )];
        d2ys      = [d2xyPoly(2*i   )      , d2xyPoly(2*ip1   )      ];
        d2ys_dot  = [d2xyPoly_dot(2*i   ), d2xyPoly_dot(2*ip1   )];
%
% Find the max and min y-value on the segment
%
        ymax = -100000000;
        ymin = +100000000;
        for j = 1 : 2
            if(ys(j) > ymax)
                ymax = ys(j);
            end %if
            if(ys(j) < ymin)
                ymin = ys(j);
            end %if
        end %for j
%
% Check if yp is within ymax and ymin
%
        if(ymin < xy(2) && xy(2) <= ymax)
%
% Check where the ray intersects the segment
%
            if(poly_fit == 1 || poly_fit == 2)
            % -> Linear approximation
                if(ys(1) ~= ys(2))
                    yint     = xy(2);
                    t        = (yint - ys(1))/(ys(2) - ys(1));
                    xint     = xs(1) + t*(xs(2) - xs(1));
                    yint_dot = 0.0;
                    t_dot    = ((yint_dot - ys_dot(1))*(ys(2) - ys(1)) - (yint - ys(1))
                        *(ys_dot(2) - ys_dot(1)))...
                                /(ys(2) - ys(1))^2;
                    xint_dot = xs_dot(1) + t_dot*(xs(2) - xs(1)) + t*(xs_dot(2) - xs_dot
                        (1));
                elseif(ys(1) == ys(2)) % segment is horiztonal
                    yint     = xy(2);
                    t        = 0;
                    xint     = xs(1) + t*(xs(2) - xs(1));
                    yint_dot = 0.0;
                    t_dot    = 0.0;
                    xint_dot = xs_dot(1) + t_dot*(xs(2) - xs(1)) + t*(xs_dot(2) - xs_dot
                        (1));
                    continue
                end %if
            end %if

            icross = icross + 1;
            xyint(4*icross-3) = xint;
            xyint(4*icross-2) = yint;
            xyint(4*icross-1) =  i   ;
            xyint(4*icross  ) =  ip1;

            xyint_dot(4*icross-3) = xint_dot;
            xyint_dot(4*icross-2) =     0;
            xyint_dot(4*icross-1) =  i   ;
            xyint_dot(4*icross  ) =  ip1;
        end %if
    end %for i
elseif(ray_type == 2) % vertical ray
    for i = 1 : npts

        if(isnan(xyPoly(2*i-1)) && isnan(xyPoly(2*i)))
            continue;
        end %if

        ip1 = i+1;

        if(isnan(xyPoly(2*ip1-1)) && isnan(xyPoly(2*ip1)))
            ip1 = ibeg;
            ibeg= i+2;
        end %if

        if( i ~= ibeg)
            ts     = [   tPoly(  i   )     ,    tPoly(  ip1  )     ];
            ts_dot = [   tPoly_dot(  i   ),    tPoly_dot(  ip1  )];
        elseif(i == ibeg)
            ts     = [        0           ,    tPoly(  ip1  )     ];
            ts_dot = [        0           ,    tPoly_dot(  ip1  )];
        end %if

        xs       = [  xyPoly(2*i-1)      ,   xyPoly(2*ip1-1)      ];
        xs_dot   = [  xyPoly_dot(2*i-1),   xyPoly_dot(2*ip1-1)];
        d2xs     = [d2xyPoly(2*i-1)      , d2xyPoly(2*ip1-1)      ];
        d2xs_dot = [d2xyPoly_dot(2*i-1), d2xyPoly_dot(2*ip1-1)];
%
% Check if ray crosses segment
%
        ys       = [  xyPoly(2*i   )     ,   xyPoly(2*ip1   )     ];
        ys_dot   = [  xyPoly_dot(2*i   ),   xyPoly_dot(2*ip1   )];
        d2ys     = [d2xyPoly(2*i   )     , d2xyPoly(2*ip1   )     ];
        d2ys_dot = [d2xyPoly_dot(2*i   ), d2xyPoly_dot(2*ip1   )];
%
% Find the max and min x-value on the segment
%
        xmax = -100000000;
        xmin = +100000000;
        for j = 1 : 2
            if(xs(j) > xmax)
                xmax = xs(j);
            end %if
            if(xs(j) < xmin)
                xmin = xs(j);
            end %if
        end %for j
%
% Check if xp is within xmax and xmin
%
        if(xmin < xy(1) && xy(1) <= xmax)
%
% Check where the ray intersects the segment
%
            if(poly_fit == 1 || poly_fit == 2)
            % -> Linear approximation
                if(xs(1) ~= xs(2))
                    xint     = xy(1);
                    t        = (xint - xs(1))/(xs(2) - xs(1));
                    yint     = ys(1) + t*(ys(2) - ys(1));
                    xint_dot = 0.0;
                    t_dot    = ((xint_dot - xs_dot(1))*(xs(2) - xs(1)) - (xint - xs(1))
                        *(xs_dot(2) - xs_dot(1)))...
                                /(xs(2) - xs(1))^2;
                    yint_dot = ys_dot(1) + t_dot*(ys(2) - ys(1)) + t*(ys_dot(2) - ys_dot
                        (1));
                elseif(xs(1) == xs(2)) % segment is vertical
                    yint     = xy(2);
                    t        = 0;
                    t_dot    = 0.0;
                    xint     = xs(1) + t*(xs(2) - xs(1));
                    yint_dot = ys_dot(1) + t_dot*(ys(2) - ys(1)) + t*(ys_dot(2) - ys_dot
                        (1));
                    continue
                end %if
            end %if

            icross = icross + 1;
            xyint(4*icross-3) = xint;
            xyint(4*icross-2) = yint;
            xyint(4*icross-1) =  i   ;
            xyint(4*icross  ) =  ip1;

            xyint_dot(4*icross-3) =     0.0   ;
            xyint_dot(4*icross-2) = yint_dot;
            xyint_dot(4*icross-1) =  i        ;
            xyint_dot(4*icross  ) =  ip1      ;
        end %if
    end %for i
  end %if
end
%function raycast_JSR_Dot

%--------------------------------------------------------------------
```

```
function [ x ...
         ] = SquareMatrixSolver( A      , ...
                                 b      , ...
                                 DEBUG )
    % function SquareMatrixSolver(A, b, DEBUG) solves a square system of
    % equations using LU Decomposition.
    %
    % Inputs:
    %                  A          ->  A square coefficient matrix
    %                  b          ->  The solution vector of the system
    %                  DEBUG      ->  Check matrix solutions with MATLAB
    %
    % Outputs:
    %                  x          -> The solution to the system
    %
    % Written by Jack Rossetti_________________________________07/03/19
    % Annotated by Jack Rossetti_______________________________02/24/20

    if(nargin < 1)
        A = [     2,  3,  1,  5    ;...
                  1,  1,  5,  2    ;...
                  3,  1,  4,  1    ;...
                  7,  5,  1,  1    ];

        b = [    31   ;...
                 26   ;...
                 21   ;...
                 24   ];

        DEBUG = 0;

    end %if

    [M,N]          = size(A); %Number of rows, M, and number of columns, N.
    [LU, indx] = LU_Decomposition( A );

    if(DEBUG == 1)
        fprintf(1,'a consists of %d rows and %d columns.\n',M,N);
        %--- Check with MATLAB's lu function ---%
        [L,U,P] = lu(A);

        fprintf(1,'\n[L]\n');
        fprintf(1,'\n');
        for i = 1 : M
            fprintf(1,'[');
            for j = 1 : N
                fprintf(1,' %+10.2f  ', L(i,j));
            end %for j
            fprintf(1,']\n');
        end %for i

        fprintf(1,'\n[U]\n');
        fprintf(1,'\n');
        for i = 1 : M
            fprintf(1,'[');
            for j = 1 : N
                fprintf(1,' %+10.2f  ', U(i,j));
            end %for j
            fprintf(1,']\n');
        end %for i

        fprintf(1,'\n[LU]\n');
        fprintf(1,'\n');
        for i = 1 : M
            fprintf(1,'[');
            for j = 1 : N
                fprintf(1,' %+10.2f  ', LU(i,j));
            end %for j
            fprintf(1,']\n');
        end %for i

        fprintf(1,'\n[P]\n');
        fprintf(1,'\n');
        for i = 1 : M
            fprintf(1,'[');
            for j = 1 : N
                fprintf(1,' %+10.2f ', P(i,j));
            end %for j
            fprintf(1,']\n');
        end %for i
        pause
    end %if

    x = zeros(size(b));

    if(length(x) ~= length(b))
        error('x and b should be vectors of the same length');
    end %if

    x  = b;
    ii = 0.0; % For efficiency if b is full of zeroes

    %—— Forward substitution ——%
    for i = 1 : N
        ip     = indx(i);
        sum    = x(ip);
        x(ip) = x(i);
        if(ii ~= 0)
            for j = ii-1 : i-1
                sum = sum - LU(i,j)*x(j);
            end %for j
        elseif(sum ~= 0.0)
            ii = i+1;
        end %if
        x(i) = sum;
    end %for i

    %—— Backward substitution ——%
    for i =  N : -1 : 1
        sum = x(i);
        for j = i+1 : N
            sum = sum - LU(i,j)*x(j);
        end %for j
        x(i) = sum/LU(i,i);
    end %for i

    if(DEBUG == 1)
        %—— Check with MATLAB ——%
        xcheck = A\b;
        for i = 1 : N
            fprintf(1,'x(%d) = %+8.2f, xcheck(%d) = %+8.2f,\n', i, x(i), i, xcheck(i));
        end %for i
    end %if

end
%function SquareMatrixSolver

%————————————————————————————————————————
```

```
function [ LU   , ...
           indx   ...
                ] = LU_Decomposition( A )
    % function LU_Decomposition( A ) factorizes a square system of
    % equations using LU Decomposition.
    %
    % Inputs:
    %                   A           ->  A square coefficient matrix
    %
    % Outputs:
    %                   LU          -> The factorization in one matrix
    %                   indx        -> An index vector to unscramble the
    %                                  solution vector so the variables
    %                                  solved for are in the proper order
    %                                  and no post-processing is required.
    %
    % Written by Jack Rossetti___________________________________07/03/19
    % Annotated by Jack Rossetti_________________________________02/24/20

    %--- Perform partial pivoting ---%
    d = 1; % Determines whether the number of row switches is even or odd
    TINY = eps;
    [M,N] = size(A); %Number of rows, M, and number of columns, N.

    LU = A;
    %--- Loop over each row to determine implicit scaling factors ---%
    vv = zeros(M,1); % Vector of scaling variables
    for i = 1 : M
        big = 0.0; % A possibly big number
        for j = 1 : N
            temp = abs(LU(i,j));
            if(temp > big) % Determine the largest value in the row
                big = temp;
            end %if
        end %for j
        vv(i) = (1/big); % Update scaling vector
    end %for i

    %--- Search for largest pivot ---%
    indx = zeros(M,1);
    for k = 1 : M
        big = 0.0;
        for i = k : N % from the diagonal element on in each row
            temp = vv(k)*abs(LU(i,k));
            if(temp > big)
                big = temp; % Update the largest value
                imax= i;    % Update the row where the largest value is
            end %if
        end %for i
        if(k ~= imax) % The maximum value is not on the diagonal
                      % Swap the rows
            for j = 1 : M
                temp       = LU(imax, j); % The row to be swapped
                LU(imax,j) = LU(k,j);     % The kth row being swapped
                LU(k,j)    = temp;        % Complete the swap
            end %for j
            d          = -d;    % Switch the parity, indicating a row swap
            vv(imax) = vv(k); % Also swap scale factors
        end %if
        indx(k) = imax;

        %--- Handle singular martices ---%
        if(LU(k,k) == 0.0)
            LU(k,k) = TINY;
        end %if

        %--- Divide by pivot element ---%
        for i = k+1 : N
```

```
            LU(i,k) = LU(i,k) / LU(k,k);
            temp    = LU(i,k);
            for j = k+1 : N
                LU(i,j) = LU(i,j) - temp*LU(k,j);
            end %for j
        end %for i

    end %for k

end
%function LU_Decomposition

%-----------------------------------------------------------------------
```

```
function dOdp = finite_diff_sensitivities( xRBF      , ...
                                           yRBF      , ...
                                           aRBF      , ...
                                           SR        , ...
                                           nbdy      , ...
                                           poly_fit  , ...
                                           xySpln    , ...
                                           xyCurv    , ...
                                           xyTpar    , ...
                                           xrays_des , ...
                                           nrays     , ...
                                           NRML      , ...
                                           hmin      , ...
                                           hmax      , ...
                                           vmin      , ...
                                           vmax       )
    % function finite_diff_sensitivities( ) calculates the design
    % sensitivities of the RBF locations and coefficients w.r.t. a given
    % objective function using finite-differences
    %
    % Inputs:
    %                  xRBF       -> Inital x-coord.     of RBFs
    %                  yRBF       -> Inital y-coord.     of RBFs
    %                  aRBF       -> Inital coefficient  of RBFs
    %                  SR         -> Support radius
    %                  nbdy       -> Total number of bodies
    %                  poly_fit   -> Fit type
    %                  xySpln     -> Initial spline xy-coordinates
    %                  xyCurv     -> Initial curvature data
    %                  xyTpar     -> Initial parametric coordinate data
    %                  xrays_des  -> Desired x-rays for objective calcs
    %                  nrays      -> Number of rays used for objective
    %                                calculations
    %                  NRML       -> Whether the objective function
    %                                should be normalized or not
    %                  hmin       -> Minimum y-coordinate of the
    %                                horizontal x-rays
    %                  hmax       -> Maximum y-coordinate of the
    %                                horizontal x-rays
    %                  vmin       -> Minimum x-coordinate of the
    %                                vertical x-rays
    %                  vmax       -> Maximum x-coordinate of the
    %                                vertical x-rays
    %
    % Outputs:
    %                  dOdp       -> Gradient array of the design
    %                                parameters
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti________________________________02/24/20

    ObjFunc  = EvaluateObjective(  xySpln        , ...
                                   xyCurv        , ...
                                   xyTpar        , ...
                                   poly_fit      , ...
                                   xrays_des     , ...
                                   NRML          , ...
                                   nrays         , ...
                                   hmin          , ...
                                   hmax          , ...
                                   vmin          , ...
                                   vmax          , ...
                                   0              );

    FDstep  = 5e-7;

    mRBF     = length(xRBF);
    dOdp     = zeros(3*mRBF,1);
```

```
dp       = zeros(3*mRBF,1);
dxySpln = zeros(size(xySpln));

for ipar = 1 : 3*mRBF
    dp(ipar) = FDstep;
    dxRBF = dp(        1:  mRBF);
    dyRBF = dp(  mRBF+1:2*mRBF);
    dalfa = dp(2*mRBF+1:3*mRBF);

    for j = 1 : length(xySpln)/2
        if(isnan(xySpln(2*j-1)) && isnan(xySpln(2*j  )))
            dxySpln(2*j-1) = NaN;
            dxySpln(2*j  ) = NaN;
            continue;
        end %if
        xyk  = [xySpln(2*j-1);...
                xySpln(2*j  )];
        dphidx  = 0;
        dphidy  = 0;
        RHS     = 0;
        for i = 1 : mRBF
%
% Calculate the radius from the RBF location to the
% intersection point:
%
            r =((xyk(1) -xRBF(i))^2 +...
                (xyk(2) -yRBF(i))^2)^(1/2);
            zeta = r/SR;
%
% Check if intersection point is outside of the RBFs support
% radius:
%
            if (zeta > 1)
                psi     = 0.0;
                dphidx_i= 0.0;
                dphidy_i= 0.0;
                continue
            elseif( zeta == 0 )
                psi     = 1.0;
                dphidx_i= 0.0;
                dphidy_i= 0.0;
                continue
            elseif( zeta < 1 )
                psi     = (1 - zeta)^4 * (4*zeta + 1);
                psi_dot =-(20/SR) * zeta * (1 - zeta)^3;
                drdx    = (xyk(1) -xRBF(i)) / r;
                drdy    = (xyk(2) -yRBF(i)) / r;

                dphidx_i= aRBF(i) * psi_dot * drdx;
                dphidy_i= aRBF(i) * psi_dot * drdy;
            end %if

            dphidx  = dphidx + dphidx_i;
            dphidy  = dphidy + dphidy_i;
            RHS     = RHS    - dalfa(i)*psi + (dphidx_i*dxRBF(i) ...
                                           +  dphidy_i*dyRBF(i));
        end %for i
%
% Assuming x = const * dphidy and y = const * dphidx
%
        den            =(dphidx^2 + dphidy^2);
        const          = RHS/den;
        dxySpln(2*j-1) = const * dphidx;
        dxySpln(2*j  ) = const * dphidy;
    end %for j

    xySpln_FD  = xySpln + dxySpln;
```

```
        if(nbdy > 1)
            xy_par = [];
            tpar_FD= [];
            d2xy_FD= [];
            ibeg   = ones(nbdy,1);
            ifin   = zeros(nbdy,1);
            k      = 0;
            for ip = 1 : length(xySpln_FD)/2
                if(isnan(xySpln_FD(2*ip-1)))
                    k = k + 1;
                    if(k < nbdy)
                        ibeg(k+1) = ip+1;
                        ifin(k  ) = ip-1;
                    elseif(k == nbdy)
                        ifin(k  ) = ip-1;
                    end %if
                end %if
            end %for i
            for ibdy = 1 : nbdy
                i1   = 2*ibeg(ibdy)-1;
                i2   = 2*ifin(ibdy)  ;
                [ xy_ibdy  , ...
                  knot_ibdy, ...
                  d2xy_ibdy ] = spline_fit( poly_fit           , ...
                                            xySpln_FD( i1 : i2 ), ...
                                            3                    );
                xy_par = [ xy_par ; xy_ibdy  ; NaN; NaN];
                tpar_FD= [ tpar_FD; knot_ibdy; NaN     ];
                d2xy_FD= [ d2xy_FD; d2xy_ibdy; NaN; NaN];
            end %for ibdy
        elseif(nbdy == 1)
            [ ~      , ...
              tpar_FD, ...
              d2xy_FD ] = spline_fit( poly_fit , ...
                                      xySpln_FD, ...
                                      3         );
        end %if
        xyCurv_FD = d2xy_FD;
        xyTpar_FD = tpar_FD;

        ObjFunc_FD = EvaluateObjective( xySpln_FD    , ...
                                        xyCurv_FD    , ...
                                        xyTpar_FD    , ...
                                        poly_fit     , ...
                                        xrays_des, ...
                                        NRML         , ...
                                        nrays        , ...
                                        hmin         , ...
                                        hmax         , ...
                                        vmin         , ...
                                        vmax         , ...
                                        0             );

        dOdp(ipar) = (ObjFunc_FD - ObjFunc) / FDstep;
        dp(ipar)   = 0.0;

    end %for ipar

end
%function finite_diff_sensitivities( )

%------------------------------------------------------------------------------
```

```
function dOdp = complexstep_sensitivities(  xRBF          , ...
                                            yRBF          , ...
                                            aRBF          , ...
                                            SR            , ...
                                            nbdy          , ...
                                            poly_fit      , ...
                                            mpts          , ...
                                            xySpln        , ...
                                            xrays_desired , ...
                                            nrays         , ...
                                            NRML          , ...
                                            hmin          , ...
                                            hmax          , ...
                                            vmin          , ...
                                            vmax          , ...
                                            comp_calc      )
    % function complexstep_sensitivities( ) calculates the design
    % sensitivities of the RBF locations and coefficients w.r.t. a given
    % objective function using complex step algorithmic differentiation
    %
    % Inputs:
    %                   xRBF        -> Inital x-coord.    of RBFs
    %                   yRBF        -> Inital y-coord.    of RBFs
    %                   aRBF        -> Inital coefficient of RBFs
    %                   SR          -> Support radius
    %                   nbdy        -> Total number of bodies
    %                   poly_fit    -> Fit type
    %                   mpts        -> number of points along each spline
    %                                  segment
    %                   xySpln      -> Initial spline xy-coordinates
    %                   xrays_des   -> Desired x-rays for objective calcs
    %                   nrays       -> Number of rays used for objective
    %                                  calculations
    %                   NRML        -> Whether the objective function
    %                                  should be normalized or not
    %                   hmin        -> Minimum y-coordinate of the
    %                                  horizontal x-rays
    %                   hmax        -> Maximum y-coordinate of the
    %                                  horizontal x-rays
    %                   vmin        -> Minimum x-coordinate of the
    %                                  vertical x-rays
    %                   vmax        -> Maximum x-coordinate of the
    %                                  vertical x-rays
    %                   comp_calc   -> Indicator for number of design
    %                                  variables to use for optimization:
    %                                  1 - RBF locations and coefficients;
    %                                  2 - RBF locations;
    %                                  3 - RBF coefficients
    %
    % Outputs:
    %                   dOdp        -> Gradient array of the design
    %                                  parameters
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    CSstep  = 1e-8;
    mRBF    = length(xRBF);

    if(comp_calc == 1)
        npar = 3;
    elseif(comp_calc == 2)
        npar = 2;
    elseif(comp_calc == 3)
        npar = 1;
    elseif(comp_calc == 4)
        npar = 1;
    end %if
```

```
dOdp     = zeros(npar*mRBF,1);
dp       = zeros(npar*mRBF,1);
dxySpln = zeros(size(xySpln));

for ipar = 1 : npar*mRBF
    dp(ipar) = CSstep * sqrt(-1);
    if(comp_calc == 1)
        dxRBF = dp(       1 : 1*mRBF);
        dyRBF = dp(1*mRBF+1: 2*mRBF);
        daRBF = dp(2*mRBF+1: 3*mRBF);
    elseif(comp_calc == 2)
        dxRBF = dp(       1 : 1*mRBF);
        dyRBF = dp(1*mRBF+1: 2*mRBF);
        daRBF = zeros(size(dxRBF));
    elseif(comp_calc == 3)
        daRBF = dp;
        dxRBF = zeros(size(daRBF));
        dyRBF = zeros(size(daRBF));
    elseif(comp_calc == 4)
        dxRBF = dp;
        dyRBF = zeros(size(dxRBF));
        daRBF = zeros(size(dxRBF));
    end %if

    for j = 1 : length(xySpln)/2
        if(isnan(xySpln(2*j-1)) && isnan(xySpln(2*j   )))
            dxySpln(2*j-1) = NaN;
            dxySpln(2*j   ) = NaN;
            continue;
        end %if
        xpnt    = xySpln(2*j-1);
        ypnt    = xySpln(2*j   );
        dphidx  = 0;
        dphidy  = 0;
        RHS     = 0;
        for i = 1 : mRBF
%
% Calculate the radius from the RBF location to the
% intersection point:
%
            r =((xpnt -xRBF(i))^2 +...
                (ypnt -yRBF(i))^2)^(1/2);
            zeta     = r/SR;
            zeta_dot = 1/SR;
%
% Check if intersection point is outside of the RBFs support
% radius:
%
            if (zeta > 1)
                psi     = 0.0;
                dphidx_i= 0.0;
                dphidy_i= 0.0;
                continue
            elseif( zeta == 0 )
                psi     = 1.0;
                dphidx_i= 0.0;
                dphidy_i= 0.0;
                continue
            elseif( zeta < 1 )
                psi     = (1 - zeta)^4 * (4*zeta + 1);
                psi_dot =-(20) * zeta * (1 - zeta)^3;
                rx_dot  = (xpnt -xRBF(i)) / r;
                ry_dot  = (ypnt -yRBF(i)) / r;

                dphidx_i= aRBF(i) * psi_dot * zeta_dot * rx_dot;
                dphidy_i= aRBF(i) * psi_dot * zeta_dot * ry_dot;
            end %if
            dphidx  = dphidx + dphidx_i;
            dphidy  = dphidy + dphidy_i;
            RHS     = RHS    - daRBF(i)*psi + (dphidx_i*dxRBF(i) ...
                                            +  dphidy_i*dyRBF(i));
        end %for i
%
% Assuming x = const * dphidy and y = const * dphidx
%
        den             =(dphidx^2 + dphidy^2);
        const           = RHS/den;
        dxySpln(2*j-1) = const * dphidx;
        dxySpln(2*j  ) = const * dphidy;
    end %for j

    xySpln_CS  = xySpln + dxySpln;

    xy_par = [];
    tpar_CS= [];
    d2xy_CS= [];
    ibeg   = ones(nbdy,1);
    ifin   = zeros(nbdy,1);
    k      = 0;
    for ip = 1 : length(xySpln_CS)/2
        if(isnan(xySpln_CS(2*ip-1)))
            k = k + 1;
            if(k < nbdy)
                ibeg(k+1) = ip+1;
                ifin(k  ) = ip-1;
            elseif(k == nbdy)
                ifin(k  ) = ip-1;
            end %if
        end %if
    end %for i
    for ibdy = 1 : nbdy
        i1   = 2*ibeg(ibdy)-1;
        i2   = 2*ifin(ibdy)  ;
        [ xy_ibdy  , ...
          knot_ibdy , ...
          d2xy_ibdy ] = spline_fit( poly_fit            , ...
                                    xySpln_CS( i1 : i2 ), ...
                                    mpts                 );
        xy_par = [ xy_par ; xy_ibdy  ; NaN; NaN];
        tpar_CS= [ tpar_CS; knot_ibdy; NaN     ];
        d2xy_CS= [ d2xy_CS; d2xy_ibdy; NaN; NaN];
    end %for ibdy
    if(poly_fit == 1 || poly_fit == 3)
        ObjFunc = EvaluateObjective(   xySpln_CS     , ...
                                       d2xy_CS       , ...
                                       tpar_CS       , ...
                                       poly_fit      , ...
                                       xrays_desired , ...
                                       NRML          , ...
                                       nrays         , ...
                                       hmin          , ...
                                       hmax          , ...
                                       vmin          , ...
                                       vmax          , ...
                                       0              );
    elseif(poly_fit == 2)
        ObjFunc = EvaluateObjective(   xy_par        , ...
                                       d2xy_CS       , ...
                                       tpar_CS       , ...
                                       poly_fit      , ...
                                       xrays_desired , ...
                                       NRML          , ...
                                       nrays         , ...
                                       hmin          , ...
```

```
                                              hmax           , ...
                                              vmin           , ...
                                              vmax           , ...
                                              0               );
        end %if

        dOdp(ipar) = imag(ObjFunc) / CSstep;
        dp(ipar)   = 0.0;
    end %for ipar

end
%function complexstep_sensitivities( )

%---------------------------------------------------------------------
```

```
                                                                    yRBF        , ...
                                                                    aRBF        , ...
                                                                    SR          , ...
                                                                    nbdy        , ...
                                                                    poly_fit    , ...
                                                                    mpts        , ...
                                                                    xySpln      , ...
                                                                    xyCurv      , ...
                                                                    xyTpar      , ...
                                                                    xrays_des   , ...
                                                                    nrays       , ...
                                                                    NRML        , ...
                                                                    hmin        , ...
                                                                    hmax        , ...
                                                                    vmin        , ...
                                                                    vmax          )
% function tangent_sensitivities( ) calculates the design sensitivities
% of the RBF locations and coefficients w.r.t. a given objective
% function using tangent linear approximation and algorithmic
% differentiation
%
% Inputs:
%               xRBF        -> Inital x-coord.    of RBFs
%               yRBF        -> Inital y-coord.    of RBFs
%               aRBF        -> Inital coefficient of RBFs
%               SR          -> Support radius
%               nbdy        -> Total number of bodies
%               poly_fit    -> Fit type
%               mpts        -> number of points along each spline
%                              segment
%               xySpln      -> Initial spline xy-coordinates
%               xyCurv      -> Initial curvature data
%               xyTpar      -> Initial parametric coordinate data
%               xrays_des   -> Desired x-rays for objective calcs
%               nrays       -> Number of rays used for objective
%                              calculations
%               NRML        -> Whether the objective function
%                              should be normalized or not
%               hmin        -> Minimum y-coordinate of the
%                              horizontal x-rays
%               hmax        -> Maximum y-coordinate of the
%                              horizontal x-rays
%               vmin        -> Minimum x-coordinate of the
%                              vertical x-rays
%               vmax        -> Maximum x-coordinate of the
%                              vertical x-rays
%
% Outputs:
%               dOdp        -> Gradient array of the design
%                              parameters
%
% Written by Jack Rossetti
% Annontated by Jack Rossetti_________________________________02/24/20

mRBF        = length(xRBF);
dOdp        = zeros(3*mRBF,1);
dp          = zeros(3*mRBF,1);
xySpln_dot = zeros(size(xySpln));

for ipar = 1 : 3*mRBF
    dp(ipar) = 1.0;
    xRBF_dot = dp(        1:  mRBF);
    yRBF_dot = dp(  mRBF+1:2*mRBF);
    aRBF_dot = dp(2*mRBF+1:3*mRBF);

    for j = 1 : length(xySpln)/2
        if(isnan(xySpln(2*j-1)) && isnan(xySpln(2*j  )))
            xySpln_dot(2*j-1) = NaN;
            xySpln_dot(2*j   ) = NaN;
            continue;
        end %if
        xyk  = [xySpln(2*j-1);...
                xySpln(2*j   )];
        dphidx  = 0;
        dphidy  = 0;
        RHS     = 0;
        for i = 1 : mRBF
%
% Calculate the radius from the RBF location to the
% intersection point:
%
            r =((xyk(1) -xRBF(i))^2 +...
                (xyk(2) -yRBF(i))^2)^(1/2);
            zeta = r/SR;
%
% Check if intersection point is outside of the RBFs support
% radius:
%
            if (zeta > 1)
                psi     = 0.0;
                dphidx_i= 0.0;
                dphidy_i= 0.0;
                continue
            elseif( zeta == 0 )
                psi     = 1.0;
                dphidx_i= 0.0;
                dphidy_i= 0.0;
                continue
            elseif( zeta < 1 )
                psi     = (1 - zeta)^4 * (4*zeta + 1);
                psi_dot =-(20/SR) * zeta * (1 - zeta)^3;
                drdx    = (xyk(1) -xRBF(i)) / r;
                drdy    = (xyk(2) -yRBF(i)) / r;

                dphidx_i= aRBF(i) * psi_dot * drdx;
                dphidy_i= aRBF(i) * psi_dot * drdy;
            end %if

            dphidx  = dphidx + dphidx_i;
            dphidy  = dphidy + dphidy_i;
            RHS     = RHS    - aRBF_dot(i)*psi + (dphidx_i*xRBF_dot(i) + ...
                                                  dphidy_i*yRBF_dot(i));
        end %for i
%
% Assuming x = const * dphidy and y = const * dphidx
%
        den             =(dphidx^2 + dphidy^2);
        const           = RHS/den;
        xySpln_dot(2*j-1) = const * dphidx;
        xySpln_dot(2*j   ) = const * dphidy;
    end %for j

    xy_par = [];
    xy_dot = [];
    xyCurv = [];
    d2_dot = [];
    xyTpar = [];
    t_dot  = [];
    ibeg   = ones(nbdy,1);
    ifin   = zeros(nbdy,1);
    k      = 0;
    for ip = 1 : length(xySpln_dot)/2
        if(isnan(xySpln_dot(2*ip-1)))
            k = k + 1;
            if(k < nbdy)
                ibeg(k+1) = ip+1;
                ifin(k  ) = ip-1;
            elseif(k == nbdy)
                ifin(k  ) = ip-1;
            end %if
        end %if
    end %for i
    for ibdy = 1 : nbdy
        i1   = 2*ibeg(ibdy)-1;
        i2   = 2*ifin(ibdy)  ;
        [ xy_pts    , ...
          xy_tng    , ....
          knot_ibdy , ...
          knot_dot  , ...
          d2xy_ibdy , ...
          d2xy_dot  ] = spline_fit_Dot( poly_fit              , ...
                                        xySpln( i1 : i2 )     , ...
                                        xySpln_dot( i1 : i2 ), ...
                                        mpts                  );
        xy_par   = [ xy_par; xy_pts    ; NaN; NaN  ];
        xy_dot   = [ xy_dot; xy_tng    ; NaN; NaN  ];
        xyCurv   = [ xyCurv; d2xy_ibdy; NaN; NaN  ];
        d2_dot   = [ d2_dot; d2xy_dot ; NaN; NaN  ];
        xyTpar   = [ xyTpar; knot_ibdy; NaN       ];
        t_dot    = [ t_dot ; knot_dot ; NaN       ];
    end %for ibdy
%     xy_par
%     pause
%     xy_dot
%     pause
    if(poly_fit == 1 || poly_fit == 3)
        [ ~     , ...
          O_dot ] = EvaluateObjective_Dot( xySpln        , ...
                                           xySpln_dot    , ...
                                           xyCurv        , ...
                                           d2_dot        , ...
                                           xyTpar        , ...
                                           t_dot         , ...
                                           poly_fit      , ...
                                           xrays_des     , ...
                                           NRML          , ...
                                           nrays         , ...
                                           hmin          , ...
                                           hmax          , ...
                                           vmin          , ...
                                           vmax          , ...
                                           0             );
    elseif(poly_fit == 2)
        [ ~     , ...
          O_dot ] = EvaluateObjective_Dot( xy_par        , ...
                                           xy_dot        , ...
                                           xyCurv        , ...
                                           d2_dot        , ...
                                           xyTpar        , ...
                                           t_dot         , ...
                                           poly_fit      , ...
                                           xrays_des     , ...
                                           NRML          , ...
                                           nrays         , ...
                                           hmin          , ...
                                           hmax          , ...
                                           vmin          , ...
                                           vmax          , ...
                                           0             );
    end %if

    dOdp(ipar) = O_dot;
    dp(ipar)   = 0.0;
    end %for ipar

end
%function tangent_sensitivities( )

%------------------------------------------------------------------------
```

```
function X = bspline_func_jsr( curv, ...
                              npts )
    % function bspline_func_jsr( ) generates a b-spline curve based on
    % inputs
    %
    % Inputs:
    %                   curv        -> curve to be produced
    %                   npts        -> number of points along the curve
    %
    % Outputs:
    %                   X           -> xy-coordinates of the b-spline curve
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti________________________________02/24/20

    n = 3;
    if(curv == 1)
        t = [1 3 4 5 7 8 10 11 12 13 14];
        P = [ 0.3993 0.4965 0.6671 0.7085 0.5000 0.4500 0.3993 0.4965; ...
              0.8377 0.8436 0.7617 0.5126 0.2120 0.6500 0.8377 0.8436 ];  % 7 points, 2
                  overlap
    elseif(curv == 2)
        t = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15];
        P = [ 1.0   0.4   0.50   0.00 -0.50 -0.60 -1.00 -0.60 -0.50   0.40   1.00   0.40; ...
              0.0   0.2   0.87   0.40   0.87   0.20   0.00 -0.40 -0.87 -0.67 -0.00   0.20 ];
    elseif(curv == 3)
        t = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15];
        P = [ 1.0   0.0   0.50   0.00 -0.70   0.00 -1.00 -0.60 -0.50   0.40   1.00   0.00; ...
              0.0   0.2   0.87   0.70   0.87   0.20   0.00 -0.40 -0.87 -0.67 -0.00   0.20 ];
    end %if

    X = bspline_deboor(n,t,P,npts);

end
%function bspline_func_jsr

%------------------------------------------------------------------------
```

```
function [ C, ...
           U ] = bspline_deboor( n, ...
                                 t, ...
                                 P, ...
                                 U )
% Evaluate explicit B-spline at specified locations.
%
% Input arguments:
% n:
%    B-spline order (2 for linear, 3 for quadratic, etc.)
% t:
%    knot vector
% P:
%    control points, typically 2-by-m, 3-by-m or 4-by-m (for weights)
% u (optional):
%    values where the B-spline is to be evaluated, or a positive
%    integer to set the number of points to automatically allocate
%
% Output arguments:
% C:
%    points of the B-spline curve
% Copyright 2010 Levente Hunyadi

    validateattributes(n, {'numeric'}, {'positive','integer','scalar'});
    d = n-1;  % B-spline polynomial degree (1 for linear, 2 for quadratic, etc.)
    validateattributes(t, {'numeric'}, {'real','vector'});
    assert(all( t(2:end)-t(1:end-1) >= 0 ), 'bspline:deboor:InvalidArgumentValue', ...
        'Knot vector values should be nondecreasing.');
    validateattributes(P, {'numeric'}, {'real','2d'});
    nctrl = numel(t)-(d+1);
    assert(size(P,2) == nctrl, 'bspline:deboor:DimensionMismatch', ...
        'Invalid number of control points, %d given, %d required.', size(P,2), nctrl);
    if nargin < 4
        U = linspace(t(d+1), t(end-d), 10*size(P,2));  % allocate points uniformly
    elseif isscalar(U) && U > 1
        validateattributes(U, {'numeric'}, {'positive','integer','scalar'});
        U = linspace(t(d+1), t(end-d), U);  % allocate points uniformly
    else
        validateattributes(U, {'numeric'}, {'real','vector'});
        assert(all( U >= t(d+1) & U <= t(end-d) ), 'bspline:deboor:InvalidArgumentValue',
            ...
            'Value outside permitted knot vector value range.');
    end

    m = size(P,1);  % dimension of control points
    t = t(:).';     % knot sequence
    U = U(:);
    S = sum(bsxfun(@eq, U, t), 2);  % multiplicity of u in t (0 <= s <= d+1)
    I = bspline_deboor_interval(U,t);

    Pk = zeros(m,d+1,d+1);
    a = zeros(d+1,d+1);

    C = zeros(size(P,1), numel(U));
    for j = 1 : numel(U)
        u = U(j);
        s = S(j);
        ix = I(j);
        Pk(:) = 0;
        a(:) = 0;

        % identify d+1 relevant control points
        Pk(:, (ix-d):(ix-s), 1) = P(:, (ix-d):(ix-s));
        h = d - s;

        if h > 0
            % de Boor recursion formula
            for r = 1 : h
```

```
                q = ix-1;
                for i = (q-d+r) : (q-s)
                    a(i+1,r+1) = (u-t(i+1)) / (t(i+d-r+1+1)-t(i+1));
                    Pk(:,i+1,r+1) = (1-a(i+1,r+1)) * Pk(:,i,r) + a(i+1,r+1) * Pk(:,i+1,r);
                end
            end
            C(:,j) = Pk(:,ix-s,d-s+1);  % extract value from triangular computation scheme
        elseif ix == numel(t)  % last control point is a special case
            C(:,j) = P(:,end);
        else
            C(:,j) = P(:,ix-d);
        end
    end

end
%function bspline_deboor

%------------------------------------------------------------------------
```

```
function ix = bspline_deboor_interval( u, ...
                                       t )
% Index of knot in knot sequence not less than the value of u.
% If knot has multiplicity greater than 1, the highest index is returned.

    i = bsxfun(@ge, u, t) & bsxfun(@lt, u, [t(2:end) 2*t(end)]);  % indicator of knot
        interval in which u is
    [row,col] = find(i);
    [row,ind] = sort(row);  %#ok<ASGLU> % restore original order of data points
    ix = col(ind);

end
%function bspline_deboor_interval

%----------------------------------------------------------------------
```

```
function [ in ] = inpolygon_JSR( xpoly, ...
                                 ypoly, ...
                                 xp   , ...
                                 yp    )
   % function inpolygon_JSR( ) checks to see if a given point (xp, yp) is
   % within a polygon described by xpoly, ypoly
   %
   % Inputs:
   %                  xpoly       -> x-coordinates of the polygon
   %                  ypoly       -> y-coordinates of the polygon
   %                  xp          -> x-coordinate of the point in
   %                                 questions
   %                  yp          -> y-coordinate of the point in
   %                                 questions
   %
   % Outputs:
   %                  in          -> parameter that indicates whether the
   %                                 point is in or out of the body
   %
   % Written by Jack Rossetti
   % Annontated by Jack Rossetti_________________________________02/24/20

   %--- Loop through the segments in the polygon ---%
   %       == Assumes polygon is ordered ==         %
   cross = 0;
   for i = 1 : length(xpoly)
       if(i < length(xpoly))
           ip1 = i+1;
       elseif(i == length(xpoly))
           ip1 = 1;
       end %if
       if(isnan(xpoly(i)) || isnan(xpoly(ip1)))
           continue;
       end %if
       xs = [xpoly(i), xpoly(ip1)];
   %
   % Find the max and min x-value on the segment
   %
       xmax = -100000000;
       for j = 1 : 2
           if(xs(j) > xmax)
               xmax = xs(j);
           end %if
       end %for j
   %
   % Check if the segment is to the right of the point
   %
       if(xp > xmax) % segment is to the left of the point
           continue;
       end %if
   %
   % Check if ray crosses segment
   %
       ys = [ypoly(i), ypoly(ip1)];
   %
   % Find the max and min y-value on the segment
   %
       ymax = -100000000;
       ymin = +100000000;
       for j = 1 : 2
           if(ys(j) > ymax)
               ymax = ys(j);
           end %if
           if(ys(j) < ymin)
               ymin = ys(j);
           end %if
       end %for j
   %
```

```
    % Handle horizontal segments
    %
        if(ymax == ymin && yp == ymax)
            fprintf(1, 'Segment is horizontal and point is either on segment or left of
                segment\n');
        end %if
    %
    % Check if point aligns with a vertex
    %
        if(yp == ymax)
            yp = yp - eps;
        elseif(yp == ymin)
            yp = yp + eps;
        end %if
    %
    % Check if yp is within ymax and ymin
    %
        if(ymin < yp && yp < ymax)
    %
    % Check where the ray intersects the segment
    %
            m      =(ys(2) - ys(1))/(xs(2) - xs(1));
            if(isnan(m)) % m is zero (vertical line)
                cross = cross + 1;
                continue;
            end %if
            yint  = yp;
            xint  = xs(2) - (1/m)*(ys(2) - yint);
            if(xint > xp) % Ray intersects segment to the right of point
                cross = cross + 1;
            end %if
        end %if
    end %for i

    if(mod(cross,2) == 0)
        in = 0;
    elseif(mod(cross,2) == 1)
        in = 1;
    end %if

end

%function inpolygon_JSR

%-----------------------------------------------------------------------------
```

```
function xyn = RK4_Step(  xy   , ...
                          xRBF, ...
                          yRBF, ...
                          SR   , ...
                          aRBF, ...
                          h    , ...
                          dir   )
    % function RK4_Step( ) solves a system of ODEs using a 4th-order
    % Runge-Kutta algorithm
    %
    % Inputs:
    %                   xy          -> xy-coordinate of the initial point
    %                   xRBF        -> x-coordinates of the RBFs
    %                   yRBF        -> y-coordinates of the RBFs
    %                   SR          -> support radius of the RBFs
    %                   aRBF        -> coefficients of the RBFs
    %                   h           -> step size
    %                   dir         -> direction of the march, +1 or -1
    %
    % Outputs:
    %                   xyn         -> xy-coordinate of the new point
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti___________________________________02/24/20

    xyn= zeros(1,2);
    xy1= xy;
    uv1= dir*dLSF_tot(  xy1 , ...
                        xRBF, ...
                        yRBF, ...
                        SR   , ...
                        aRBF     );
    k1 = h*uv1(1);
    m1 = h*uv1(2);

    xy2= [xy1(1)+0.5*k1, xy1(2)+0.5*m1];
    uv2= dir*dLSF_tot(  xy2 , ...
                        xRBF, ...
                        yRBF, ...
                        SR   , ...
                        aRBF     );
    k2 = h*uv2(1);
    m2 = h*uv2(2);

    xy3= [xy1(1)+0.5*k2, xy1(2)+0.5*m2];
    uv3= dir*dLSF_tot(  xy3 , ...
                        xRBF, ...
                        yRBF, ...
                        SR   , ...
                        aRBF     );
    k3 = h*uv3(1);
    m3 = h*uv3(2);

    xy4= [xy1(1)+1.0*k3, xy1(2)+1.0*m3];
    uv4= dir*dLSF_tot(  xy4 , ...
                        xRBF, ...
                        yRBF, ...
                        SR   , ...
                        aRBF     );
    k4 = h*uv4(1);
    m4 = h*uv4(2);

    xyn(1) = xy1(1) + (1/6) * (k1 + 2*k2 + 2*k3 + k4);
    xyn(2) = xy1(2) + (1/6) * (m1 + 2*m2 + 2*m3 + m4);

end
%function RK4_Step
%------------------------------------------------------------------------
```

```
function [ tn , ...
           xyn ] = RK4_LSF( t       , ...
                            xy      , ...
                            xRBF    , ...
                            yRBF    , ...
                            SR      , ...
                            aRBF    , ...
                            OFFSET  , ...
                            h_guess , ...
                            eta_tol , ...
                            dir      )
    % function RK4_LSF( ) iteratively generates points along the zero
    % level-set curve using cubic spline fitting methods and an adaptive
    % 4th-order Runge-Kutta-like technique.
    %
    % Inputs:
    %                   t           -> range of t, parameter along the
    %                                  boundary
    %                   xy          -> xy-coordinate of the initial point
    %                   xRBF        -> x-coordinates of the RBFs
    %                   yRBF        -> y-coordinates of the RBFs
    %                   SR          -> support radius of the RBFs
    %                   aRBF        -> coefficients of the RBFs
    %                   OFFSET      -> offset value for the level-set func
    %                   h_guess     -> initial step size
    %                   eta_tol     -> tolerance for distance from the zero
    %                                  level-set curve
    %                   dir         -> direction of the march, +1 or -1
    %
    % Outputs:
    %                   xyn         -> xy-coordinate of the new point
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti__________________________________02/24/20

    xyn      = ones(99999,2)*NaN;
    xyn(1,:) = [xy(1) xy(2)];
    tn       = ones(99999,1)*NaN;
    tn(1)    = t(1);
    i        = 1;
    h_temp   = h_guess;
    h        = h_temp;
    nstep    = 1;
    for iter = 1 : 1000000
    %
    % Take a nsteps at h
    %
        xyh = zeros(nstep, 2);
        xyt  = xyn(i,:);
        for istep = 1 : nstep
            xyh(istep,:) = RK4_Step( xyt , ...
                                     xRBF, ...
                                     yRBF, ...
                                     SR  , ...
                                     aRBF, ...
                                     h   , ...
                                     dir  );

            xyt          = xyh(istep,:);
        end %for istep


        phi = EvaluateLSF( xyh(nstep,1), ...
                           xyh(nstep,2), ...
                           xRBF        , ...
                           yRBF        , ...
                           aRBF        , ...
                           SR          , ...
```

```
                        OFFSET        );

     dphi= grad_phi( [xyh(nstep,1), ...
                      xyh(nstep,2) ], ...
                      xRBF          , ...
                      yRBF          , ...
                      SR            , ...
                      aRBF          );
     deta= -phi /(dphi(1)^2 + dphi(2)^2);

     if( abs(deta) >= eta_tol)
         phi = EvaluateLSF( xyn(i,1), ...
                            xyn(i,2), ...
                            xRBF    , ...
                            yRBF    , ...
                            aRBF    , ...
                            SR      , ...
                            OFFSET  );

         dphi= grad_phi( [xyn(i,1), ...
                          xyn(i,2) ], ...
                          xRBF      , ...
                          yRBF      , ...
                          SR        , ...
                          aRBF      );
         deta= -phi /(dphi(1)^2 + dphi(2)^2);
         xyn(i,:) = xyn(i,:) + dphi' * deta;
         h     = 0.50*h;
         nstep = 2*nstep;
         if(nstep > 16)
             i = 1;
             h_temp  = 0.9 * h_temp;
             h       = h_temp;
             nstep   = 1;
             xyn     = ones(99999,2)*NaN;
             xyn(1,:)= [xy(1) xy(2)];
         end %if
     elseif( abs(deta) < eta_tol)
%
% Increment i to advance to next point
%
         if(i == 1)
             step1 = nstep;
         end %if
         for istep = 1 : nstep
             xyn(i+1,:) = xyh(istep, :);
             tn(i+1)    = tn(i) + h;
             i          = i + 1;
         end %for istep
         h     =  h_temp;
         nstep = 1;
     end %if

     if(i > 10)
%
% Calculate the distance from the first point of the first segment to
% the first point of the second segment (d212), the distance from the
% first point to the ith point (di12), and the distance of the last
% segment (di1m) and compare.
%
         di12 = (xyn(i,1) - xyn(      1,1))^2 + (xyn(i,2) - xyn(      1,2))^2;
         di1m = (xyn(i,1) - xyn(i-nstep,1))^2 + (xyn(i,2) - xyn(i-nstep,2))^2;
         d212 = (xyn(1,1) - xyn(1+step1,1))^2 + (xyn(1,2) - xyn(1+step1,2))^2;
         if(2.0*d212 > di12 || ...
                di1m > di12   ) % This criteria is not perfect and can
                                % be removed.
%
% Assume curve is closed
```

```
    %
                break;
            end %if
        end %if

        if(i > 2)
    %
    % Check if new line segment has intersected any other previous segments
    %
            % First line segment
            x11 = xyn(1,1);
            y11 = xyn(1,2);
            x12 = xyn(2,1);
            y12 = xyn(2,2);
            % Current line segment
            x21 = xyn(i-1,1);
            y21 = xyn(i-1,2);
            x22 = xyn(i   ,1);
            y22 = xyn(i   ,2);
    %
    % Solve for t and s parameter along each line. If both are less than
    % one and greater than zero, the lines have intersected.
    %
    % From the solution of a system of two equations and two unknowns t and
    % s can be directly solved for as,
    %
            tt=((x12 - x11) * (y11 - y21) - (x11 - x21) * (y12 - y11))/ ...
               ((x12 - x11) * (y22 - y21) - (x22 - x21) * (y12 - y11));
            ss=((x22 - x21) * (y11 - y21) - (x11 - x21) * (y22 - y21))/ ...
               ((x12 - x11) * (y22 - y21) - (x22 - x21) * (y12 - y11));
%              fprintf(1, 'tt = %+f, ss = %+f\n', tt, ss);
%              figure(1026)
%              hold on
%              w = plot(xyn(:,1), xyn(:,2), 'mo-');
%              hold off
%              pause(0.001)
%              delete(w)
            if( (tt > 0 && tt < 1) && ...
                (ss > 0 && ss < 1))
%                  fprintf(1, 'INTERSECTION FOUND!\n');
%                  figure(1026)
%                  hold on
%                  w = plot(xyn(:,1), xyn(:,2), 'mo-');
%                  hold off
%                  pause(0.1)
%                  delete(w)
                temp = xyn;
                clear xyn;
                xyn          = ones(i,2) .* NaN;
                xyn(1:i-1,:) = temp(1:i-1,:);
                clear temp
    %
    % Assume curve is closed
    %
                break;
            end %if
        end %if
        if(tn(i) > t(2))
            break;
        end %if
    end %for iter

end
%function RK4_LSF

%-------------------------------------------------------------------
```

```
function dydx = dLSF_tot( xy   , ...
                          xRBF, ...
                          yRBF, ...
                          SR   , ...
                          aRBF )
    % function dLSF_tot( ) calculates the tangent vector at a particular
    % point in the level-set field
    %
    % Inputs:
    %                   xy          -> xy-coordinate of the initial point
    %                   xRBF        -> x-coordinates of the RBFs
    %                   yRBF        -> y-coordinates of the RBFs
    %                   SR          -> support radius of the RBFs
    %                   aRBF        -> coefficients of the RBFs
    %
    % Outputs:
    %                   dydx        -> tangent vector
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20
    x = xy(1);
    y = xy(2);
    %
    % Calculate the tangent and normal vectors at the x,y location:
    %
    dLSFx = 0;
    dLSFy = 0;
    k = 1;
    for j = 1 : length(xRBF)
        if(isnan(xRBF(j)))
            continue
        end %if
    %
    % Calculate the radius from the RBF location to the
    % intersection point:
    %
        r2=(x -xRBF(j))^2 + (y -yRBF(j))^2;
        r = sqrt(r2);
    %
    % Check if intersection point is outside of the RBFs support
    % radius:
    %
        dRBFx = 0;
        dRBFy = 0;
        if( r <= SR ) && ( r ~= 0.0 )
            term1 = (1 - (r/SR))^4;
            term2 = (4*(r/SR)+1)*(1 - (r/SR))^3;
            dRBFx = (4/SR)*(term1 - term2)*(r2^(-1/2)*(x - xRBF(j)));
            dRBFy = (4/SR)*(term1 - term2)*(r2^(-1/2)*(y - yRBF(j)));
        end %if
        dLSFx = dLSFx + aRBF(k)*dRBFx;
        dLSFy = dLSFy + aRBF(k)*dRBFy;
        k = k+1;
    end %for j

    dLSFn = sqrt(dLSFx^2 + dLSFy^2);

    if(dLSFn^2 < 1e-10)
        u = 0;
        v = 0;
    else
        u = dLSFy/dLSFn;
        v =-dLSFx/dLSFn;
    end %if

    dydx = [u; v];

end
```

```
%function dLSF_tot

%------------------------------------------------------------------------
```

```
function dphi = grad_phi(  xy   , ...
                           xRBF, ...
                           yRBF, ...
                           SR   , ...
                           aRBF )
    % function grad_phi( ) calculates the gradient vector at a particular
    % point in the level-set field
    %
    % Inputs:
    %                  xy          -> xy-coordinate of the initial point
    %                  xRBF        -> x-coordinates of the RBFs
    %                  yRBF        -> y-coordinates of the RBFs
    %                  SR          -> support radius of the RBFs
    %                  aRBF        -> coefficients of the RBFs
    %
    % Outputs:
    %                  dphi        -> gradient vector
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti________________________________02/24/20
    x = xy(1);
    y = xy(2);
    %
    % Calculate the normal vectors at the x,y location:
    %
    dLSFx = 0;
    dLSFy = 0;
    k = 1;
    for j = 1 : length(xRBF)
        if(isnan(xRBF(j)))
            continue
        end %if
    %
    % Calculate the radius from the RBF location to the
    % intersection point:
    %
        r2=(x -xRBF(j))^2 + (y -yRBF(j))^2;
        r = sqrt(r2);
    %
    % Check if intersection point is outside of the RBFs support
    % radius:
    %
        dRBFx = 0;
        dRBFy = 0;
        if( r <= SR ) && ( r ~= 0.0 )
            term1 = (1 - (r/SR))^4;
            term2 = (4*(r/SR)+1)*(1 - (r/SR))^3;
            dRBFx = (4/SR)*(term1 - term2)*(r2^(-1/2)*(x - xRBF(j)));
            dRBFy = (4/SR)*(term1 - term2)*(r2^(-1/2)*(y - yRBF(j)));
        end %if
        dLSFx = dLSFx + aRBF(k)*dRBFx;
        dLSFy = dLSFy + aRBF(k)*dRBFy;
        k     = k+1;
    end %for j

    dphi = [dLSFx; dLSFy];

end
%function grad_phi

%------------------------------------------------------------------------
```

```
function x = thomas_algorithm( A   , ...
                               RHS, ...
                               N    )
    % function thomas_algorithm( ) solves a tridiagonal aperiodic or
    % period system or equations using the Thomas algorithm
    %
    % Inputs:
    %                   A           -> square coefficient matrix
    %                   RHS         -> right-hand side for system
    %                   N           -> length of a row/column in the matrix
    %
    % Outputs:
    %                   x           -> solution vector
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti---------------------------------02/24/20

    %
    % Check if tridiagonal system is periodic
    %
    if(A(N,1) == 0 && A(1,N) == 0)      % system is not periodic
    %
    % Regular implementation of Thomas Algorithm:
    % ==> Parse matrix Ap into a, b, c, and d bins:
    %
        a = zeros(N,1);
        b = zeros(N,1);
        c = zeros(N,1);
        d = zeros(N,1);

        for i = 1 : N
            if(i > 1 && i < N)
                a(i) = A(i, i-1);
                b(i) = A(i, i   );
                c(i) = A(i, i+1);
                d(i) = RHS(i);
            elseif(i == 1    )
                b(i) = A(i, i   );
                c(i) = A(i, i+1);
                d(i) = RHS(i);
            elseif(i == N)
                a(i) = A(i, i-1);
                b(i) = A(i, i   );
                d(i) = RHS(i);
            end %if
        end %for i
    %
    % ==> Solve systems of equations using the Thomas algorithm
    %                             Ax = RHS
    %
    % Thomas algorithm:
    %
        x   = zeros(N,1);
    %
    % ==> Forward elimination
    %
        for k = 2 : N
            m      = a(k)/b(k-1);
            b(k) = b(k) - m * c(k-1);
            d(k) = d(k) - m * d(k-1);
        end %for k
    %
    % ==> Backward substitution
    %
        x(N) = d(N)/b(N);
        for k = N-1 : -1 : 1
            x(k) = (d(k) - c(k)*x(k+1))/b(k);
        end %for k
```

```
elseif(A(N,1) ~= 0 && A(1,N) ~= 0) % system is       periodic
%
% Periodic implementation of Thomas Algorithm:
% ==> Modify A using the Sherman-Morris forumla as,
%                  Ap  = (A - uv')
%      where u' = [-b1 0 ... cn] and v' = [1 0 ... 0 -a1/b1].
%
    u        = zeros(N,1);
    u(    1) =-A(    1,1);
    u(N)     = A(N,1);

    v        = zeros(N,1);
    v(    1) = 1;
    v(N)     =-A(1,N) / A(1,1);

    Ap       = zeros(N,N);
    for i = 1 : N
        for j = 1 : N
            Ap(i,j) = A(i,j) - u(i)*v(j);
        end %for j
    end %for i
%
% ==> Parse matrix Ap into a, b, c, and d bins:
%
    a = zeros(N,1);
    b = zeros(N,1);
    c = zeros(N,1);
    d = zeros(N,1);

    for i = 1 : N
        if(i > 1 && i < N)
            a(i) = Ap(i, i-1);
            b(i) = Ap(i, i   );
            c(i) = Ap(i, i+1);
            d(i) = RHS(i);
        elseif(i == 1    )
            b(i) = Ap(i, i   );
            c(i) = Ap(i, i+1);
            d(i) = RHS(i);
        elseif(i == N)
            a(i) = Ap(i, i-1);
            b(i) = Ap(i, i   );
            d(i) = RHS(i);
        end %if
    end %for i
%
% ==> Solve two systems of equations using the Thomas algorithm
%                  Ap * q = u,          Ap * y = d
%
% Thomas algorithm:
% (Store parsed matrix variables for repeated use)
% Solve Ap * q = u,
%
    aq = a;
    bq = b;
    cq = c;
    dq = u;
    q  = zeros(N,1);
%
% ==> Forward elimination
%
    for k = 2 : N
        m     = aq(k)/bq(k-1);
        bq(k) = bq(k) - m * cq(k-1);
        dq(k) = dq(k) - m * dq(k-1);
    end %for k
%
% ==> Backward substitution
```

```
    %
        q(N) = dq(N)/bq(N);
        for k = N-1 : -1 : 1
            q(k) = (dq(k) - cq(k)*q(k+1))/bq(k);
        end %for k
    %
    % Solve Ap * y = d,
    %
        ay = a;
        by = b;
        cy = c;
        dy = d;
        y  = zeros(N,1);
    %
    % ==> Forward elimination
    %
        for k = 2 : N
            m     = ay(k)/by(k-1);
            by(k) = by(k) - m * cy(k-1);
            dy(k) = dy(k) - m * dy(k-1);
        end %for k
    %
    % ==> Backward substitution
    %
        y(N) = dy(N)/by(N);
        for k = N-1 : -1 : 1
            y(k) = (dy(k) - cy(k)*y(k+1))/by(k);
        end %for k
        % Use q and y to solve for x
        sumy = 0;
        sumq = 0;
        for j = 1 : N
            sumy     = sumy     + v(j) * y(j);
            sumq     = sumq     + v(j) * q(j);
        end %for j
        x    = zeros(N, 1);
        for k = 1 : N
            x(k) = y(k) - q(k) * (sumy)/(1 + sumq);
        end %for k
    end %if
end
%function thomas_algorithm

%-----------------------------------------------------------------------
```

```
function x_dot = thomas_algorithm_Dot( A       , ...
                                       A_dot   , ...
                                       RHS     , ...
                                       RHS_dot , ...
                                       N        )
    % function thomas_algorithm_Dot( ) solves a tridiagonal aperiodic or
    % period system or equations using the Thomas algorithm
    %
    % Inputs:
    %                  A          -> square coefficient matrix
    %                  A_dot      -> derivative of square coefficient
    %                                matrix
    %                  RHS        -> right-hand side for system
    %                  RHS_dot    -> derivative of right-hand side for
    %                                system
    %                  N          -> length of a row/column in the matrix
    %
    % Outputs:
    %                  x_dot      -> derivative of the solution vector
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti___________________________________02/24/20

    %
    % Check if tridiagonal system is periodic
    %
    if(A(N,1) == 0 && A(1,N) == 0)     % system is not periodic
    %
    % Regular implementation of Thomas Algorithm:
    % ==> Parse matrix Ap into a, b, c, and d bins:
    %
      a = zeros(N,1);
      b = zeros(N,1);
      c = zeros(N,1);
      d = zeros(N,1);

      a_dot = zeros(N,1);
      b_dot = zeros(N,1);
      c_dot = zeros(N,1);
      d_dot = zeros(N,1);

      for i = 1 : N
          if(i > 1 && i < N)
              a(i) = A(i, i-1);
              b(i) = A(i, i  );
              c(i) = A(i, i+1);
              d(i) = RHS(i);

              a_dot(i) = A_dot(i, i-1);
              b_dot(i) = A_dot(i, i  );
              c_dot(i) = A_dot(i, i+1);
              d_dot(i) = RHS_dot(i);
          elseif(i == 1   )
              b(i) = A(i, i  );
              c(i) = A(i, i+1);
              d(i) = RHS(i);

              b_dot(i) = A_dot(i, i  );
              c_dot(i) = A_dot(i, i+1);
              d_dot(i) = RHS_dot(i);
          elseif(i == N)
              a(i) = A(i, i-1);
              b(i) = A(i, i  );
              d(i) = RHS(i);

              a_dot(i) = A_dot(i, i-1);
              b_dot(i) = A_dot(i, i  );
              d_dot(i) = RHS_dot(i);
```

```
        end %if
    end %for i
%
% ==> Solve systems of equations using the Thomas algorithm
%                           Ax = RHS
%
% Thomas algorithm:
%
    x   = zeros(N,1);
%
% ==> Forward elimination
%
    for k = 2 : N
        m      = a(k)/b(k-1);
        b(k) = b(k) - m * c(k-1);
        d(k) = d(k) - m * d(k-1);

        m_dot     =((a_dot(k)*b(k-1)) - (a(k)*b_dot(k-1)))/b(k-1)^2;
        b_dot(k) = b_dot(k) - (m_dot * c(k-1) + m * c_dot(k-1));
        d_dot(k) = d_dot(k) - (m_dot * d(k-1) + m * d_dot(k-1));
    end %for k
%
% ==> Backward substitution
%
    x(N) = d(N)/b(N);
    x_dot(N) = (d_dot(N)*b(N) - d(N)*b_dot(N))/b(N)^2;
    for k = N-1 : -1 : 1
        x(k)     = (d(k) - c(k)*x(k+1))/b(k);
        x_dot(k) = ((d_dot(k) - (c_dot(k)*x(k+1) + c(k)*x_dot(k+1)))*b(k) -     ...
                    (d(k)     -              c(k)*x(k+1)             )*b_dot(k)) ...
                    /b(k)^2;
    end %for k
elseif(A(N,1) ~= 0 && A(1,N) ~= 0) % system is     periodic
%
% Periodic implementation of Thomas Algorithm:
% ==> Modify A using the Sherman-Morris forumla as,
%                Ap  = (A - uv')
%     where u' = [-b1 0 ... cn] and v' = [1 0 ... 0 -a1/b1].
%
    u        = zeros(N,1);
    u(   1) =-A(   1,1);
    u(N)     = A(N,1);

    v        = zeros(N,1);
    v(   1) = 1;
    v(N)     =-A(1,N) / A(1,1);

    Ap       = zeros(N,N);
    for i = 1 : N
        for j = 1 : N
            Ap(i,j) = A(i,j) - u(i)*v(j);
        end %for j
    end %for i

    u_dot    = zeros(N,1);
    u_dot(1)=-A_dot(1,1);
    u_dot(N)= A_dot(N,1);

    v_dot    = zeros(N,1);
    v_dot(1)= 0;
    v_dot(N)=-(A_dot(1,N)*A(1,1) - A(1,N)*A_dot(1,1)) / A(1,1)^2;

    Ap_dot  = zeros(N,N);
    for i = 1 : N
        for j = 1 : N
            Ap_dot(i,j) = A_dot(i,j) - (u_dot(i)*v(j) + u(i)*v_dot(j));
        end %for j
    end %for i
```

```
%
% ==> Parse matrix Ap into a, b, c, and d bins:
%
    a = zeros(N,1);
    b = zeros(N,1);
    c = zeros(N,1);
    d = zeros(N,1);

    a_dot = zeros(N,1);
    b_dot = zeros(N,1);
    c_dot = zeros(N,1);
    d_dot = zeros(N,1);

    for i = 1 : N
        if(i > 1 && i < N)
            a(i) = Ap(i, i-1);
            b(i) = Ap(i, i  );
            c(i) = Ap(i, i+1);
            d(i) = RHS(i);

            a_dot(i) = Ap_dot(i, i-1);
            b_dot(i) = Ap_dot(i, i  );
            c_dot(i) = Ap_dot(i, i+1);
            d_dot(i) = RHS_dot(i);
        elseif(i == 1   )
            b(i) = Ap(i, i  );
            c(i) = Ap(i, i+1);
            d(i) = RHS(i);

            b_dot(i) = Ap_dot(i, i  );
            c_dot(i) = Ap_dot(i, i+1);
            d_dot(i) = RHS_dot(i);
        elseif(i == N)
            a(i) = Ap(i, i-1);
            b(i) = Ap(i, i  );
            d(i) = RHS(i);

            a_dot(i) = Ap_dot(i, i-1);
            b_dot(i) = Ap_dot(i, i  );
            d_dot(i) = RHS_dot(i);
        end %if
    end %for i
%
% ==> Solve two systems of equations using the Thomas algorithm
%                Ap * q = u,        Ap * y = d
%
% Thomas algorithm:
% (Store parsed matrix variables for repeated use)
% Solve Ap * q = u,
%
    aq = a;
    bq = b;
    cq = c;
    dq = u;
    q  = zeros(N,1);

    aq_dot = a_dot;
    bq_dot = b_dot;
    cq_dot = c_dot;
    dq_dot = u_dot;
    q_dot  = zeros(N,1);
%
% ==> Forward elimination
%
    for k = 2 : N
        m     = aq(k)/bq(k-1);
        bq(k) = bq(k) - m * cq(k-1);
        dq(k) = dq(k) - m * dq(k-1);
        m_dot = (aq_dot(k)*bq(k-1) - aq(k)*bq_dot(k-1))/bq(k-1)^2;
        bq_dot(k) = bq_dot(k) - (m_dot * cq(k-1) + m * cq_dot(k-1));
        dq_dot(k) = dq_dot(k) - (m_dot * dq(k-1) + m * dq_dot(k-1));
    end %for k
%
% ==> Backward substitution
%
    q(N) = dq(N)/bq(N);
    q_dot(N) = (dq_dot(N)*bq(N) - dq(N)*bq_dot(N))/bq(N)^2;
    for k = N-1 : -1 : 1
        q(k) = (dq(k) - cq(k)*q(k+1))/bq(k);
        q_dot(k) = ((dq_dot(k) - (cq_dot(k)*q(k+1) + cq(k)*q_dot(k+1)))*bq(k) -     ...
                    (dq(k)     -                 cq(k)*q(k+1)             )*bq_dot(k)) ...
                    /bq(k)^2;
    end %for k
%
% Solve Ap * y = d,
%
    ay = a;
    by = b;
    cy = c;
    dy = d;
    y  = zeros(N,1);

    ay_dot = a_dot;
    by_dot = b_dot;
    cy_dot = c_dot;
    dy_dot = d_dot;
    y_dot  = zeros(N,1);
%
% ==> Forward elimination
%
    for k = 2 : N
        m     = ay(k)/by(k-1);
        by(k) = by(k) - m * cy(k-1);
        dy(k) = dy(k) - m * dy(k-1);

        m_dot     = (ay_dot(k)*by(k-1) - ay(k)*by_dot(k-1))/by(k-1)^2;
        by_dot(k) = by_dot(k) - (m_dot * cy(k-1) + m * cy_dot(k-1));
        dy_dot(k) = dy_dot(k) - (m_dot * dy(k-1) + m * dy_dot(k-1));
    end %for k
%
% ==> Backward substitution
%
    y(N) = dy(N)/by(N);
    y_dot(N) = (dy_dot(N)*by(N) - dy(N)*by_dot(N))/by(N)^2;
    for k = N-1 : -1 : 1
        y(k) = (dy(k) - cy(k)*y(k+1))/by(k);
        y_dot(k) = ((dy_dot(k) - (cy_dot(k)*y(k+1) + cy(k)*y_dot(k+1)))*by(k) -      ...
                    (dy(k)     -                 cy(k)*y(k+1)             )*by_dot(k))   ...
                   /by(k)^2;
    end %for k
%
% Use q and y to solve for x
%
    x          = zeros(N, 1);
    x_dot      = zeros(N, 1);
    for k = 1 : N
        sumy     = 0;
        sumq     = 0;
        sumy_dot = 0;
        sumq_dot = 0;
        for j = 1 : N
            sumy     = sumy     + v(j) * y(j);
            sumq     = sumq     + v(j) * q(j);
            sumy_dot = sumy_dot +(v_dot(j) * y(j) + v(j) * y_dot(j));
            sumq_dot = sumq_dot +(v_dot(j) * q(j) + v(j) * q_dot(j));
            end %for j
            x(k) = y(k) - q(k) * (sumy)/(1 + sumq);
            x_dot(k) = y_dot(k) - ((q_dot(k)*(sumy) + q(k)*(sumy_dot)) * (1 + sumq) - ...
                                                q(k)*(sumy)               * (sumq_dot))   ...
                                        /(1 + sumq)^2;
        end %for k
    end %if

end
%function thomas_algorithm_Dot

%-----------------------------------------------------------------------
```

```
function [ xy_par, ...
           tpar  , ...
           d2xy    ] = spline_fit( poly_fit, ...
                                   xyTop    , ...
                                   mpts      )
    % function spline_fit( ) fits a linear or cubic spline to the curve
    % using the spline points input
    %
    % Inputs:
    %                  poly_fit    -> fit type
    %                  xyTop       -> spline points along the boundaries
    %                  mpts        -> number of points to place along the
    %                                 spline segments
    %
    % Outputs:
    %                  xy_par      -> xy-coordinates of the parametric
    %                                 curve generated around the spline
    %                  tpar        -> parametric variable along the spline
    %                  d2xy        -> curvature information at each spline
    %                                 point
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    npts  = length(xyTop)/2;
    xTop  = zeros(npts,1);
    yTop  = zeros(npts,1);
    dt    = zeros(npts,1);
    for i = 1 : npts
        if(isnan(xyTop(2*i-1)))
            npts = npts-1;
            continue
        end %if
        xTop(i) = xyTop(2*i-1);
        yTop(i) = xyTop(2*i  );
    end %for i

    for i = 1 : npts
        ip1 = i+1;

        if(i == npts)
            ip1 = 1;
        end %if

        dt(i) = sqrt((xTop(ip1) - xTop(i))^2 + (yTop(ip1) - yTop(i))^2);
    end %for i

    if(poly_fit == 1)
        yp  = zeros(npts*mpts,1);
        xp  = zeros(npts*mpts,1);
        t   = zeros(npts*mpts,1);
        tpar= zeros(npts,1);

        for j = 1 : npts
            jp1 = j+1;

            if(j == npts)
                jp1 = 1;
            end %if
            tpar(jp1) = tpar(j) + dt(j);
            tcurr = linspace(tpar(j), tpar(jp1), mpts+1);
            A = (tpar(jp1) - tcurr(1:mpts))./(tpar(jp1) - tpar(j));
            B = 1 - A;
            t (1 + mpts*(j-1) : mpts*j ) = tcurr(1:mpts);
            xp(1 + mpts*(j-1) : mpts*j ) = A*xTop(j) + B*xTop(jp1);
            yp(1 + mpts*(j-1) : mpts*j ) = A*yTop(j) + B*yTop(jp1);
        end %for j
        d2x = zeros(npts,1);
        d2y = zeros(npts,1);
    elseif(poly_fit == 3 || poly_fit == 2)
        A = zeros(npts, npts);
        bx= zeros(npts,     1);
        by= zeros(npts,     1);
        for i = 1 : npts
            im1 = i-1;
            ip1 = i+1;

            if(i == 1)
                im1 = npts;
            elseif(i == npts)
                ip1 = 1;
            end %if
            A(i,im1) = (         dt(im1))/6;
            A(i,i  ) = (dt(i  )+dt(im1))/3;
            A(i,ip1) = (dt(i  )         )/6;

            by(i)    = (yTop(ip1) - yTop(i  ))/(dt(i  )) -...
                       (yTop(i  ) - yTop(im1))/(dt(im1));

            bx(i)    = (xTop(ip1) - xTop(i  ))/(dt(i  )) -...
                       (xTop(i  ) - xTop(im1))/(dt(im1));
        end % for i

        d2x = thomas_algorithm( A    , ...
                                bx   , ...
                                npts );
        d2y = thomas_algorithm( A   , ...
                                by  , ...
                                npts );

        yp  = zeros(npts*mpts,1);
        xp  = zeros(npts*mpts,1);
        t   = zeros(npts*mpts,1);
        tpar= zeros(npts,1);

        for j = 1 : npts
            jp1 = j+1;

            if(j == npts)
                jp1 = 1;
            end %if
            tpar(jp1) = tpar(j) + dt(j);
            tcurr = linspace(tpar(j), tpar(jp1), mpts+1);
            A = (tpar(jp1) - tcurr(1:mpts))./(tpar(jp1) - tpar(j));
            B = 1 - A;
            C = (A .^ 3 - A) ./ 6 * (tpar(jp1) - tpar(j))^2;
            D = (B .^ 3 - B) ./ 6 * (tpar(jp1) - tpar(j))^2;
            t (1 + mpts*(j-1) : mpts*j) =tcurr(1:mpts);
            xp(1 + mpts*(j-1) : mpts*j) =A*xTop(j) + B*xTop(jp1)...
                                        +C* d2x(j) + D* d2x(jp1);
            yp(1 + mpts*(j-1) : mpts*j) =A*yTop(j) + B*yTop(jp1)...
                                        +C* d2y(j) + D* d2y(jp1);
        end %for j
    end %if

    xy_par  = zeros(2*npts*mpts,1);
    for i = 1 : npts*mpts
        xy_par(2*i-1) = xp(i);
        xy_par(2*i  ) = yp(i);
    end %for i
    d2xy = zeros(2*npts      ,1);
    for i = 1 : npts
        d2xy(2*i-1) = d2x(i);
        d2xy(2*i  ) = d2y(i);
    end %for i
end
```

```
%function spline_fit
%------------------------------------------------------------------------
```

```
function [ xy_par   , ...
           xy_dot   , ...
           tpar     , ...
           tpar_dot , ...
           d2xy     , ...
           d2xy_dot ] = spline_fit_Dot( poly_fit  , ...
                                        xyTop     , ...
                                        xyTop_dot , ...
                                        mpts      )
    % function spline_fit_Dot( ) fits a linear or cubic spline to the curve
    % using the spline points input and outputs the derivative information
    % for the points on the curve, the parametric variable, and the
    % curvature.
    %
    % Inputs:
    %                   poly_fit    -> fit type
    %                   xyTop       -> spline points along the boundaries
    %                   xyTop_dot   -> derivative of the spline points
    %                                  along the boundaries w.r.t. the
    %                                  design variables
    %                   mpts        -> number of points to place along the
    %                                  spline segments
    %
    % Outputs:
    %                   xy_par      -> xy-coordinates of the parametric
    %                                  curve generated around the spline
    %                   xy_dot      -> derivative of xy-coordinates of the
    %                                  parametric curve generated around
    %                                  the spline w.r.t. the design
    %                                  variables
    %                   tpar        -> parametric variable along the spline
    %                   tpar_dot    -> derivative of parametric variable
    %                                  along the spline w.r.t. the design
    %                                  variables
    %                   d2xy        -> curvature information at each spline
    %                                  point
    %                   d2xy_dot    -> derivative of curvature information
    %                                  at each spline point w.r.t. the
    %                                  design variables
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    npts      = length(xyTop)/2;
    xTop      = zeros(npts,1);
    xTop_dot  = zeros(npts,1);
    yTop      = zeros(npts,1);
    yTop_dot  = zeros(npts,1);
    dt        = zeros(npts,1);
    dt_dot    = zeros(npts,1);

    for i = 1 : npts
        if(isnan(xyTop(2*i-1)))
            npts = npts-1;
            continue
        end %if
        xTop(i)     = xyTop(2*i-1);
        xTop_dot(i) = xyTop_dot(2*i-1);
        yTop(i)     = xyTop(2*i  );
        yTop_dot(i) = xyTop_dot(2*i  );
    end %for i

    for i = 1 : npts
        ip1 = i+1;

        if(i == npts)
            ip1 = 1;
    end %if

    dt(i)      = sqrt((xTop(ip1) - xTop(i))^2 + (yTop(ip1) - yTop(i))^2);
    dt_dot(i) = ((xTop(ip1) - xTop(i))*(xTop_dot(ip1) - xTop_dot(i)) + ...
                 (yTop(ip1) - yTop(i))*(yTop_dot(ip1) - yTop_dot(i)))/ dt(i);
end %for i

if(poly_fit == 1)
    yp      = zeros(npts*mpts,1);
    xp      = zeros(npts*mpts,1);
    t       = zeros(npts*mpts,1);
    tpar    = zeros(npts,1);
    tpar_dot= zeros(npts,1);

    for j = 1 : npts
        jp1 = j+1;

        if(j == npts)
            jp1 = 1;
        end %if
        tpar(jp1)     = tpar(j)     + dt(j);
        tpar_dot(jp1) = tpar_dot(j) + dt_dot(j);
        tcurr = linspace(tpar(j), tpar(jp1), mpts);
        A = (tpar(jp1) - tcurr)./(tpar(jp1) - tpar(j));
        B = 1 - A;
        t (1 + mpts*(j-1) : mpts*j ) = tcurr;
        xp(1 + mpts*(j-1) : mpts*j ) = A*xTop(j) + B*xTop(jp1);
        yp(1 + mpts*(j-1) : mpts*j ) = A*yTop(j) + B*yTop(jp1);
    end %for j
elseif(poly_fit == 3 || poly_fit == 2)
    A     = zeros(npts, npts);
    bx    = zeros(npts,    1);
    by    = zeros(npts,    1);

    A_dot = zeros(npts, npts);
    bx_dot= zeros(npts,    1);
    by_dot= zeros(npts,    1);

    for i = 1 : npts
        im1 = i-1;
        ip1 = i+1;

        if(i == 1)
            im1 = npts;
        elseif(i == npts)
            ip1 = 1;
        end %if
        A(i,im1)     = (        dt(im1))/6;
        A(i,i  )     = (dt(i  )+dt(im1))/3;
        A(i,ip1)     = (dt(i  )        )/6;

        A_dot(i,im1) = (            dt_dot(im1))/6;
        A_dot(i,i  ) = (dt_dot(i  )+dt_dot(im1))/3;
        A_dot(i,ip1) = (dt_dot(i  )            )/6;

        bx(i)    = (xTop(ip1) - xTop(i  ))/(dt(i  )) -...
                   (xTop(i  ) - xTop(im1))/(dt(im1));

        bx_dot(i)= ((xTop_dot(ip1) - xTop_dot(i  ))*dt(i  ) - ...
                    (xTop(ip1)     - xTop(i  )    )*dt_dot(i  ))/ ...
                   ( dt(i  )^2) ...
                   - ...
                   ((xTop_dot(i  ) - xTop_dot(im1))*dt(im1) - ...
                    (xTop(i  )     - xTop(im1)    )*dt_dot(im1))/ ...
                   ( dt(im1)^2);

        by(i)    = (yTop(ip1) - yTop(i  ))/(dt(i  )) -...
                   (yTop(i  ) - yTop(im1))/(dt(im1));
    by_dot(i)= ((yTop_dot(ip1) - yTop_dot(i   ))*dt(i   ) - ...
                (yTop(ip1)      - yTop(i   )     )*dt_dot(i   ))/ ...
               ( dt(i   )^2) ...
               - ...
               ((yTop_dot(i   ) - yTop_dot(im1))*dt(im1) - ...
                (yTop(i   )     - yTop(im1)     )*dt_dot(im1))/ ...
               ( dt(im1)^2);
end % for i

d2x = thomas_algorithm( A    , ...
                        bx   , ...
                        npts );

d2y = thomas_algorithm( A   , ...
                        by  , ...
                        npts );

d2x_dot = thomas_algorithm_Dot( A      , ...
                                A_dot  , ...
                                bx     , ...
                                bx_dot , ...
                                npts    );

d2y_dot = thomas_algorithm_Dot( A      , ...
                                A_dot  , ...
                                by     , ...
                                by_dot , ...
                                npts    );

yp       = zeros(npts*mpts,1);
xp       = zeros(npts*mpts,1);
t        = zeros(npts*mpts,1);
tpar     = zeros(npts,1);
tpar_dot= zeros(npts,1);

clear A
clear A_dot
for j = 1 : npts
    jp1 = j+1;

    if(j == npts)
        jp1 = 1;
    end %if

    tpar(jp1)     = tpar(j)     + dt(j);
    tpar_dot(jp1) = tpar_dot(j) + dt_dot(j);
    tcurr = linspace(tpar(j), tpar(jp1), mpts+1);
    tcurr_dot = linspace(tpar_dot(j), tpar_dot(jp1), mpts+1);
    A = (tpar(jp1) - tcurr(1:mpts))./(tpar(jp1) - tpar(j));
    B     = 1 - A;
    C     = (A .^ 3 - A) ./ 6 * (tpar(jp1) - tpar(j))^2;
    D     = (B .^ 3 - B) ./ 6 * (tpar(jp1) - tpar(j))^2;
    t (1 + mpts*(j-1) : mpts*j) =tcurr(1:mpts);
    xp(1 + mpts*(j-1) : mpts*j) =A*xTop(j) + B*xTop(jp1)...
                                +C* d2x(j) + D* d2x(jp1);
    yp(1 + mpts*(j-1) : mpts*j) =A*yTop(j) + B*yTop(jp1)...
                                +C* d2y(j) + D* d2y(jp1);

    Anum1 = (tpar_dot(jp1) - tcurr_dot(1:mpts)) .* (tpar(jp1)     - tpar(j)     );
    Anum2 = (tpar(jp1)     - tcurr(1:mpts)     ) .* (tpar_dot(jp1) - tpar_dot(j));
    Aden  = (tpar(jp1) - tpar(j))^2;
    A_dot = (Anum1 - Anum2)./Aden;
    B_dot = -A_dot;
    C_dot = (3*A .^ 2 - 1) .* A_dot ./ 6 * (tpar(jp1) - tpar(j))^2 + ...
            (A .^ 3   - A)          ./ 3 * (tpar(jp1) - tpar(j)) * (tpar_dot(jp1) -
                tpar_dot(j));
    D_dot = (3*B .^ 2 - 1) .* B_dot ./ 6 * (tpar(jp1) - tpar(j))^2  + ...
```

```
                    (B .^ 3    - B)            ./ 3 * (tpar(jp1) - tpar(j)) * (tpar_dot(jp1) -
                        tpar_dot(j));
            xp_dot(1 + mpts*(j-1) : mpts*j) =A_dot*xTop(j   ) + A*xTop_dot(j   ) + ...
                                             B_dot*xTop(jp1) + B*xTop_dot(jp1) + ...
                                             C_dot* d2x(j   ) + C* d2x_dot(j   ) + ...
                                             D_dot* d2x(jp1) + D* d2x_dot(jp1);
            yp_dot(1 + mpts*(j-1) : mpts*j) =A_dot*yTop(j   ) + A*yTop_dot(j   ) + ...
                                             B_dot*yTop(jp1) + B*yTop_dot(jp1) + ...
                                             C_dot* d2y(j   ) + C* d2y_dot(j   ) + ...
                                             D_dot* d2y(jp1) + D* d2y_dot(jp1);
        end %for j

    end %if

    xy_par = zeros(2*npts*mpts,1);
    xy_dot = zeros(2*npts*mpts,1);
    for i = 1 : npts*mpts
        xy_par(2*i-1) = xp(i);
        xy_par(2*i   ) = yp(i);
        xy_dot(2*i-1) = xp_dot(i);
        xy_dot(2*i   ) = yp_dot(i);
    end %for i

    d2xy      = zeros(2*npts      ,1);
    d2xy_dot = zeros(2*npts      ,1);
    for i = 1 : npts
        d2xy(2*i-1)      = d2x(i);
        d2xy(2*i   )      = d2y(i);
        d2xy_dot(2*i-1) = d2x_dot(i);
        d2xy_dot(2*i   ) = d2y_dot(i);
    end %for i
end
%function spline_fit_Dot

%------------------------------------------------------------------------
```

```
function xy_smooth = linear_smoothing( xy_ss , ...
                                       npnt , ...
                                       nbdy , ...
                                       NPASS )
    % function linear_smoothing( ) smooths the input curve using linear
    % smoothing techniques
    %
    % Inputs:
    %                   xy_ss       -> xy-coordinates of boundaries
    %                   npnt        -> number of points in xy_ss array
    %                   nbdy        -> number of boundaries
    %                   NPASS       -> number of passes of linear smoothing
    %
    % Outputs:
    %                   xy_smooth   -> xy-coordinates of the smoothed
    %                                  curve
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti_________________________________02/24/20

    %
    % Do a pass of linear smoothing on stair-step
    %
    xtemp = zeros(npnt,1);
    ytemp = zeros(npnt,1);
    %
    % Determine the beginning and ending i-values for each body:
    %
    ibeg = ones(nbdy,1);
    ifin = zeros(nbdy,1);
    k    = 0;
    for ipnt = 1 : npnt
        if(isnan(xy_ss(2*ipnt-1)) && isnan(xy_ss(2*ipnt )))
            k = k + 1;
            if(k < nbdy)
                ibeg(k+1) = ipnt+1;
                ifin(k)   = ipnt-1;
            elseif(k == nbdy)
                ifin(k) = ipnt-1;
            end %if
        end %if
    end %for i

    if(NPASS > 0)
        for ipass = 1 : NPASS
            k     = 0;
            for ipnt = 1 : npnt-1
                ip1 = ipnt+1;
                im1 = ipnt-1;
                if(isnan(xy_ss(2*ip1-1)) && isnan(xy_ss(2*ip1 )))
                    xtemp(ip1) = NaN;
                    ytemp(ip1) = NaN;
                    k          = k+1;
                    ip1        = ibeg(k);
                elseif(im1 == 0 || (isnan(xy_ss(2*im1-1)) && isnan(xy_ss(2*im1 ))))
                    im1        = ifin(k+1);
                end %if

                xtemp(ipnt) = 0.25 * (xy_ss(2*ip1-1) + 2*xy_ss(2*ipnt-1) + xy_ss(2*im1-1));
                ytemp(ipnt) = 0.25 * (xy_ss(2*ip1  ) + 2*xy_ss(2*ipnt  ) + xy_ss(2*im1  ));
            end %for i

            for ipnt = 1 : npnt
                xy_ss(2*ipnt-1) = xtemp(ipnt);
                xy_ss(2*ipnt  ) = ytemp(ipnt);
            end %for i

            xy_smooth    = zeros(npnt,2);
            for ipnt = 1 : npnt
                xy_smooth(ipnt,1) = xy_ss(2*ipnt-1);
                xy_smooth(ipnt,2) = xy_ss(2*ipnt  );
            end %for i
        end %for ipass
    elseif(NPASS == 0)
        xy_smooth    = zeros(npnt,2);
        for ipnt = 1 : npnt
            xy_smooth(ipnt,1) = xy_ss(2*ipnt-1);
            xy_smooth(ipnt,2) = xy_ss(2*ipnt  );
        end %for i
    end %if
end
%function linear_smoothing

%-----------------------------------------------------------------------
```

```
function xyRBF = RBF_distributor( xy    , ...
                                  nRBF  , ...
                                  npnt  , ...
                                  nbdy   )
    % function RBF_distributor( ) defines the midpoints of the
    % inside/outside RBF pairs
    %
    % Inputs:
    %               xy          -> xy-coordinates of boundaries
    %               nRBF        -> number of inside RBFs
    %               npnt        -> number of points in xy array
    %               nbdy        -> number of boundaries
    %
    % Outputs:
    %               xyRBF       -> xy-coordinates of the midpoints of
    %                              the RBF pairs
    %
    % Written by Jack Rossetti
    % Annontated by Jack Rossetti__________________________________02/24/20

    %
    % Distribute RBFs along the curve using linear interpolation
    % ==> Determine the beginning and ending i-values for each body:
    %
    ibeg = ones(nbdy,1);
    ifin = zeros(nbdy,1);
    k    = 0;
    for i = 1 : npnt
        if(isnan(xy(i,1)) && isnan(xy(i,2)))
            k = k + 1;
            if(k < nbdy)
                ibeg(k+1) = i+1;
                ifin(k)   = i-1;
            elseif(k == nbdy)
                ifin(k) = i-1;
            end %if
        end %if
    end %for i

    xRBF    = zeros(nRBF*nbdy,1);
    yRBF    = zeros(nRBF*nbdy,1);
    tRBF    = zeros(nRBF*nbdy,1);
    for ibdy = 1 : nbdy
        arc  =  0;
        for i = ibeg(ibdy) : ifin(ibdy)
            ip1 = i+1;
            if(isnan(xy(ip1,1)))
                ip1 = ibeg(ibdy);
            end %if
            arc     = arc + sqrt((xy(i,1) - xy(ip1,1))^2 + (xy(i,2) - xy(ip1,2))^2);
        end %for i

        dt   = arc/nRBF;

        for iRBF = 1 + (ibdy-1)*nRBF : ibdy*nRBF
            tRBF(iRBF) = (iRBF - (ibdy-1)*nRBF - 1) * dt;
        end %for iRBF

        xRBF(1 + (ibdy-1) * (nRBF)) = xy(ibeg(ibdy),1);
        yRBF(1 + (ibdy-1) * (nRBF)) = xy(ibeg(ibdy),2);
        iRBF  = 1 + (ibdy-1) * (nRBF);
        tcurr = 0;
        for i = ibeg(ibdy) : ifin(ibdy)
            ip1 = i + 1;
            if(isnan(xy(ip1,1)))
                ip1 = ibeg(ibdy);
            end %if
            Delt  = sqrt((xy(i,1) - xy(ip1,1))^2 + (xy(i,2) - xy(ip1,2))^2);
            tcurr = tcurr + Delt;
            for jj = 1 : 10
                if(tcurr > tRBF(iRBF))
                    A           = (tcurr - tRBF(iRBF))/(Delt);
                    B           = 1 - A;
                    xRBF(iRBF) = A * xy(i,1) + B * xy(ip1,1);
                    yRBF(iRBF) = A * xy(i,2) + B * xy(ip1,2);
                    iRBF        = iRBF + 1;
                    if(iRBF > ibdy *nRBF)
                        break;
                    end %if
                elseif(tcurr < tRBF(iRBF))
                    break;
                end %if
            end %for jj
            if(iRBF > ibdy *nRBF)
                break;
            end %if
        end %for i
    end %for ibdy

    xyRBF = [xRBF, yRBF];

end
%function RBF_distributor

%--------------------------------------------------------------------------
```

```
function [xy_spln , ...
          xy_tpar , ...
          xy_curv , ...
          xy_topo , ...
          nbdy      ...
                  ] = LevelSetSpline( xg       , ...
                                      yg       , ...
                                      xRBF     , ...
                                      yRBF     , ...
                                      aRBF     , ...
                                      SR       , ...
                                      OFFSET   , ...
                                      x_tol    , ...
                                      intol    , ...
                                      tspan    , ...
                                      hstep    , ...
                                      eta_tol  , ...
                                      poly_fit , ...
                                      mpts     , ...
                                      hmin     , ...
                                      hmax     , ...
                                      vmin     , ...
                                      vmax     , ...
                                      DEBUG      )
    % function LevelSetSpline( ) finds the zero-crossing of the level-set
    % function, implements an adaptive RK4 algorithm to march around each
    % acceptable zero-crossing found, and generates a spline about the
    % zero-level-set curve
    %
    % Inputs:
    %                  xg          -> Array of x-values for zero-point
    %                                 identification on one or more
    %                                 level-set curves
    %                  yg          -> Array of y-values for zero-point
    %                                 identification on one or more
    %                                 level-set curves
    %                  xRBF        -> Inital x-coord.    of RBFs
    %                  yRBF        -> Inital y-coord.    of RBFs
    %                  aRBF        -> Inital coefficient of RBFs
    %                  SR          -> Support radius
    %                  OFFSET      -> Offset for LSF calcs
    %                  x_tol       -> Tolerance for the difference between
    %                                 free variables in the zero-point
    %                                 identification algorithm
    %                  intol       -> Tolerance for whether a point is
    %                                 inside/outside an existing boundary
    %                                 in the zero-point identification
    %                                 algorithm
    %                  tspan       -> Range of parametric coordinate for
    %                                 the level-set RK4 algorithm
    %                  hstep       -> Initial step size for the level-set
    %                                 RK4 algorithm
    %                  eta_tol     -> Tolerance used to test whether a
    %                                 step taken by the RK4 algorithm is
    %                  poly_fit    -> Fit type
    %                  mpts        -> number of points along spline
    %                                 segments
    %                  hmin        -> Minimum y-coordinate of the
    %                                 horizontal x-rays
    %                  hmax        -> Maximum y-coordinate of the
    %                                 horizontal x-rays
    %                  vmin        -> Minimum x-coordinate of the
    %                                 vertical x-rays
    %                  vmax        -> Maximum x-coordinate of the
    %                                 vertical x-rays
    %                  DEBUG       -> Indicator for debugging the code:
    %                                 0 - run as normal;
    %                                 1 - debugging output to screen;
```

```
%
% Outputs:
%                   xy_spln     -> array of xy-coordinates for the
%                                  spline points for each body where
%                                  the bodies are separated  by a pair
%                                  of NaNs
%                   xy_tpar     -> array of parametric values along
%                                  each body, separated by NaNs
%                   xy_curv     -> array of curvature information for
%                                  each body, separated by NaNs
%                   xy_topo     -> array of xy-coordinates for the
%                                  refined spline curve between spline
%                                  points for each body where the
%                                  bodies are separated  by a pair
%                                  of NaNs. The number of points placed
%                                  along each spline segment is
%                                  determined within the algorithm as
%                                  the variable mpts
%                   nbdy        -> Number of bodies found by algorithm
%
% Annontated by Jack Rossetti__________________________________02/24/20

xy_spln = [];
xy_tpar = [];
xy_curv = [];
xy_topo = [];

if(DEBUG == 1)
    LSF  = GenerateLSF(  xRBF   , ...
                         yRBF   , ...
                         hmax   , ...
                         hmin   , ...
                         vmax   , ...
                         vmin   , ...
                         aRBF   , ...
                         SR     , ...
                         OFFSET );
    delx = 0.1;
    dely = 0.1;
    xLSF = hmin : delx : hmax;
    yLSF = vmin : dely : vmax;
    [xmesh, ymesh] = meshgrid(xLSF, yLSF);

    figure(1026);
    clf;
    set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
    contourf(xmesh, ymesh, LSF, [0 0])
    axis image
    axis([hmin hmax vmin vmax])
    set(gca, 'FontSize', 20)
    title(sprintf('Zero level-set curve'))
    xlabel('x')
    ylabel('y')
end

for ibdy = 1 : 100
    brk = 0;
    if( (DEBUG == 1         ) && ...
        (~isempty(xy_topo))    )
        figure(1026)
        hold on
        plot(xy_topo(:,1), xy_topo(:,2), 'm--', 'LineWidth', 2);
        hold off
        pause(0.05)
    end %if DEBUG
    [xGrid, yGrid] = meshgrid(xg,yg);
    for j = 1 : length(yg)
        if(DEBUG == 1)
    figure(1026)
    hold on
    w = plot(xGrid(j,1), yGrid(j,1), 'm>');
    hold off
    xlabel('x')
    ylabel('y')
    pause(0.001)
end %if DEBUG
for i = 1 : length(xg)-1
    xlft = xGrid(j,i  );
    xrit = xGrid(j,i+1);
    ypnt = yGrid(j,i  );

    Llft = EvaluateLSF(  xlft   , ...
                         ypnt   , ...
                         xRBF   , ...
                         yRBF   , ...
                         aRBF   , ...
                         SR     , ...
                         OFFSET );

    Lrit = EvaluateLSF(  xrit   , ...
                         ypnt   , ...
                         xRBF   , ...
                         yRBF   , ...
                         aRBF   , ...
                         SR     , ...
                         OFFSET );

    if(Llft * Lrit < 0 && ibdy == 1)
        brk = 1;
        for iter = 1 : 50000
            Llft = EvaluateLSF(  xlft   , ...
                                 ypnt   , ...
                                 xRBF   , ...
                                 yRBF   , ...
                                 aRBF   , ...
                                 SR     , ...
                                 OFFSET );

            xmid= (xrit - xlft)/2 + xlft;

            Lmid = EvaluateLSF(  xmid   , ...
                                 ypnt   , ...
                                 xRBF   , ...
                                 yRBF   , ...
                                 aRBF   , ...
                                 SR     , ...
                                 OFFSET );
            if(Lmid*Llft < 0.0)
                xrit = xmid;
            else
                xlft = xmid;
            end %if
            if(xrit - xlft < x_tol)
                break;
            end %if
        end %for iter
        break;
    elseif(Llft * Lrit < 0 && ibdy > 1)
        for iter = 1 : 50000
            Llft = EvaluateLSF(  xlft   , ...
                                 ypnt   , ...
                                 xRBF   , ...
                                 yRBF   , ...
                                 aRBF   , ...
                                 SR     , ...
                                 OFFSET );
                xmid= (xrit - xlft)/2 + xlft;

                Lmid = EvaluateLSF(   xmid    , ...
                                      ypnt    , ...
                                      xRBF    , ...
                                      yRBF    , ...
                                      aRBF    , ...
                                      SR      , ...
                                      OFFSET );
                if(Lmid*Llft < 0.0)
                    xrit = xmid;
                else
                    xlft = xmid;
                end %if
                if(xrit - xlft < 1e-10)
                    break;
                end %if
            end %for iter
            xrit0= xmid + intol;
            xlft0= xmid - intol;
            ypls = 0.5 * (xrit0 - xlft0);
            inRx = inpolygon_JSR(xy_topo(:,1), ...
                                 xy_topo(:,2), ...
                                 xrit0       , ...
                                 ypnt          );
            inLx = inpolygon_JSR(xy_topo(:,1), ...
                                 xy_topo(:,2), ...
                                 xlft0       , ...
                                 ypnt          );
            inUy = inpolygon_JSR(xy_topo(:,1), ...
                                 xy_topo(:,2), ...
                                 xmid        , ...
                                 ypnt+ypls     );
            inLy = inpolygon_JSR(xy_topo(:,1), ...
                                 xy_topo(:,2), ...
                                 xmid        , ...
                                 ypnt-ypls     );
            inMd = inpolygon_JSR(xy_topo(:,1), ...
                                 xy_topo(:,2), ...
                                 xmid        , ...
                                 ypnt          );
            in   = inpolygon_JSR(xy_topo(:,1), ...
                                 xy_topo(:,2), ...
                                 xmid        , ...
                                 ypnt          );
            if( inRx || ...
                inLx || ...
                inUy || ...
                inLy || ...
                inMd || ...
                in      )
                continue
            elseif( ~inRx && ...
                    ~inLx && ...
                    ~inUy && ...
                    ~inLy && ...
                    ~inMd && ...
                    ~in      )
                brk = 1;
                break;
            end %if
        end %if
    end %for j
    if(DEBUG == 1)
        delete(w)
    end %if DEBUG
    if(brk == 1)
```

```
            break;
        end %if
    end %for i
    if(brk == 0)
        break;
    end %if

    xy0= [xmid, ypnt];
    hp = hstep;
    for icheck = 1 : 1000
        [tn, xyn]= RK4_LSF( tspan       , ...
                            xy0         , ...
                            xRBF        , ...
                            yRBF        , ...
                            SR          , ...
                            aRBF        , ...
                            OFFSET      , ...
                            hp          , ...
                            eta_tol     , ...
                            +1           );
        for i = 1 : 99999
            if(isnan(xyn(i,1)))
                break;
            end %if
        end %for
        ipts = i-1;
        temp = xyn(1:ipts,:);
        clear xyn
        xyn = temp;

        temp = tn(1:ipts,1);
        clear tn
        tn = temp;
        clear temp
%
% Check the end points:
%
        xBOX = [min(xyn(1,1),xyn(2,1)) max(xyn(1,1),xyn(2,1))];
        yBOX = [min(xyn(1,2),xyn(2,2)) max(xyn(1,2),xyn(2,2))];
        if( xBOX(1) < xyn(ipts,1) &&...
            xBOX(2) > xyn(ipts,1) &&...
            yBOX(1) < xyn(ipts,2) &&...
            yBOX(2) > xyn(ipts,2) )
%
% End point crossed over first point
%
            temp = xyn;
            clear xyn
            xyn  = temp(1:ipts-1,:);

            temp = tn;
            clear tn
            tn  = temp(1:ipts-1,:);
            clear temp
        end %if

        cpts  = length(xyn(:,1));
        xyTop = zeros(2*cpts,1);
        for i = 1 : cpts
            xyTop(2*i-1) = xyn(i,1);
            xyTop(2*i  ) = xyn(i,2);
        end %for i

        spln_pts = 3;
        [ xy_par, ...
          tpar  , ...
          d2xy   ] = spline_fit( poly_fit , ...
                                 xyTop    , ...
```

```
                             spln_pts );
    npts  = length(xy_par)/2;
    xpar  = zeros(npts,1);
    ypar  = zeros(npts,1);

    for i = 1 : npts
        xpar(i) = xy_par(2*i-1);
        ypar(i) = xy_par(2*i  );
    end %for i

    max_err = -99999;
    for i = 1 : npts
        phi = EvaluateLSF(xpar(i), ...
                          ypar(i), ...
                          xRBF   , ...
                          yRBF   , ...
                          aRBF   , ...
                          SR     , ...
                          OFFSET  );
        dphi = grad_phi( [xpar(i), ypar(i)], ...
                         xRBF              , ...
                         yRBF              , ...
                         SR                , ...
                         aRBF               );
        deta = -phi /(dphi(1)^2 + dphi(2)^2);
        if(max_err < abs(deta))
            max_err = abs(deta);
            imax    = i;
        end %if
    end %for i

    if(DEBUG == 1)
        figure(123121)
        clf;
        set(gcf, 'unit', 'normalized', 'position', [.1 .025 .55 .85])
        contourf(xmesh, ymesh, LSF, [0 0])
        axis image
        axis([hmin hmax vmin vmax])
        set(gca, 'FontSize', 20)
        title(sprintf('Zero level-set curve'))
        xlabel('x')
        ylabel('y')

        gcf;
        hold on
        plot(xyn(:,1)   , xyn(:,2)   , 'bo' , 'MarkerSize',  8, 'LineWidth', 1.5)
        plot(xpar       , ypar       , 'r--v',                  'LineWidth', 1.5)
        plot(xpar(imax), ypar(imax), 'kx' , 'MarkerSize', 12, 'LineWidth', 1.5)
        w1 = plot(NaN,NaN, 'bo-' );
        w2 = plot(NaN,NaN, 'r--v');
        w3 = plot(NaN,NaN, 'kx'  );
        title(sprintf('Step size, h = %6.4f, max error = %5.3e', hp, max_err))
        legend([w1,w2,w3], 'RK4 points', 'Cubic fit', 'Max Error', 'location', 'best
            ')
        hold off
        axis([-1 1 -1 1])
    end %if DEBUG

    if(max_err < eta_tol)
        break;
    elseif(max_err >= eta_tol)
        hp = 0.50 *(tn(2) - tn(1));
    end %if
end %for icheck

npnt = mpts;
[ xy_par, ...
  tpar  , ...
```

```
          d2xy    ] = spline_fit( poly_fit , ...
                                  xyTop    , ...
                                  npnt      );

        ppts = length(xy_par)/2;
        xy_c  = zeros(ppts,2);
        for ip = 1 : ppts
            xy_c(ip,1) = xy_par(2*ip-1);
            xy_c(ip,2) = xy_par(2*ip  );
        end %for ip

        xy_topo = [ xy_topo; xy_c      ; NaN, NaN];
        xy_spln = [ xy_spln; xyn       ; NaN, NaN];
        xy_tpar = [ xy_tpar; tpar      ; NaN     ];
        xy_curv = [ xy_curv; d2xy      ; NaN; NaN];
    end %for ibdy
    nbdy = ibdy - 1;

end
%function LevelSetSpline

%-----------------------------------------------------------------------
```

```
function [ case_dir , ...
           opt_dir  , ...
           gss_dir  , ...
           int_dir  , ...
           fin_dir    ...
                    ] = SetupSolutionDirectory( icase    , ...
                                                 jgeom    , ...
                                                 poly_fit , ...
                                                 case_num )
    % function SetupSolutionDirectory( ) sets up the directory names for
    % the folders that are used by the algorithm to save data files.
    %
    % Inputs:
    %                   icase       -> initial geometry case number
    %                   jgeom       -> desired geometry case number
    %                   poly_fit    -> fit type
    %                   case_num    -> optimization case number
    %
    % Outputs:
    %                   case_dir    -> case directory for saving files
    %                   opt_dir     -> optimization directory for saving
    %                                  files at each optimization iteration
    %                   gss_dir     -> golden section search directory for
    %                                  saving files at each search
    %                                  iteration
    %                   int_dir     -> directory for saving initial files
    %
    %                   fin_dir     -> directory for saving final files
    %
    % Annontated by Jack Rossetti----------------------------------02/24/20

    if(poly_fit == 1)
        fit_type = 'C0';
    elseif(poly_fit == 2)
        fit_type = 'C20';
    elseif(poly_fit == 3)
        fit_type = 'C2';
    end %if

    if(icase == 1)
        case_name = 'circle';
    elseif(icase == 2)
        case_name = 'vert_ellipses';
    elseif(icase == 3)
        case_name = 'diag_ellipses';
    elseif(icase == 4)
        case_name = 'turbine_blade';
    elseif(icase == 5)
        case_name = 'tblade_array';
    elseif(icase == 6)
        case_name = 'potato';
    elseif(icase == 7)
        case_name = 'ellipse_4_1';
    elseif(icase == 9)
        case_name = 'ellipse_cas';
    end %if

    if(jgeom ~= icase)
        if(jgeom == 1)
            mtch_name = 'circle';
        elseif(jgeom == 2)
            mtch_name = 'two_ellipses';
        elseif(jgeom == 3)
            mtch_name = 'diag_ellipses';
        elseif(jgeom == 4)
            mtch_name = 'turbine_blade';
        elseif(jgeom == 5)
            mtch_name = 'tblade_array';
        elseif(jgeom == 6)
            mtch_name = 'potato';
        elseif(jgeom == 7)
            mtch_name = 'ellipse_4_1';
        elseif(jgeom == 9)
            mtch_name = 'ellipse_cas';
        end %if
    end %if

    if(jgeom == icase)
        case_dir = sprintf('%s/%s/case%03d', case_name, fit_type, case_num);
    elseif(jgeom ~= icase)
        case_dir = sprintf('%s_INTO_%s/%s/case%03d', case_name, mtch_name, fit_type,
            case_num);
    end %if

    opt_dir = sprintf('%s/opt_iterations', case_dir);
    gss_dir = sprintf('%s/gss_iterations', case_dir);
    int_dir = sprintf('%s/initial', case_dir);
    fin_dir = sprintf('%s/final', case_dir);

end
%function SetupSolutionDirectory

%------------------------------------------------------------------
```

```
function [ ] = WriteREADME( case_dir  , ...
                            case_num  , ...
                            icase     , ...
                            jgeom     , ...
                            rect_elli , ...
                            nseed     , ...
                            ngrid     , ...
                            NTRY      , ...
                            NPASS     , ...
                            mRBF      , ...
                            SR        , ...
                            nSR       , ...
                            fSR       , ...
                            OFFSET    , ...
                            inout     , ...
                            dxg       , ...
                            dyg       , ...
                            hstep     , ...
                            tspan     , ...
                            eta_tol   , ...
                            x_tol     , ...
                            intol     , ...
                            mpts      , ...
                            nrays     , ...
                            NRML      , ...
                            poly_fit  , ...
                            hmin      , ...
                            hmax      , ...
                            vmin      , ...
                            vmax      , ...
                            sens_calc , ...
                            comp_calc , ...
                            sd_cg     , ...
                            ctol      , ...
                            dftol     , ...
                            ftol      , ...
                            MAX_ITER  , ...
                            delta     , ...
                            I_tol     , ...
                            DEBUG       )
    % function WriteREADME( ) takes in all the pertainent variables for the
    % optimization run and writes a README.txt file for reference.
    %
    % Inputs:
    %        case_num       -> Case number
    %        icase          -> Initial geom
    %        jgeom          -> Desired geom
    %        rect_elli      -> Initial guess generation,
    %                       -> 0 - rectangle(s)
    %                       -> 1 - ellipse(s)
    %        nseed          -> Seed value for the stair-stepped
    %        ngrid          -> Number of grid cells for stair-stepped
    %        NTRY           -> Number of attempts for the stair-stepped
    %        NPASS          -> Number of passes for the linear smoothing
    %        mRBF           -> Total number of RBFs
    %        SR             -> Support radius
    %        nRBF           -> Number of RBFs divided by two
    %        nSR            -> Number of RBFs affected by each
    %        fSR            -> Fraction of separation of inside/outside RBFs
    %                       -> w.r.t. SR
    %        OFFSET         -> Offset to obtain zero-curve
    %        inout          -> 0 for only inside RBFs,
    %                       -> 1 for inside and outside RBFs
    %        dxg            -> grid resolution to find boundaries
    %        dyg
    %        hstep          -> initial step size for surface point gen.
    %        tspan          -> tspan for the surface point gen.
    %        eta_tol        -> tolerance for distance from zero-curve
%         x_tol          -> tolerance for bisection
%         intol          -> tolerance for inpolygon
%         mpts           -> number of points along each spline segment
%         nrays          -> Number of rays used in objective
%         NRML           -> 0 unscaled objective,
%                        -> 1 scaled   objective
%         poly_fit       -> Fit type for points generator:
%                        -> 1 - linear spline,
%                        -> 2 - cubic polygon (cubic fit with linear
%                        ->     segments)
%                        -> 3 - cubic spline
%         hmin           -> Minimum y-coordinate of the
%                           horizontal x-rays
%         hmax           -> Maximum y-coordinate of the
%                           horizontal x-rays
%         vmin           -> Minimum x-coordinate of the
%                           vertical x-rays
%         vmax           -> Maximum x-coordinate of the
%                           vertical x-rays
%         sens_calc      -> type of derivative calculation:
%                        -> 1 - finite difference,
%                        -> 2 - tangent linear   ,
%                        -> 3 - adjoint mode     ,
%                        -> 4 - complex step
%         comp_calc      -> 1 for all variables,
%                        -> 2 for location     ,
%                        -> 3 for alfa
%         sd_cg          -> 0 for steepest,
%                        -> 1 for conj
%         ctol           -> Tolerance on the norm of gradients
%         dftol          -> Tolerance on the change in objective
%         ftol           -> Tolerance on the value of objective
%         MAX_ITER       -> Maximum number of iterations
%         delta          -> Delta parameter for golden section search
%         I_tol          -> Interval of uncertainty tolerance
%         DEBUG          -> Debugging parameter
%                        -> 0 - Run as normal
%                        -> 1 - Output debugging
%                        -> 2 - Plot   debugging
%
% Outputs:
%
% Annontated by Jack Rossetti_________________________________02/24/20

fpo = 1;
fprintf(fpo,' Geometry generator case numbers: \n'  );
fprintf(fpo,' icase     = %10d\n'    , icase        );
fprintf(fpo,' jgeom     = %10d\n'    , jgeom        );
fprintf(fpo,' rect_elli = %10d\n'    , rect_elli    );
fprintf(fpo,'                                  \n'  );
fprintf(fpo,' Stair-stepper parameters: \n'         );
fprintf(fpo,' iseed     = %10d\n'    , nseed        );
fprintf(fpo,' igrid     = %10d\n'    , ngrid        );
fprintf(fpo,' NTRY      = %10d\n'    , NTRY         );
fprintf(fpo,'                                  \n'  );
fprintf(fpo,' Smoothing parameters: \n'             );
fprintf(fpo,' NPASS     = %10d\n'    , NPASS        );
fprintf(fpo,'                                  \n'  );
fprintf(fpo,' RBF parameters: \n'                   );
fprintf(fpo,' mRBF      = %10d\n'    , mRBF         );
fprintf(fpo,' SR        = %+20.16f\n', SR           );
fprintf(fpo,' nSR       = %+20.16f\n', nSR          );
fprintf(fpo,' fSR       = %+20.16f\n', fSR          );
fprintf(fpo,' OFFSET    = %+20.16f\n', OFFSET       );
fprintf(fpo,' inout     = %10d\n'    , inout        );
fprintf(fpo,'                                  \n'  );
fprintf(fpo,' Surface point generator parameters:\n');
fprintf(fpo,' dxg       = %+20.16f\n', dxg          );
fprintf(fpo,' dyg       = %+20.16f\n', dyg          );
fprintf(fpo,' hstep     = %+20.16f\n', hstep        );
fprintf(fpo,' tspan     = [%+6.4f,  %+6.4f]\n', tspan);
fprintf(fpo,' eta_tol   =  %+16.12e\n', eta_tol     );
fprintf(fpo,' x_tol     = %+20.16f\n', x_tol        );
fprintf(fpo,' intol     = %+20.16f\n', intol        );
fprintf(fpo,' mpts      = %10d\n'    , mpts         );
fprintf(fpo,'                                   \n'  );
fprintf(fpo,' Objective parameters: \n'             );
fprintf(fpo,' nrays     = %10d\n'    , nrays        );
fprintf(fpo,' NRML      = %10d\n'    , NRML         );
fprintf(fpo,' poly_fit  = %10d\n'    , poly_fit     );
fprintf(fpo,' hmin      = %+20.16f\n', hmin         );
fprintf(fpo,' hmax      = %+20.16f\n', hmax         );
fprintf(fpo,' vmin      = %+20.16f\n', vmin         );
fprintf(fpo,' vmax      = %+20.16f\n', vmax         );
fprintf(fpo,'                                   \n'  );
fprintf(fpo,' Sensitivity parameters: \n'           );
fprintf(fpo,' sens_calc = %10d\n'    , sens_calc    );
fprintf(fpo,' comp_calc = %10d\n'    , comp_calc    );
fprintf(fpo,'                                   \n'  );
fprintf(fpo,' Optimization parameters: \n'          );
fprintf(fpo,' sd_cg     = %10d\n'    , sd_cg        );
fprintf(fpo,' ctol      =  %+16.12e\n',ctol         );
fprintf(fpo,' dftol     =  %+16.12e\n',dftol        );
fprintf(fpo,' ftol      =  %+16.12e\n',ftol         );
fprintf(fpo,' MAX_ITER  = %10d\n'    , MAX_ITER     );
fprintf(fpo,'                                   \n'  );
fprintf(fpo,' Golden section search parameters: \n' );
fprintf(fpo,' delta     = %+20.16f\n', delta        );
fprintf(fpo,' I_tol     =  %+16.12e\n', I_tol       );
fprintf(fpo,'                                   \n'  );
fprintf(fpo,' Case Number : %03d\n'   , case_num    );
fprintf(fpo,'                                   \n'  );

if(DEBUG == 0)
    fname = sprintf('./%s/README.txt', case_dir);
    fpo   = fopen(fname, 'w');
    fprintf(fpo,' Geometry generator case numbers: \n'  );
    fprintf(fpo,' icase     = %10d\n'    , icase        );
    fprintf(fpo,' jgeom     = %10d\n'    , jgeom        );
    fprintf(fpo,' rect_elli = %10d\n'    , rect_elli    );
    fprintf(fpo,'                                   \n'  );
    fprintf(fpo,' Stair-stepper parameters: \n'         );
    fprintf(fpo,' iseed     = %10d\n'    , nseed        );
    fprintf(fpo,' igrid     = %10d\n'    , ngrid        );
    fprintf(fpo,' NTRY      = %10d\n'    , NTRY         );
    fprintf(fpo,'                                   \n'  );
    fprintf(fpo,' Smoothing parameters: \n'             );
    fprintf(fpo,' NPASS     = %10d\n'    , NPASS        );
    fprintf(fpo,'                                   \n'  );
    fprintf(fpo,' RBF parameters: \n'                   );
    fprintf(fpo,' nRBF      = %10d\n'    , mRBF         );
    fprintf(fpo,' SR        = %+20.16f\n', SR           );
    fprintf(fpo,' nSR       = %+20.16f\n', nSR          );
    fprintf(fpo,' fSR       = %+20.16f\n', fSR          );
    fprintf(fpo,' OFFSET    = %+20.16f\n', OFFSET       );
    fprintf(fpo,' inout     = %10d\n'    , inout        );
    fprintf(fpo,'                                   \n'  );
    fprintf(fpo,' Surface point generator parameters:\n');
    fprintf(fpo,' dxg       = %+20.16f\n', dxg          );
    fprintf(fpo,' dyg       = %+20.16f\n', dyg          );
    fprintf(fpo,' hstep     = %+20.16f\n', hstep        );
    fprintf(fpo,' tspan     = [%+6.4f,  %+6.4f]\n', tspan);
    fprintf(fpo,' eta_tol   =  %+16.12e\n', eta_tol     );
    fprintf(fpo,' intol     = %+20.16f\n', intol        );
    fprintf(fpo,'                                   \n'  );
    fprintf(fpo,' Objective parameters: \n'             );
```

```
        fprintf(fpo,' nrays     = %10d\n'     , nrays        );
        fprintf(fpo,' NRML      = %10d\n'     , NRML         );
        fprintf(fpo,' poly_fit  = %10d\n'     , poly_fit     );
        fprintf(fpo,' hmin      = %+20.16f\n', hmin          );
        fprintf(fpo,' hmax      = %+20.16f\n', hmax          );
        fprintf(fpo,' vmin      = %+20.16f\n', vmin          );
        fprintf(fpo,' vmax      = %+20.16f\n', vmax          );
        fprintf(fpo,'                                   \n'  );
        fprintf(fpo,' Sensitivity parameters: \n'            );
        fprintf(fpo,' sens_calc = %10d\n'     , sens_calc    );
        fprintf(fpo,' comp_calc = %10d\n'     , comp_calc    );
        fprintf(fpo,'                                   \n'  );
        fprintf(fpo,' Optimization parameters: \n'           );
        fprintf(fpo,' sd_cg     = %10d\n'     , sd_cg        );
        fprintf(fpo,' ctol      =  %+16.12e\n',ctol          );
        fprintf(fpo,' dftol     =  %+16.12e\n',dftol         );
        fprintf(fpo,' ftol      =  %+16.12e\n',ftol          );
        fprintf(fpo,' MAX_ITER  = %10d\n'     , MAX_ITER     );
        fprintf(fpo,'                                   \n'  );
        fprintf(fpo,' Golden section search parameters: \n' );
        fprintf(fpo,' delta     = %+20.16f\n', delta         );
        fprintf(fpo,' I_tol     =  %+16.12e\n', I_tol        );
        fprintf(fpo,'                                   \n'  );
        fprintf(fpo,' Case Number : %03d\n'   , case_num     );
        fprintf(fpo,'                                   \n'  );
        fclose(fpo);
    end %if
end
%function WriteREADME

%----------------------------------------------------------------------
```

```
function [] = write_data(  nRBF          , ...
                           nvar          , ...
                           desparam      , ...
                           npts          , ...
                           dim           , ...
                           xyPoly        , ...
                           comp_calc     , ...
                           gradient      , ...
                           conj_gradient , ...
                           ObjFunc       , ...
                           alpha         , ...
                           beta          , ...
                           bracket_param , ...
                           GSS_param     , ...
                           directory      )
    % function write_data( ) takes in the variables from the optimization
    % run and writes all the data to a file
    %
    % Inputs:
    %                   nRBF         -> number of inside RBFs
    %                   nvar         -> number of variables
    %                   desparam     -> an array containing all the design
    %                                   variable information
    %                   npts         -> number of spline points
    %                   dim          -> number of dimensions in the problem
    %                   xyPoly       -> xy-coordinates of spline for design
    %                   comp_calc    -> number of design variables
    %                   gradient     -> steepest descent gradient
    %                   conj_gradient-> conjugate gradient
    %                   ObjFunc      -> objective function value
    %                   alpha        -> golden section search step
    %                   beta         -> conjugate gradient coefficient
    %                   bracket_param-> golden section search bracketing
    %                                   steps
    %                   GSS_param    -> golden section search converging
    %                                   steps
    %                   directory    -> directory to write savefile to
    %
    % Outputs:
    %
    % Annontated by Jack Rossetti__________________________________02/24/20

    currd = cd;
    mkdir(directory)
    cd(directory)

    % Write data:
    fpo = fopen('desparam.txt', 'w');
    fprintf(fpo, '%d %d\n', nvar, nRBF);
    for i = 1 : nRBF
        for j = 1 : nvar
            fprintf(fpo, ' %+20.16f\n', desparam(nvar*(i-1)+j));
        end %for j
    end %for i
    fclose(fpo);

    fpo = fopen('zero_levelset_xy.txt', 'w');
    fprintf(fpo, '%d %d\n', dim, npts);
    for i = 1 : npts
        for j = 1 : dim
            fprintf(fpo, ' %+20.16f\n', xyPoly(dim*(i-1)+j));
        end %for j
    end %for i
    fclose(fpo);

    dvar = 4 - comp_calc;
    if(~isempty(gradient))
        fpo = fopen('gradient.txt', 'w');
        fprintf(fpo, '%d %d\n', dvar, nRBF);
        for i = 1 : nRBF
            for j = 1 : dvar
                fprintf(fpo, ' %+20.16f\n', gradient(dvar*(i-1)+j));
            end %for j
        end %for i
        fclose(fpo);
    end %if

    if(~isempty(conj_gradient))
        fpo = fopen('conjugate_gradient.txt', 'w');
        fprintf(fpo, '%d %d\n', dvar, nRBF);
        for i = 1 : nRBF
            for j = 1 : dvar
                fprintf(fpo, ' %+20.16f\n', conj_gradient(dvar*(i-1)+j));
            end %for j
        end %for i
        fclose(fpo);
    end %if

    if(~isempty(ObjFunc))
        fpo = fopen('objective_value.txt', 'w');
        fprintf(fpo, ' %+20.16f\n', ObjFunc);
        fclose(fpo);
    end %if

    if(~isempty(alpha))
        fpo = fopen('alpha_value.txt', 'w');
        fprintf(fpo, ' %+20.16f\n', alpha);
        fclose(fpo);
    end %if

    if(~isempty(beta))
        fpo = fopen('beta_value.txt', 'w');
        fprintf(fpo, ' %+20.16f\n', beta);
        fclose(fpo);
    end %if

    if(~isempty(bracket_param))
        fpo = fopen('GSS_bracketing.txt', 'w');
        bpts= bracket_param(1);
        bvar= bracket_param(2);
        fprintf(fpo, '%d %d\n', bvar, bpts);
        for i = 1 : bpts
            for j = 1 : bvar
                fprintf(fpo, ' %+20.16f\n', bracket_param(bvar*(i-1)+j));
            end %for j
        end %for i
        fclose(fpo);
    end %if

    if(~isempty(GSS_param))
        gpts= GSS_param(1);
        gvar= GSS_param(2);
        fpo = fopen('GSS_converging.txt', 'w');
        fprintf(fpo, '%d %d\n', gvar, gpts);
        for i = 1 : gpts
            for j = 1 : gvar
                fprintf(fpo, ' %+20.16f\n', GSS_param(gvar*(i-1)+j));
            end %for j
        end %for i
        fclose(fpo);
    end %if
    cd(currd)
end
%function write_data
%-------------------------------------------------------------------------
```

```
function [xy_ss, ...
          npnt , ...
          nbdy  ] = stair_stepped_representation( ncycle  , ...
                                                   xyPoly  , ...
                                                   ngrid   , ...
                                                   nseed    )
    %
    % Initialize the seed value in MATLAB so the stair-stepped results can
    % be repeated;
    %
    if(~isempty(nseed))
        s = RandStream('mt19937ar','Seed', nseed);
        RandStream.setGlobalStream(s);
    end %if
    %
    % Generate the desired shape and x-rays using the grid spacing:
    %
    npts = length(xyPoly)/2;
    xShap= zeros(npts,1);
    yShap= zeros(npts,1);
    for i = 1 : npts
        xShap(i) = xyPoly(2*i-1);
        yShap(i) = xyPoly(2*i  );
    end %for i

    xmax = -99999;
    xmin = +99999;
    ymax = -99999;
    ymin = +99999;

    for i = 1 : npts
        if(xShap(i) > xmax)
            xmax = xShap(i);
        end %if
        if(xShap(i) < xmin)
            xmin = xShap(i);
        end %if
        if(yShap(i) > ymax)
            ymax = yShap(i);
        end %if
        if(yShap(i) < ymin)
            ymin = yShap(i);
        end %if
    end %for i

    dx    = (xmax - xmin)/(ngrid-1);
    dy    = (ymax - ymin)/(ngrid-1);
    hmin1 = xmin + dx*0.5;
    hmax1 = xmax - dx*0.5;
    vmin1 = ymin + dy*0.5;
    vmax1 = ymax - dy*0.5;

    ho_ve = GetXray(  xyPoly, ...
                      []    , ...
                      []    , ...
                      1     , ...
                      ngrid , ...
                      hmin1 , ...
                      hmax1 , ...
                      vmin1 , ...
                      vmax1  );

    h_rays1 = ones(ngrid+1,1) * NaN;
    d_horz1 = ones(ngrid+1,1) * NaN;
    v_rays1 = ones(ngrid+1,1) * NaN;
    d_vert1 = ones(ngrid+1,1) * NaN;
    ray_type = 1; % indicates the ray being unzipped
    npnt = 0;
```

```
for i = 1 : 2*(ngrid+1)
    if(isnan(ho_ve(2*i-1)) && isnan(ho_ve(2*i)))
        ray_type = 2;
        npnt      = 0; % restart counter
        continue;
    end %if

    if(ray_type == 1)
        npnt        = npnt + 1;
        h_rays1(npnt)= ho_ve(2*i-1);
        d_horz1(npnt)= ho_ve(2*i  );
    elseif(ray_type == 2)
        npnt        = npnt + 1;
        v_rays1(npnt)= ho_ve(2*i-1);
        d_vert1(npnt)= ho_ve(2*i  );
    end %if

end %for i

nrow = ngrid;
ncol = ngrid;

htgt = zeros(nrow+2,1);
vtgt = zeros(ncol+2,1);

htgt(2:nrow+1) = round(d_horz1(1:ngrid) ./ dy);
vtgt(2:ncol+1) = round(d_vert1(1:ngrid) ./ dx);

htemp1 = [h_rays1(1)-dy h_rays1(1:ngrid)' h_rays1(ngrid)+dy h_rays1(ngrid+1)];
htemp2 = [0 d_horz1(1:ngrid)' 0 d_horz1(ngrid+1)];

vtemp1 = [v_rays1(1)-dx v_rays1(1:ngrid)' v_rays1(ngrid)+dx v_rays1(ngrid+1)];
vtemp2 = [0 d_vert1(1:ngrid)' 0 d_vert1(ngrid+1)];

figure(1234)
plot(htemp1, htemp2, 'LineWidth', 2)
xlabel('y')
ylabel('width')
title('Horizontal x-ray')
view(90,270)

figure(2234)
plot(vtemp1, vtemp2, 'LineWidth', 2)
xlabel('x')
ylabel('height')
title('Vertical x-ray')

figure(3234)
bar(htgt)
xlabel('y')
ylabel('width')
title('Discrete Horizontal x-ray')
view(90,270)

figure(4234)
bar(vtgt)
xlabel('x')
ylabel('height')
title('Discrete Vertical x-ray')

figure(5234)
bar(htgt)
hold on
plot(htemp1 ./ dy + 0.5*(ngrid+3), htemp2 ./ dy, 'r--', 'LineWidth', 3)
hold off
xlabel('y')
ylabel('width')
title('Discrete Horizontal x-ray')
```

```
view(90,270)

figure(6234)
bar(vtgt)
hold on
plot(vtemp1 ./ dx + 0.5*(ngrid+3), vtemp2 ./ dx, 'r--', 'LineWidth', 3)
hold off
xlabel('x')
ylabel('height')
title('Discrete Vertical x-ray')
%
% Solve for the geometry:
%
graph = stair_step_generator( htgt   , ...
                              vtgt   , ...
                              nrow   , ...
                              ncol   , ...
                              ncycle );
%
% Print graph paper at end of this temperature
%
fprintf(1,"    ");
for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
    fprintf(1,"%3d", jcol);
end %for jcol
fprintf(1,"\n");
for irow = 1 : nrow+2 %: -1 : 1 %(irow = nrow+1; irow>= 0; irow--) {
    fprintf(1,"%3d", irow);
    for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
        fprintf(1, "%3d", graph(irow, jcol));
    end %for jcol
    fprintf(1,"%3d\n", 0);
end %for irow
fprintf(1,"    ");
for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
    fprintf(1,"%3d", 0);
end %for jcol
fprintf(1,"\n");
x_ss = [];
y_ss = [];
npnt = 0;
for irow = 2 : nrow+1
    for jcol = 2 : ncol+1
        if(graph(irow  , jcol  ) ~= 0 && ...
          (graph(irow+1, jcol  ) == 0 || ...
           graph(irow-1, jcol  ) == 0 || ...
           graph(irow  , jcol+1) == 0 || ...
           graph(irow  , jcol-1) == 0 || ...
           graph(irow+1, jcol-1) == 0 || ...
           graph(irow-1, jcol+1) == 0 || ...
           graph(irow-1, jcol-1) == 0 || ...
           graph(irow+1, jcol+1) == 0))
            npnt       = npnt + 1;
            y_ss(npnt) = (irow - 1.5) * dy + ymin;
            x_ss(npnt) = (jcol - 1.5) * dx + xmin;
        end %if
    end %for jcol
end %for irow

xy_ss = zeros(2*npnt,1);
for i = 1 : npnt
    xy_ss(2*i-1) = x_ss(i);
    xy_ss(2*i  ) = y_ss(i);
end %for i

if (~isempty(graph))
    clear graph;
end %if
```

```
x_ss = zeros(npnt,1);
y_ss = zeros(npnt,1);

for i = 1 : npnt
    x_ss(i) = xy_ss(2*i-1);
    y_ss(i) = xy_ss(2*i  );
end %for i

% Find dx and dy
dx =+99999;
dy =+99999;
for j = 2 : npnt
    dtempx = sqrt((x_ss(1) - x_ss(j))^2);
    dtempy = sqrt((y_ss(1) - y_ss(j))^2);
    if(dtempx < dx && dtempy == 0)
        dx = dtempx;
    end %if
    if(dtempy < dy && dtempx == 0)
        dy = dtempy;
    end %if
end %for j

xtemp = zeros(size(x_ss));
ytemp = zeros(size(y_ss));
ipt   = 0;
opts  = 0;
for i = 1 : npnt
    cnt = 0;
    for j = 1 : npnt
        dtempx = sqrt((x_ss(i) - x_ss(j))^2);
        dtempy = sqrt((y_ss(i) - y_ss(j))^2);
        if(((dtempx-dx)^2 < 1e-6 && dtempy == 0) || ...
           ((dtempy-dy)^2 < 1e-6 && dtempx == 0)     )
            cnt = cnt + 1;
        end %if
    end %for j
    if(cnt < 2)
        opts = opts + 1;
        continue
    end %if
    ipt = ipt+1;
    xtemp(ipt) = x_ss(i);
    ytemp(ipt) = y_ss(i);
end %for i

xxtmp = xtemp(1:npnt-opts);
yytmp = ytemp(1:npnt-opts);

clear xtemp
clear ytemp
clear x_ss
clear y_ss

x_ss  = xxtmp;
y_ss  = yytmp;

clear xxtmp
clear yytmp

npnt = npnt - opts;

figure(2392)
plot(x_ss, y_ss, 'b*')
axis image
axis([min(x_ss) max(x_ss) min(y_ss) max(y_ss)].*1.5)

%--- Walk around the geometry ---%
```

```
    [ xy_ord, ...
      nbdy   ] = ordering_points(   x_ss, ...
                                    y_ss, ...
                                    npnt     );

    figure(2392)
    hold on
    plot(xy_ord(:,1), xy_ord(:,2), 'mo--', 'MarkerSize', 8)
    hold off
    axis image
    axis([min(x_ss) max(x_ss) min(y_ss) max(y_ss)].*1.5)
    pause(0.1)

    clear xy_ss
    npnt     = npnt + nbdy;

    xy_ss = zeros(2*npnt,1);
    for i = 1 : npnt
        xy_ss(2*i-1) = xy_ord(i,1);
        xy_ss(2*i  ) = xy_ord(i,2);
    end %for i
end
%function stair_stepped_representation

%---------------------------------------------------------------------
```

```
function graph = stair_step_generator(  htgt   , ...
                                        vtgt   , ...
                                        nrow   , ...
                                        ncol   , ...
                                        ncycle )
    %
    % function stair_step_generator was developed in collaboration with Dr. John F.
    % Dannenhoffer.
    %
    %   0    1    2    3    4    5    6    7       8    9    10   11   12   13   14   15
    %   ...  ...  ...  ...  ...  ...  ...  ...     ...  ...  ...  ...  ...  ...  ...  ...
    %   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   16   17   18   19   20   21   22   23      24   25   26   27   28   29   30   31
    %   ...  ...  ...  ...  ...  ...  ...  ...     ...  ...  ...  ...  ...  ...  ...  ...
    %   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   32   33   34   35   36   37   38   39      40   41   42   43   44   45   46   47
    %   x..  x..  x..  x..  x..  x..  x..  x..     x..  x..  x..  x..  x..  x..  x..  x..
    %   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   48   49   50   51   52   53   54   55      56   57   58   59   60   61   62   63
    %   x..  x..  x..  x..  x..  x..  x..  x..     x..  x..  x..  x..  x..  x..  x..  x..
    %   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %
    %   64   65   66   67   68   69   70   71      72   73   74   75   76   77   78   79
    %   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.
    %   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   80   81   82   83   84   85   86   87      88   89   90   91   92   93   94   95
    %   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.
    %   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   96   97   98   99   100  101  102  103     104  105  106  107  108  109  110  111
    %   xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
    %   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   112  113  114  115  116  117  118  119     120  121  122  123  124  125  126  127
    %   xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
    %   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %
    %   128  129  130  131  132  133  134  135     136  137  138  139  140  141  142  143
    %   ..x  ..x  ..x  ..x  ..x  ..x  ..x  ..x     ..x  ..x  ..x  ..x  ..x  ..x  ..x  ..x
    %   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   144  145  146  147  148  149  150  151     152  153  154  155  156  157  158  159
    %   ..x  ..x  ..x  ..x  ..x  ..x  ..x  ..x     ..x  ..x  ..x  ..x  ..x  ..x  ..x  ..x
    %   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   160  161  162  163  164  165  166  167     168  169  170  171  172  173  174  175
    %   x.x  x.x  x.x  x.x  x.x  x.x  x.x  x.x     x.x  x.x  x.x  x.x  x.x  x.x  x.x  x.x
    %   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
    %   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
    %
    %   176  177  178  179  180  181  182  183     184  185  186  187  188  189  190  191
    %   x.x  x.x  x.x  x.x  x.x  x.x  x.x  x.x     x.x  x.x  x.x  x.x  x.x  x.x  x.x  x.x
%   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
%   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
%
%
%   192  193  194  195  196  197  198  199     200  201  202  203  204  205  206  207
%   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx
%   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
%   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
%
%   208  209  210  211  212  213  214  215     216  217  218  219  220  221  222  223
%   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx
%   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
%   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
%
%   224  225  226  227  228  229  230  231     232  233  234  235  236  237  238  239
%   xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
%   .x.  .x.  .x.  .x.  .x.  .x.  .x.  .x.     xx.  xx.  xx.  xx.  xx.  xx.  xx.  xx.
%   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
%
%   240  241  242  243  244  245  246  247     248  249  250  251  252  253  254  255
%   xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
%   .xx  .xx  .xx  .xx  .xx  .xx  .xx  .xx     xxx  xxx  xxx  xxx  xxx  xxx  xxx  xxx
%   ...  x..  .x.  xx.  ..x  x.x  .xx  xxx     ...  x..  .x.  xx.  ..x  x.x  .xx  xxx
%
%---------------------------------------------------------------------------------------

%
% Initialize graph and current horizontal/vertical arrays
%
graph = zeros(nrow+2, ncol+2); %(int *) malloc((nrow+2)*(ncol+2)*sizeof(int));
hcur  = zeros(nrow+2,      1); %(int *) malloc((nrow+2)         *sizeof(int));
vcur  = zeros(ncol+2,      1); %(int *) malloc(         (ncol+2)*sizeof(int));
ntry  = nrow^3;
for icycle = 1 : ncycle
%
% The initial graph paper is for all internal boxes to contain
% a part of the body(ies)
%
   for irow = 1 : nrow+2 %(irow = 0; irow < nrow+2; irow++) {
       for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
           if (htgt(irow) > 0 && vtgt(jcol) > 0)
               graph(irow,jcol) = 1;
               hcur (irow     ) = hcur(irow) + 1;
               vcur (     jcol) = vcur(jcol) + 1;
           else
               graph(irow,jcol) = 0;
           end %if
       end %for jcol
   end %for irow
   if(icycle == 1)
%
% compute the objective function
%
       obj = 0;
       for irow = 1 : nrow+2 %(irow = 0; irow < nrow+2; irow++) {
           obj = obj + abs(hcur(irow) - htgt(irow));
       end %for irow
       for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
           obj = obj + abs(vcur(jcol) - vtgt(jcol));
       end %for jcol
       fprintf(1,"initial obj=%d\n", obj);
       fprintf(1,"\n   ");
       for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
           fprintf(1,"%3d", jcol);
       end %for jcol
       fprintf(1,"\n");
       for irow = 1 : nrow+2 %: -1 : 1 %(irow = nrow+1; irow>= 0; irow--) {
           fprintf(1,"%3d", irow);
```

```
            for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
                fprintf(1, "%3d", graph(irow, jcol));
            end %for jcol
            fprintf(1,"%3d\n", htgt(irow)-hcur(irow));
        end %for irow
        fprintf(1,"    ");
        for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
            fprintf(1,"%3d", vtgt(jcol)-vcur(jcol));
        end %for jcol
        fprintf(1,"\n");
        pause(0.1)
    end %if

    for itry = 1 : ntry %(itry = 0; itry < ntry; itry++) {
        irow = mod(randi((1/eps)),nrow) + 2;
        jcol = mod(randi((1/eps)),ncol) + 2;

        mask =   1 * graph(irow-1, jcol-1) + ...
                 2 * graph(irow-1, jcol  ) + ...
                 4 * graph(irow-1, jcol+1) + ...
                 8 * graph(irow  , jcol-1) + ...
                16 * graph(irow  , jcol+1) + ...
                32 * graph(irow+1, jcol-1) + ...
                64 * graph(irow+1, jcol  ) + ...
               128 * graph(irow+1, jcol+1);
%
% only compute change if change retains fact that region is a convex
% hull
%
        if (graph(irow, jcol) ~= 1)
            continue;
        end %if

        if(mask ==  11 || mask ==  23 || mask == 105 || mask == 212 || ...
           mask ==  15 || mask ==  43 || mask == 150 || mask == 232 || ...
           mask ==  22 || mask == 104 || mask == 208 || mask == 240    )
           if (hcur(irow) > htgt(irow) && vcur(jcol) > vtgt(jcol))
                    graph(irow,jcol) = 0;
                    hcur (irow     ) = hcur(irow)-1;
                    vcur (     jcol) = vcur(jcol)-1;
           end %if
        end %if
    end %for itry
%
% print graph paper at end of this cycle
%
    fprintf(1,"\n    ");
    for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
        fprintf(1,"%3d", jcol);
    end %for jcol
    fprintf(1,"\n");
    for irow = 1 : nrow+2 %: -1 : 1 %(irow = nrow+1; irow>= 0; irow--) {
        fprintf(1,"%3d", irow);
        for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
            fprintf(1, "%3d", graph(irow, jcol));
        end %for jcol
        fprintf(1,"%3d\n", htgt(irow)-hcur(irow));
    end %for irow
    fprintf(1,"    ");
    for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
        fprintf(1,"%3d", vtgt(jcol)-vcur(jcol));
    end %for jcol
    fprintf(1,"\n");
%
% remove "sharp" points
%
    for isharp = 1 : 1000
        spnt = 0;
```

```
            brk  = 0;
            for irow = 1 : nrow+2
                for jcol = 1 : ncol+2
                    if(graph(irow, jcol) == 1)
                        if(graph(irow, jcol-1) == 0 && ... % 0 1 0
                           graph(irow, jcol+1) == 0)
                           graph(irow,jcol) = 0;
                           spnt = 1;
                           brk  = 1;
                           break;
                        elseif(graph(irow-1, jcol) == 0 && ... % 0
                               graph(irow+1, jcol) == 0)       % 1
                                                               % 0
                           graph(irow,jcol) = 0;
                           spnt = 1;
                           brk  = 1;
                           break;
                        end %if
                    end %if
                end %for jcol
                if(brk == 1)
                    break
                end %if
            end %for irow
            if(spnt == 0)
                break;
            end %if
        end %for isharp
    %
    % print graph paper after sharp point removal
    %
        fprintf(1,"\n    ");
        for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
            fprintf(1,"%3d", jcol);
        end %for jcol
        fprintf(1,"\n");
        for irow = 1 : nrow+2 %: -1 : 1 %(irow = nrow+1; irow>= 0; irow--) {
            fprintf(1,"%3d", irow);
            for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
                fprintf(1, "%3d", graph(irow, jcol));
            end %for jcol
            fprintf(1,"%3d\n", htgt(irow)-hcur(irow));
        end %for irow
        fprintf(1,"    ");
        for jcol = 1 : ncol+2 %(icol = 0; icol < ncol+2; icol++) {
            fprintf(1,"%3d", vtgt(jcol)-vcur(jcol));
        end %for jcol
        fprintf(1,"\n");
    end %for icycle

% cleanup:
    if (~isempty(htgt))
        clear htgt;
    end %if
    if (~isempty(vtgt))
        clear vtgt;
    end %if
    if (~isempty(hcur))
        clear hcur;
    end %if
    if (~isempty(vcur))
        clear vcur;
    end %if

end
%function stair_step_generator

%--------------------------------------------------------------------------
```

```
function [ xy_ord, ...
           nbdy    ] = ordering_points(  x_ss, ...
                                         y_ss, ...
                                         ii    )

    % Determine dx and dy
    dx = 99999;
    dy = 99999;
    for i = 2 : ii
        tempx = sqrt((x_ss(1) - x_ss(i))^2);
        tempy = sqrt((y_ss(1) - y_ss(i))^2);
        if(tempx < dx && tempx ~= 0)
            dx = tempx;
        end %if
        if(tempy < dy && tempy ~= 0)
            dy = tempy;
        end %if
    end %for i

    jj = ii;
    x_ord = zeros(ii,1);
    y_ord = zeros(ii,1);
    left     = -dx;
    down     = -dy;
    rite     = +dx;
    uupp     = +dy;
    sdir     = [left, down, rite, uupp];
    brk      = 0;
    ibeg     = 1;
    xy_ord   = [ ];
    for nbdy = 1 : 10
        x_ord(ibeg) = x_ss(1);
        y_ord(ibeg) = y_ss(1);
        for i = 1 : 4
            for j = 1 : jj
                if(i == 1 || i == 3) % left or right
                    d1 = (x_ss(j) - x_ord(ibeg));
                    d2 = (y_ss(j) - y_ord(ibeg));
                    if(sqrt((d1-sdir(i))^2) < 1e-6 && sqrt(d2^2) < 1e-6)
                        % take step left or right
                        x_ord(ibeg+1) = x_ss(j);
                        y_ord(ibeg+1) = y_ss(j);
                        tempx      = zeros(jj-1,1);
                        tempy      = zeros(jj-1,1);
                        kk         = 0;
                        for k = 1 : jj
                            if(k ~= j)
                                kk = kk + 1;
                                tempx(kk) = x_ss(k);
                                tempy(kk) = y_ss(k);
                            end %if
                        end %for k
                        clear x_ss
                        clear y_ss
                        x_ss = tempx;
                        y_ss = tempy;
                        clear tempx
                        clear tempy
                        jj   = jj -1;
                        brk = 1;
                        break;
                    end %if
                elseif(i == 2 || i == 4) % down or up
                    d1 = (x_ss(j) - x_ord(ibeg));
                    d2 = (y_ss(j) - y_ord(ibeg));
                    if(sqrt((d2-sdir(i))^2) < 1e-6 && sqrt(d1^2) < 1e-6)
                        % take step down or up
                        x_ord(ibeg+1) = x_ss(j);
                        y_ord(ibeg+1) = y_ss(j);
                        tempx      = zeros(jj-1,1);
                        tempy      = zeros(jj-1,1);
                        kk         = 0;
                        for k = 1 : jj
                            if(k ~= j)
                                kk = kk + 1;
                                tempx(kk) = x_ss(k);
                                tempy(kk) = y_ss(k);
                            end %if
                        end %for k
                        clear x_ss
                        clear y_ss
                        x_ss = tempx;
                        y_ss = tempy;
                        clear tempx
                        clear tempy
                        jj    = jj -1;
                        brk = 1;
                        break;
                    end %if
                end %if
            end %for j
            if(brk == 1)
                brk = 0;
                break;
            end %if
        end %for i

        ip     = ibeg+1;
        im1    = i-1;
        ip1    = i+1;
        if(im1 < 1)
            im1 = 4;
        elseif(ip1 > 4)
            ip1 = 1;
        end %if

        steps = [im1 i ip1];
        for itry = 2 : ii
            for istep = steps
                for j = 1 : jj
                    if(istep == 1 || istep == 3) % left or right
                        d1 = (x_ss(j) - x_ord(ip));
                        d2 = (y_ss(j) - y_ord(ip));
                        if(sqrt((d1-sdir(istep))^2) < 1e-6 && sqrt(d2^2) < 1e-6)
                            ip = ip + 1;
                            % take step left or right
                            x_ord(ip) = x_ss(j);
                            y_ord(ip) = y_ss(j);
                            tempx      = zeros(jj-1,1);
                            tempy      = zeros(jj-1,1);
                            kk         = 0;
                            for k = 1 : jj
                                if(k ~= j)
                                    kk = kk + 1;
                                    tempx(kk) = x_ss(k);
                                    tempy(kk) = y_ss(k);
                                end %if
                            end %for k
                            clear x_ss
                            clear y_ss
                            x_ss = tempx;
                            y_ss = tempy;
                            clear tempx
                            clear tempy
                            jj    = jj -1;
                            brk = 1;
                            break;
                        end %if
                    elseif(istep == 2 || istep == 4) % down or up
                        d1 = (x_ss(j) - x_ord(ip));
                        d2 = (y_ss(j) - y_ord(ip));
                        if(sqrt((d2-sdir(istep))^2) < 1e-6 && sqrt(d1^2) < 1e-6)
                            ip = ip + 1;
                            % take step down or up
                            x_ord(ip) = x_ss(j);
                            y_ord(ip) = y_ss(j);
                            tempx     = zeros(jj-1,1);
                            tempy     = zeros(jj-1,1);
                            kk        = 0;
                            for k = 1 : jj
                                if(k ~= j)
                                    kk = kk + 1;
                                    tempx(kk) = x_ss(k);
                                    tempy(kk) = y_ss(k);
                                end %if
                            end %for k
                            clear x_ss
                            clear y_ss
                            x_ss = tempx;
                            y_ss = tempy;
                            clear tempx
                            clear tempy
                            jj   = jj -1;
                            brk = 1;
                            break;
                        end %if
                    end %if
                end %for j
                if(brk == 1)
                    brk = 0;
                    break;
                end %if
            end %for i

            im1   = istep-1;
            ip1   = istep+1;
            if(im1 < 1)
                im1 = 4;
            elseif(ip1 > 4)
                ip1 = 1;
            end %if
            steps = [im1 istep ip1];

            %--- test point to check if original point recovered ---%
            dtest = (x_ord(ibeg) - x_ord(ip))^2 + (y_ord(ibeg) - y_ord(ip))^2;
            if(dtest < 1e-6)
                ip = ip-1;
                break;
            end %if

          end %for itry

          xy_ord = [xy_ord; x_ord(ibeg:ip,1), y_ord(ibeg:ip,1); NaN, NaN];

          if(isempty(x_ss))
              break;
          elseif(~isempty(x_ss))
              ibeg = ip+2;
          end %if
         end %for nbdy

        end
        %function ordering_points
        %------------------------------------------------------------------------
```

# Bibliography

1. Z. Lyn, G. K. W. Kenway, and J. R. R. A. Martins. Aerodynamics shape optimization investigations of the common research model wing benchmark. *AIAA Journal*, 53(4): 968–985, 2015.

2. E. Becache, A. Chaigne, G. Derveaux, and P. Joly. Numerical simulation of a guitar. *Proc Eur Conf Comput*, 2001.

3. A. Jameson. Aerodynamic design and optimization. *16th AIAA Comput Fluid Dynamics Conf.*, 2003.

4. J. J. Alonso, I. M. Kroo, and A. Jameson. Advanced algorithms for design and optimization of qsp. *AIAA Journal*, 2002.

5. J. Reuther, A. Jameson, J. Farmer, L. Martinelli, and D. Saunders. Aerodynamic shape optimization of complex aircraft configurations via an adjoint formulation. *AIAA Journal*, pages 96–94, 1996.

6. P. Moin, H. Choi, and J. Kim. Turbulent drag reduction: studies of feedback control and flow over riblets. *CTR Rep*, 1992.

7. F. Bassi and S. Rebay. High-order accurate discontinuous finite element solution of the 2d euler equations. *J Computat Phys*, 138(2):251–285, 1997.

8. E. Collins and E. Luke. Evaluation of curved element discontinuous galerkin meshes.

Paper presented at: 44th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit, 2008.

9. M. Hemmerling and P. Meise U. Nether. The generico chair, 2014. URL `http://www.marcohemmerling.com/projects/product/generico.html`.

10. M. P. Bendsøe and N. Kikuchi. Generating optimal topologies in structural design using a homogenization method. *Computat Meth Appl Mech Eng.*, 71(2):197–224, 1988.

11. M. P. Bendsøe. Optimal shape design as a material distribution problem. *Struct Optim.*, 1:193–202, 1989.

12. M. Zhou and G. I. N. Rozvany. The coc algorithm, part ii: topological, geometry, and generalized shape optimization. *Comput Methods Appl Mech Eng.*, 89(1–3):309–336, 1991.

13. H. P. Mlejnek. Some aspects of the genesis of structures. *Struct Optim.*, 5:64–69, 1992.

14. Y. M. Xie and G. P. Steven. A simple evolutionary procedure for structural optimization. *Comp Struct.*, 49(5):885–896, 1992.

15. Y. M. Xie and X. Huang. Recent developments in evolutionary structural optimization (eso) for continuum structures. *Mater Sci Eng.*, 10, 2010.

16. O. M. Querin, G. P. Steven, and Y. M. Xie. Evolutionary structural optimization (eso) using a bidirectional algorithm. *Eng Computat.*, 15(8):1031–1048, 1998.

17. B. Bourdin and A. Chamboulle. Design-dependent loads in topology optimization. *ESAIM Control Optim Calc Var.*, 9:19–49, 2003.

18. J. Sokolowski and A. Zochowski. On the topological derivative in shape optimization. *SAIM J Control Opt.*, 37:1251–1272, 1999.

19. G. Allaire, F. Jouve, and A. M. Toader. A level-set method for shape optimization. *C R Math.*, 334(12):1125–1130, 2002.

20. G. Allaire, F. Jouve, and A. M. Toader. Structural optimization using sensitivity analysis and a level-set method. *J Comput Phys.*, 194(1):363–393, 2004.

21. M. Wang, X. Wang, and D. Guo. A level-set method for structural topology optimization. *Comput Methods Appl Mech Eng.*, 192(1–2):227–246, 2003.

22. T. Yamada, K. Izui, S. Nishiwaki, and A. Takezawa. A topology optimization method based on the level set method incorporating a fictitious interface energy. *Comput Methods Appl Mech Eng.*, 199(45–48):2876–2891, 2010.

23. J. Guest, J. Prevost, and T. Belytschko. Achieving minimum length scale in topology optimization using nodal design variables and projection functions. *Int J Numer Methods Eng*, 61(2):238–254, 2004.

24. O. Sigmund. Morphology-based black and white filters for topology optimization. *Struct Multidiscp Optim*, 33(4–5):401–424, 2007.

25. S. Xu, Y. Cai, and G. Cheng. Volume preserving nonlinear density filter based on heaviside functions. *Struct Multidiscp Optim*, 41:495–505, 2007.

26. F. Wang, B. Lazarov, and O. Sigmund. On projection methods, convergence and robust formulations in topology optimization. *Struct Multidiscp Optim*, 43(6):767–784, 2011.

27. O. Sigmund and J. Petersson. Numerical instabilities in topology optimization: a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. *Struct Optim*, 16(1):68–75, 1998.

28. A. R. Diáz and O. Sigmund. Checkerboard patterns in layout optimization. *Struct Optim*, 10(1):40–45, 1995.

29. C. S. Jog and R. B. Haber. Stability of finite element models for distributed-parameter optimization and topology design. *Comput Methods Appl Mech Eng*, 130(3–4):203–226, 1996.

30. M. Stolpe and M. Bendsøe. Global optima for the zhou-rozvany problem. *Struct Multidiscp Optim*, 43:151–164, 2010.

31. O. Sigmund. On the usefulness of non-gradient approaches in topology optimization. *Struct Multidiscp Optim*, 43(5):589–596, 2011.

32. O. Sigmund and K. Maute. Topology optimization approaches. *Struct Multidiscp Optim*, 48:1031–1055, 2013.

33. M. P. Bendsøe and O. Sigmund. Material interpolation schemes in topology optimization. *Arch Appl Mech*, 69(9–10):635–654, 1999.

34. M. Stolpe and K. Svanberg. An alternative interpolation scheme for minimum compliance optimization. *Struct Multidiscp Optim*, 22(2):116–124, 2001.

35. M. Stolpe and K. Svanberg. On the trajectories of penalization methods for topology optimization. *Struct Multidiscp Optim*, 21:128–139, 2001.

36. O. Sigmund and S. Torquato. Design of materials with extreme thermal expansion using a three-phase topology method. *J Mech Phys Solids*, 45(6):1037–1067, 1997.

37. D. A. Tortorelli T. E. Bruns. Topology optimization of non-linear elastic structures and compliant mechanisms. *Comput Methods Appl Mech Eng*, 190(26–27):3443–3459, 2001.

38. B. Bourdin. Filters in topology optimization. *Int J Numer Methods Eng*, 50(9):2143–2158, 2001.

39. H. A. Eschenauer, V. V. Kobelev, and A. Schumacher. Bubble method for topology and shape optimization of structures. *Struct Optim*, 8:42–51, 1994.

40. A. Novotny, R. Feijoo, E. Taroco, and C. Padra. Topological sensitivity analysis. *Comput Methods Appl Mech Eng*, 192(7–8):803–829, 2003.

41. M. Bonnet and B. Guzina. Sounding of finite solid bodies by way of topological derivative. *Int J Numer Methods Eng*, 61(13):2344–2373, 2004.

42. T. Borrvall and J. Petersson. Topology optimization of fluids in stokes flow. *Int J Numer Methods Fluids*, 41:77–107, 2003.

43. P. Guillaume and K. S. Idris. Topological sensitivity and shape optimization for the stokes equations. *SIAM J Control Optim*, 43(1):1–31, 2004.

44. N. Aage, T. H. Poulsen, and A. Gersborg-Hansen. Topology optimization of large-scale stokes flow problems. *Struct Multidiscp Optim*, 35(2):175–180, 2008.

45. M. Abdelwahed and M. Hassine. Topological optimization method for a geometric control problem in stokes flow. *Appl Numer Math*, 59(8):1823–1838, 2009.

46. J. K. Guest and J. H. Prévost. Topology optimization of creeping fluid flows using darcy-stokes finite element. *Int J Numer Methods Eng*, 66(3):461–484, 2006.

47. J. K. Guest and J. H. Prévost. Design of maximum permeability material structures. *Comput Methods Appl Mech Eng*, 196(4):1006–1017, 2007.

48. A. Gersborg-Hansen, O. Sigmund, and R. B. Haber. Topology optimization of channel flow problems. *Struct Multidiscp Optim*, 30(3):181–192, 2005.

49. A. Gersborg-Hansen, M. Berggren, and B. Dammann. Topology optimization of mass distribution problems in stokes flow. *IUTAM Symposium on Topological Design Optimization of Structures, Machines, and Materials*, pages 365–374, 2006.

50. V. J. Challis and J. K. Guest. Level set topology optimization of fluids in stokes flow. *Int J Numer Methods Eng*, 79(10):1284–1308, 2009.

51. S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. *Jrnl Computat Phys*, 79(1):12–49, 1988.

52. J. A. Sethian. *Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations.* Cambridge University Press, 32 Avenue of the Americas, New York. NY 10013-2473, USA, 1999.

53. M. Gage. Curve shortening makes convex curves circular. *Inventiones Mathematica*, 76:357, 1984.

54. M. Gage and R. Hamilton. The equation shrinking complex planes curves. *J Diff Geom*, 23:69, 1986.

55. M. Grayson. The heat equation shrinks embedded plane curves to round points. *J Diff Geom*, 26:285, 1987.

56. G. Huisken. Flow by mean curvature of convex surfaces into spheres. *J Diff Geom*, 20: 237, 1984.

57. M. Grayson. A short note on the evolution of surfaces via mean curvatures. *J Diff Geom*, 58:555, 1989.

58. J. A. Sethian. Curvature flow and entropy conditions applied to grid generation. *Jrnl Computat Phys*, 115:440–454, 1994.

59. L. Alvarez, P. L. Lions, and M. Morel. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM J Num Anal*, 29(3):845–866, 1992.

60. L. Rudin, S. Osher, and E. Fatemi. Nonlinear total variation-based noise removal algorithms. *Modelisations Matematiques pour le traitement d'images, INRIA*, pages 149–179, 1992.

61. G. Sapiro and A. Tannenbaum. Image smoothing based on affine invariant flow. In *Proc of the Conference on Information Sciences and Systems*. ISS, mar 1993.

62. R. Malladi and J. A. Sethian. Image processing via level set curvature flow. *Proc Natl Acad of Sci*, 92(15):7046–7050, 1995.

63. R. Malladi and J. A. Sethian. Image processing: flows under min/max curvature and mean curvature. *Graphical models and image processing*, 58(2):127–141, 1996.

64. R. Malladi and J. A. Sethian. A unified approach to noise removal, image enhancement, and shape recovery. *IEEE Trans on Image Processing*, 5(11):1554–1568, 1996.

65. R. Malladi and J. A. Sethian. A unified approach for shape segmentation, representation, and recognition. University of Calfornia, Berkeley, 1994.

66. R. Malladi and J. A. Sethian. Level set methods for curvature flow, image enhancement, and shape recovery in medical images. In *Proc of the Conf on Visualization and Mathematics*. Springer-Verlag, jun 1997.

67. R. Malladi, J. A. Sethian, and B. C. Vemuri. A fast level set based algorithm for topology-independent shape modeling. *J Math Imaging and Vision*, 6(2/3):269–290, 1996.

68. R. Malladi and J. A. Sethian. Shape modeling in medical imaging with marching methods. *LBNL-39541*, 1996.

69. R. Malladi and J. A. Sethian. Level set and fast marching methods in image processing and computer vision. In *Proceedings of IEEE International Conference on Image Processing*. Springer-Verlag, sep 1996.

70. R. Malladi, J. A. Sethian, and B. C. Vemuri. Shape modeling with front propagation: a level set approach. *IEEE Trans on Pattern Analysis and Machine Intelligence*, 17(2): 158–175, 1995.

71. R. B. Haber and M. P. Bendsøe. Problem formulation, solution procedures and geometric modeling-key issues in variable-topology optimization. pages 1864–1873. 7th AIAA/USAF/NASA/ISSMO symposium on multidisciplinary analysis and optimization, 1998.

72. J. A. Sethian and A. Wiegmann. Structural boundary deign via level-set and immerse interface methods. *J Comput Phys*, 163(2):489–528, 2000.

73. M. J. De Ruiter and F. Van Keulen. Topology optimization: approaching the material distribution problem using a topological function description. In *Computational techniques for materials, composites, and composite structures*, pages 111–119. Topping BHV, 2000.

74. M. J. De Ruiter and F. Van Keulen. Topology optimization using the topology description function approach. In *4th World congress on structural and multidisciplinary optimization*. G. Cheng, Y. Gu, S. Liu, and Y. Wang, 2001.

75. M. J. De Ruiter and F. Van Keulen. The topological derivative in the topology description function approach. In *Engineering design optimization, product and process improvement*. P. Gosling, 2002.

76. S. J. Osher and F. Santosa. Level-set methods for optimization problems involving geometry and constraints: I. frequencies of a two-density homogeneous drum. *J Comput Phys*, 171(1):272–288, 2001.

77. S. Wang and M. Y. Wang. Radial basis functions and level set method for structural topology optimization. *Int J Numer Methods Eng*, 65(11):1892–1922, 2006.

78. Z. Luo, M. Y. Wang, S. Wang, and P. Wei. A level set-based parameterization method for structural shape and topology optimization. *Int J Numer Methods Eng*, 76(1):1–26, 2008.

79. K. Maute, S. Kreissl, D. Makhija, and R. Yang. Topology optimization of heat conduction in nano-composites. In *9th World congress on structural and multidisciplinary optimization*. World congress, 2011.

80. M. Otomori, T. Yamada, K. Izui, and S. Nishiwaki. Level set-based topology optimization of a compliant mechanism design using mathematical programming. *Mech Sci*, 2 (1):91–98, 2011.

81. N. P. Van Dijk, M. Langelaar, and F. Van Keulen. Explicit level-set-based topology optimization using an exact heaviside function and consistent sensitivity analysis. *Int J Numer Methods Eng*, 91(1):67–97, 2012.

82. W. Zhang, W. Yang, J. Zhou, D. Li, and X. Guo. Structural topology optimization through explicit boundary evolution. *J Appl Math*, 84(1):1–10, 2017.

83. S.-X. Zhu. Compactly supported radial basis functions: how and why? Technical report, King Abdullah University of Science and Technology, 2006.

84. S. Y. Wang, K. M. Lim, B. C. Khoo, and M. Y. Wang. An extended level set method for shape and topology optimization. *J Computat Phys*, 221:395–421, 2007.

85. L. Jiang, S. Chen, and X. Jiao. Parametric shape and topology optimization: A new level set approach based on cardinal basis functions. *International Journal for Numerical Methods in Engineering*, 114(1):66–87, apr 2018. ISSN 10970207. doi: 10.1002/nme.5733. URL `http://doi.wiley.com/10.1002/nme.5733`.

86. P. Wei, Z. Li, X. Li, and M. Y. Wang. An 88-line matlab code for the parameterized level set method based topology optimization using radial basis functions. *Struct Multidisc Optim*, 58:831–849, 2018.

87. N. P. van Dijk, K. Maute, M. Langelaar, and F. Van Keulen. Level-set methods for

structural topology optimization: a review. *Struct Multidisc Optim.*, 48(3):437–472, 2013.

88. G. Pingen, M. Waidmann, A. Evgrafov, and K. Maute. A parametric level-set approach for topology optimization of flow domains. *Struct Multidisc Optim*, 41:117–131, 2010.

89. G. Pingen, A. Evgrafov, and K. Maute. Topology optimization of flow domains using the lattice boltzmann method. *Struct Multidisc Optim*, 34:507–524, 2007.

90. A. Evgrafov, G. Pingen, and K. Maute. Topology optimization of fluid problems by the lattice boltzmann method. In *M. P. Bendsøe, N. Olhoff, O. Sigmund (eds) IUTAM symposium on topological design optimization of structures, machines and materials: status and perspectives.*, pages 559–568. Springer, 2006.

91. S. Kreissl, G. Pingen, and K. Maute. An explicit level set approach for generalized shape optimization of fluids with the lattice bolztmann method. *Int J Numer Meth Fluids*, 65(5):496–519, 2011.

92. S. Kreissl, G. Pingen, A. Evgrafov, and K. Maute. Topology optimization of flexible micro-fluidic devices. *Struct Multidisc Optim*, 42:495–516, 2010.

93. S. Kreissl, G. Pingen, and K. Maute. Topology optimization of unsteady flow. *Int J Numer Meth Engng*, 87:1229–1253, 2011.

94. S. Kreissl and K. Maute. Level set based fluid topology optimization using the extended finite element method. *Struct Multidisc Optim*, 46:311–326, 2012.

95. C. Othmer, E. de Villiers, and H. G. Weller. Implementation of a continuous adjoint for topology optimization of ducted flows. *AIAA Computational Fluids Dynamics*, 2007.

96. C. Othmer. A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows. *Int J Numer Meth Fluids*, 58:861–877, 2008.

97. G. H. Yoon. Topology optimization for turbulent flows with spalart-allmaras model. *Computat Methods Appl Mech Engng*, 303:288–311, 2016.

98. C. B. Dilgen, S. B. Dilgen, D. R. Fuhrman, O. Sigmund, and B. S. Lazarov. Topology optimization of turbulent flows. *Computat Methods Appl Mech Engng*, 311:363–393, 2018.

99. P. D. Dunning and H. A. Kim. A new hole insertion method for level set based structural topology optimization. *Int. J. Numer. Meth. Engng.*, 93:118–134, 2013.

100. L. F. N. Sá, R. C. R. Amigo, A. A. Novotny, and E. C. N. Silva. Topological derivatives applied to fluid flow channel design optimization problems. *Struct. Multidisc. Optim.*, 54:249–264, 2016.

101. H. A. Kim. Topology optimization using the level set method, 2013. Presentation.

102. Z. J. Wang, K. Fidkowski, R. Abgrall, F. Bassi, D. Caraeni, A. Cary, H. Deconinck, R. Hartmann, K. Hillewaert, H. T. Huynh, N. Kroll, G. May, P.-O. Persson, B. van Leer, and M. Visbal. High-order cfd methods: current status and perspective. *Int J Numer Meth Fluids*, 72:811–845, 2013.

103. Z. Zhao, M. Li, L. He, S. Shao, and L. Zhang. High-order curvilinear mesh generation technique based on an improved radius basis function approach. *Int J Numer Meth Fluids*, 91:97–111, 2019.

104. P.-O. Persson and J. Peraire. Curved mesh generation and mesh refinement using lagrangian solid mechanics. *47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2009.

105. C. Johnston and S. Barnes. Development of high-order meshing for industrial aerospace configurations. In *N. Kroll, C. Hirsch, F. Bassi, C. Johnston, K. Hillewaert (eds) IDIHOM: Industrialization of High-Order Methods – A Top-Down Approach: Results*

*of a Collaborative Research Project Funded by the European Union, 2010-2014*, pages 65–78. Springer International Publishing, 2015.

106. R. Schaback. A Practical Guide to Radial Basis Functions. Initial chapter from a book "Scientific Computing with Radial Basis Functions", 2007.

107. R. Schaback and H. Wendland. Using compactly supported radial basis functions to solve partial differential equations. Technical report, WIT Press, 1999. URL `www.witpress.com`.

108. X. He and A. Yildirim J. R.R.A. Martins J. Li, C. A. Mader. Robust aerodynamic shape optimization – from a circle to an airfoil. *Aerospace Science and Technology*, 87: 48–61, 2019.

109. X. Xing, M. Y. Wang, and B. F. Y. Lui. Parametric shape and topology optimization with moving knots radial basis functions and level-set methods. In *7th World congress on structural and multidisciplinary optimization*, 2007.

110. H. S. Ho, B. F.Y. Lui, and M. Y. Wang. Parametric structural optimization with radial basis functions and partition of unity method. *Optim Method Softw*, 26(4–5):533–553, 2011.

111. H. S. Ho, M. Y. Wang, and M. D. Zhou. Parametric structural optimization with dynamic knot rbfs and partitionof unity method. *Struct Multidisc Optim*, 47:353–365, 2013.

112. F. Yee. *Parametric Shape and Topology Structure Optimization with Radial Basis Functions and Level Set Method.* PhD thesis, Department of Automation and Computer-Aided Engineering at the Chinese University of Hong Kong, 2008.

113. R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7:149–160, 1964.

114. M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems 1. *Journal of Research of the National Bureau of Standards*, 49(6), 1952.

115. E. Polak and G. Ribiere. Note sur la convergence de méthodes de directions conjuguées. *Revue Française D'informatique et de Recherche Opérationnelle. Série Rouge*, pages 35–43, 1969. URL `http://www.numdam.org/legal.php`.

116. J. S. Arora. *Introduction to Optimum Design.* Academic Press, 2012. ISBN 9780123813756. doi: 10.1016/C2009-0-61700-1.

117. J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 8(1):308–313, 1965.

118. J. H. Holland. *Adaptation in natural and artificial systems.* A Bradford Book, 1975.

119. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), 1983.

120. E. Russell and J. Kennedy. Particle swarm optimization. *Proceedings of the IEEE international conference on neural networks*, 4, 1995.

121. M. Abramowitz and I. A. Stegun. *Handbook of mathematical functions (Washington: National Bureau of Standards).* New York: Dover, 1964.

122. C. W. Gear. *Numerical initial value problems in ordinary differential equations.* Englewood Cliffs, NJ: Prentice-Hall, 1971.

123. L.H. Thomas. Elliptic problems in linear differential equations over a network, watson sci. comput. lab report. Technical report, Columbia University, New York, 1949.

124. J. Sherman and W.J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Annals of Mathematical Statistics*, 21(1): 124–127, 1950.

125. M. B. Giles and N. A. Pierce. An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion*, 65:393–415, 2000.

126. R. D. Neidinger. Introduction to automatic differentiation and matlab object-oriented programming. *SIAM Review*, 52(3):545–563, 2010.

127. K. Levenberg. An algorithm for least-squares estimation of nonlinear parameters. *Quart Appl Math*, 2:164–168, 1944.

128. D. W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM J Appl Math*, 11:431–441, 1963.

129. D. C. Sorensen. An algorithm for least-squares estimation of nonlinear parameters. *SIAM J Numer Anal*, 19(2):409–426, 1982.

130. J.-D. Müller, H. Deconinck, and P. L. Roe. A frontal approach for node generation in delaunay triangulations. *AGARD R 787*, 1993.

131. T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso. Su2: An open-source suite for multiphysics simulations and design. *AIAA Jrnl.*, 54(3): 828–847, 2016.

132. A. de Boer, M. S. van der Schoot, and H. Bijl. Mesh deformation based on radial basis function interpolation. *Comput Struct.*, 85:784–795, 2007.

133. S. Jakobsson and O. Amoignon. Mesh deformation using radial basis functions for gradient-based aerodynamic shape optimization. *Comput Fluids*, 36:1119–1136, 2007.

134. J. F. Dannenhoffer III and R. Haimes. Design sensitivity calculations directly on cad-based geometry. *AIAA J.*, 2015.

# Jack S. Rossetti

Doctoral Candidate, Mechanical and Aerospace Engineering, Syracuse University
Date of CV: February 2019

238 Link Hall
Syracuse, NY 13210

jsrosset@syr.edu

## Research interests

Data analysis tools applied to fluid dynamics; design optimization techniques; geometry representation for analysis; computational fluid dynamics; design of aerospace vehicles.

## Education

| | | |
|---|---|---|
| 2020 | Ph.D. | Syracuse University, Syracuse, NY, USA<br>Mechanical and Aerospace Engineering<br>*A Method of Topology Optimization for Curvature Continuous Designs*<br>Advisors: John F. Dannenhoffer III and Melissa A. Green |
| 2016 | M.S. | Syracuse University, Syracuse, NY, USA<br>Mechanical and Aerospace Engineering<br>*Snapshot proper orthogonal decomposition of cylinder wake in a moving frame*<br>Advisors: John F. Dannenhoffer III and Melissa A. Green |
| 2014 | B.S. | State University of New York at Buffalo, Buffalo, NY, USA<br>Mechanical and Aerospace Engineering |

## Honors and Awards

| | |
|---|---|
| 2014 – 2018 | Syracuse University Fellowship (Syracuse University) |
| 2020 – 2021 | National Research Council Research Associateship (National Research Council) |

## Teaching experience

Teaching Assistant, Syracuse University

| | | |
|---|---|---|
| 2019 | Spring | Introduction to Aerodynamics (AEE 342) |
| 2017 | Spring | Introduction to Aerodynamics (AEE 342) |
| 2016 | Fall | Introduction to Fluid Dynamics (MAE 341) |
| 2015 | Spring | Introduction to Aerodynamics (AEE 342) |
| 2014 | Fall | Introduction to Fluid Dynamics (MAE 341) |

## Journal publications

2020 1. Rossetti, J. S., Dannenhoffer III, J. F., & Green, M. A. *Snapshot proper orthogonal decomposition of cylinder wake in a moving frame*, AIAA Journal, *in progress*.

### Conference extended abstracts (peer-reviewed and/or invited)

2019 2. Rossetti, J. S., Dannenhoffer III, J. F., & Green, M. A. *A method for topology optimization for high Reynolds number flows*, AIAA Aviation Forum 2019, Dallas, TX, USA. 17–21 June 2019.

2018 3. Rossetti, J. S., Dannenhoffer III, J. F., & Green, M. A. *Using potential flow as a surrogate for high Reynolds number viscous flows*, AIAA Aviation Forum 2018, Atlanta, GA, USA. 25–29 June 2018.

2016 4. Falkenstein-Smith, R., Rossetti, J. S., Garret, M., & Ahn, J. *Investigating the influence of micro-videos used as a supplementary course material*, ASEE 123rd Annual Conference & Exposition 2016, New Orleans, LA, USA. 26–29 June 2016.

2016 5. Rossetti, J. S., Dannenhoffer III, J. F., & Green, M. A. *Snapshot Lagrangian proper orthogonal decomposition of cylinder wake flow*, AIAA Science and Technology Forum 2016, San Diego, CA, USA. 4–8 January 2016.

### Non-refereed abstracts and presentations

2018 6. Rossetti, J., Dannenhoffer III, J. F. & Green, M. A. *Investigation of topology optimization using the level-set method*, 71st Annual Meeting of the APS Division of Fluid Dynamics, Atlanta, GA, USA. 18–20 November 2018.

2017 7. Rossetti, J., & Dannenhoffer III, J. F *Using the level-set method combined with a genetic algorithm for topology optimzation*, 70th Annual Meeting of the APS Division of Fluid Dynamics, Denver, CO, USA. 19–21 November 2017.

2015 8. Rossetti, J. S., Green, M. A., & Dannenhoffer III, J. F. *Lagrangian proper orthogonal decomposition of the wake downstream of a cylinder*, 68th Annual Meeting of the APS Division of Fluid Dynamics, San Francisco, CA, USA. 22–24 November 2015.

2013 9. Rossetti, J. S., Berger, Z. P., Berry, M. G., Hall, A., & Glauser, M. N. *Heater applications for high speed jets*, 66th Annual Meeting of the APS Division of Fluid Dynamics, Pittsburgh, PA, USA. 24–26 November 2013.

## Service

| | |
|---|---|
| 2018 – 2019 | Syracuse University search committee for next dean of the College Engineering and Computer Science |
| 2018 – 2019 | Syracuse University College of Engineering and Computer Science Inclusive Excellence Council |
| 2017 – 2018 | Syracuse University College of Engineering and Computer Science Graduate Student Organization<br>*President* |
| 2014 – pres | Member, American Society of Physics (APS) |
| 2014 – pres | Member, American Institute of Aeronautics and Astronautics (AIAA) |